**Table of Contents – Daily Python Training Report**

|  |  |  |  |
| --- | --- | --- | --- |
| **Day** | **Date** | **Main Topics** | **Subtopics** |
| 1 | 9th June 2025 | Python Basics & Translators | Translator, Compiler vs Interpreter, Python Translation Flow, Text Editor, print() Function, sep & end, REPL vs Script Mode, input(), Variables, id() Function |
| 2 | 10th June 2025 | Data Types & Memory Allocation | Primitive vs Non-Primitive Types, Stack vs Heap, type() Function, Arithmetic, Logical, Comparison & Assignment Operators, Type Conversion, Dynamic Typing |
| 3 | 11th June 2025 | Truthy/Falsy Values & Data Structures | Truthy vs Falsy, List, Tuple, Dictionary, Set, String, Indexing, Slicing, Mutability, Shallow vs Deep Copy, Conditional Statement (if) |
| 4 | 12th June 2025 | Conditional & Looping Constructs | if-else, if-elif-else, Nested If, for and while Loops, range(), Membership Operators, break, continue, pass, f-strings, String & List Methods |
| 5 | 13th June 2025 | Absent – Worksheet Provided | Worksheet 2 Received |
| 6 | 14th June 2025 | Advanced Functions & Git | First-Class Functions, Passing & Returning Functions, Closures, Decorators, Git & GitHub, Git Commands, Importing Files/Functions |
| 7 | 17th June 2025 | Modules, Package Managers & Exception Handling | Modules, Packages, Libraries, pip, conda, uv, poetry, Virtual Environments, CAP Theorem, Exception Handling |
| 8 | 18th June 2025 | Exception Handling (raise), File Handling, JupyterLab & Databases | raise Keyword, File Modes, Using with, JupyterLab UI & Modes, Markdown, DBMS Types, MySQL Basics |
| 9 | 19th June 2025 | Programming Paradigms & SQL Basics | Procedural, Functional & OOP Paradigms, Python Classes, INSERT & SELECT, Constraints, Relationships |
| 10 | 20th June 2025 | Normalization & SQL Table/User Management | Normalization, Armstrong Rules, Closures, Candidate Keys, ALTER, UPDATE, DELETE, User Management, Relationships |
| 11 | 23rd June 2025 | Advanced SQL – Aggregation, Grouping, Subqueries | Aggregate Functions (COUNT, SUM, AVG, MIN, MAX), GROUP BY, HAVING, ORDER BY, Subqueries with syntax and examples |

Day – 1  
Date – 9th June 2025  
Python Basics

**What is a Translator?**

A **translator** is a program that converts code written in **high-level programming languages** (like Python, C, Java) into **machine language** (binary code: 0s and 1s) so that a computer can understand and execute it.

There are **two main types** of translators:

* **Compiler**
* **Interpreter**

**1. Compiler**

**Definition:**

A **compiler** is a translator that **converts the entire high-level source code** into **machine code or bytecode** **at once**, before execution.

**Python Context:**

Python internally uses a compiler to convert code to **bytecode** (not directly machine code), but this is only part of the process.

**2. Interpreter**

**Definition:**

An **interpreter** is a translator that **converts and executes code line by line**, instead of compiling the whole program at once.

**Python Context:**

Python is **interpreted**. It uses an interpreter (like **CPython**) to convert source code to bytecode and then executes it using the Python Virtual Machine (PVM).

**Python Translation Process (Behind the Scenes)**

1. **Source Code (.py)**  
   ↓
2. **Compiler** → Converts it to **Bytecode (.pyc)**  
   ↓
3. **Python Virtual Machine (PVM)** → Executes the bytecode line-by-line (interpreted execution)

**What is a Text Editor?**

A **text editor** is a software application that allows you to **write, edit, and save plain text files**, especially source code in programming languages like Python, C, Java, etc.

For Python programming, text editors are used to **write .py files** and can range from very simple tools to powerful, feature-rich environments.

Examples: Notepad, VScode, PyCharm etc

**print() Function in Python**

The print() function is one of the most **commonly used built-in functions** in Python. It is used to **display output** (text, variables, results of expressions) on the screen.

**REPL in terminal**

**REPL** stands for:

* **Read**: Takes the user input (a line of Python code).
* **Eval**: Evaluates or executes the input.
* **Print**: Displays the result/output of the code.
* **Loop**: Repeats the process, waiting for the next command.

**What is REPL?**

It’s an interactive Python shell that allows you to test and execute Python code **line by line**. It's perfect for beginners and developers to experiment and debug quickly.

>>> print("Hello")

Hello

**Modes of Python Execution**

Python provides **two main modes** for writing and executing code:

**1. Interactive Mode (REPL Mode)  
  
 Open terminal / command prompt**

** Type python or python3 → You’ll see >>> (the prompt)**

2. **Script Mode (File Mode)**

** Save the code in a file, e.g., program.py**

** Run it using:**

**python program.py**

**sep in Python print() Function**

The sep parameter in the print() function **defines the separator** between multiple values.

print(value1, value2, ..., sep='separator')

**By Default:**

Python uses a **space ' '** between items.

print("Python", "is", "fun")

# Output: Python is fun

print("2025", "06", "09", sep="-")

# Output: 2025-06-09

**end in Python print() Function**

The end parameter in the print() function **defines what to print at the end** of the output.

print(value1, value2, ..., end='ending')

print("Hello", end=" ")

print("World")

# Output: Hello World

**input() Function in Python**

The input() function is used to **take input from the user** as a **string** at runtime.

**Syntax:**

variable = input("Your message to the user: ")

name = input("Enter your name: ")

print("Hello,", name)

**Variables in Python**

A **variable** is like a **container** or **name** that stores data in your program so you can use it later.

**Key Point**

* Variables **store values** like numbers, text, etc.
* You **don’t need to declare a data type** (Python is dynamically typed).
* Use **=** to assign a value.

variable\_name = value

**Case Sensitivity in Python**

Python is a **case-sensitive** programming language.  
This means **uppercase and lowercase letters are treated as different**.

Name = "Harnoor"

name = "Kaur"

print(Name) # Output: Harnoor

print(name) # Output: Kaur

**id() Function in Python**

The id() function returns the **unique identity** (memory address) of an object in Python.

id(object)

a = 10

b = 10

print(id(a))

print(id(b))

Day – 2  
Date – 10th June 2025  
Python Basics

1. **Workbook Provided**

On Day 2, we received a detailed workbook containing theory and practice questions focused on fundamental Python concepts. This resource supported our understanding through structured examples and exercises.

1. **Data Types – Primitive and Non-Primitive**

We explored the two broad categories of data types in Python:

* **Primitive Data Types**: These are the most basic data types provided by the language.
* **Non-Primitive Data Types**: These include data structures like lists, tuples, dictionaries, and sets, which are used to store collections of values.

1. **Allocation in RAM – Stack vs Heap**

* **Primitive Data Types** are generally stored in the **stack memory**, which is faster and used for static memory allocation.
* **Non-Primitive Data Types** are stored in the **heap memory**, which allows for dynamic memory allocation and is used when the size or structure of data may change during program execution.

1. **Primitive Data Types**

We studied the following primitive types:

* **Numbers**:
  + int: Represents whole numbers (e.g., 5, -2)
  + float: Represents decimal numbers (e.g., 3.14, -0.001)
  + complex: Represents complex numbers with a real and imaginary part (e.g., 2 + 3j)
* **String** (str): A sequence of characters, enclosed in quotes (e.g., "Hello")
* **Boolean** (bool): Logical values – either True or False
* **NoneType**: Represents the absence of a value using None

1. **Example**

a = 10 # int

b = 3.5 # float

c = 2 + 4j # complex

d = "Python" # string

e = True # boolean

f = None # NoneType

1. **type() Function**

We used the type() function to determine the data type of a variable.

print(type(a))

# Output: <class 'int'>

1. **Operators in Python**

We explored four major types of operators:

* **Arithmetic Operators**: +, -, \*, /, //, %, \*\*
* **Logical Operators**: and, or, not
* **Comparison Operators**: ==, !=, >, <, >=, <=
* **Assignment Operators**: =, +=, -=, \*=, /=, etc.

1. **Example**

x = 5

y = 2

print(x + y) # Arithmetic: 7

print(x > y) # Comparison: True

print(x == y) # Comparison: False

print(x > 1 and y < 5) # Logical: True

x += 3 # Assignment: x becomes 8

1. **input() Function Returns String**

The input() function always returns input as a string, regardless of the value entered.

python

user\_input = input("Enter a number: ")

print(type(user\_input)) # <class 'str'>

1. **Python is Dynamically Typed**

Python does not require explicit declaration of variable types. The type is automatically inferred at runtime based on the assigned value. This flexibility makes Python a dynamically typed language.

a = 10 # int

a = "Hello" # Now a becomes str

1. **Type Conversion**

* **Implicit Type Conversion**: Python automatically converts one data type to another.

a = 10

b = 2.5

result = a + b # result is 12.5 (float)

* **Explicit Type Conversion (Type Casting)**: We manually convert a value from one type to another.

a = "10"

b = int(a) # Now b is an integer

Day – 3  
Date – 11th June 2025  
Python Basics

**1.Truthy and Falsy Values using and / or Operators**

In Python, every value has a truthiness — it is either considered **True** or **False** in logical contexts:

* **Falsy values**: 0, 0.0, '' (empty string), [] (empty list), {} (empty dict), None, False
* **Truthy values**: Almost everything else

**Examples:**

# using and

print(0 and 5) # Output: 0 (Falsy)

print(1 and "Hello") # Output: Hello (Truthy)

# using or

print("" or "Yes") # Output: Yes (Truthy)

print([] or {}) # Output: {} (Both Falsy, returns last)

**2. Primitive Data Types Recap**

We revised the core **primitive data types**:

* int, float, complex
* str (string)
* bool (boolean)
* NoneType

**3. Introduction to Non-Primitive Data Types**

We were introduced to data structures used to store collections:

* **List**: Ordered, mutable collection  
  my\_list = [1, 2, 3]
* **Tuple**: Ordered, immutable collection  
  my\_tuple = (1, 2, 3)
* **String**: Sequence of characters (immutable)  
  my\_str = "Hello"
* **Set**: Unordered, unique elements only  
  my\_set = {1, 2, 3}
* **Dictionary**: Key-value pairs  
  my\_dict = {'name': 'Alice', 'age': 20}

**4. Examples**

list\_ex = [10, 20, 30]

tuple\_ex = (1, 2, 3)

dict\_ex = {"a": 100, "b": 200}

set\_ex = {1, 2, 3}

str\_ex = "Python"

**5. Indexing – Fetching Single Value**

We learned to fetch individual elements using their index:

* **Positive Indexing**: Starts from 0  
  my\_list[0] gives first element
* **Negative Indexing**: Starts from -1 (last element)  
  my\_list[-1] gives last element

**6. Slicing – Fetching Multiple Values**

Used to get subparts of sequences:

my\_list = [10, 20, 30, 40, 50]

print(my\_list[1:4]) # [20, 30, 40]

print(my\_list[:3]) # [10, 20, 30]

print(my\_list[::2]) # [10, 30, 50]

**7. Updating Values**

* **List**: Mutable → can be updated  
  my\_list[1] = 99
* **Tuple**: Immutable → cannot be updated
* **String**: Immutable → cannot be updated
* **Dictionary**: Mutable  
  my\_dict["a"] = 500
* **Set**: Mutable  
  my\_set.add(4)

**8. Mutability and Immutability**

* **Mutable objects**: Can be changed after creation  
  (e.g., lists, dictionaries, sets)
* **Immutable objects**: Cannot be changed  
  (e.g., tuples, strings, integers)

**9. copy() Function & Copying Concepts**

* copy() makes a **shallow copy**: top-level structure is copied, but inner elements still point to same reference.
* **Shallow Copy**: Changes in nested objects reflect in both copies
* **Deep Copy**: Creates entirely new independent copy using copy.deepcopy()

import copy

original = [[1, 2], [3, 4]]

shallow = copy.copy(original)

deep = copy.deepcopy(original)

**10. Conditional Statements – if Statement**

We learned the basic structure of decision-making in Python:

x = 10

if x > 5:

print("x is greater than 5")

Day – 4  
Date – 12th June 2025  
Python Basics

**1. if-else Statement**

Used for conditional execution when a binary decision is required.

age = 18

if age >= 18:

print("Eligible to vote")

else:

print("Not eligible")

**2. if-elif-else Statement**

Allows checking **multiple conditions** sequentially.

marks = 85

if marks >= 90:

print("Grade A")

elif marks >= 75:

print("Grade B")

elif marks >= 60:

print("Grade C")

else:

print("Grade D")

**3. If-Else Ladder / Nested If-Else**

**Nested if-else** means using an if or else block inside another if or else.

num = 10

if num > 0:

if num % 2 == 0:

print("Positive Even")

else:

print("Positive Odd")

else:

print("Non-positive number")

**4. Looping Statements**

Used to repeat a block of code.

**a. for Loop**

Used to iterate over a sequence like list, string, etc.

for i in range(5):

print(i)

**b. while Loop**

Runs as long as a condition is true.

i = 1

while i <= 5:

print(i)

i += 1

**5. range() Function**

Generates a sequence of numbers. Commonly used with loops.

range(5) # 0 to 4

range(2, 6) # 2 to 5

range(1, 10, 2) # 1, 3, 5, 7, 9

**6. Membership Operators**

Used to check if a value is present in a sequence:

* in
* not in

print("a" in "apple") # True

print(5 not in [1, 2, 3]) # True

**7. break, continue, and pass Statements**

* **break**: Exits the loop immediately
* **continue**: Skips the current iteration
* **pass**: Does nothing; acts as a placeholder

for i in range(5):

if i == 3:

continue

print(i)

**8. f-Strings**

Formatted string literals used to embed variables directly inside strings.

name = "Harnoor"

age = 19

print(f"My name is {name} and I am {age} years old.")

**9. String Functions**

We explored several useful built-in string methods:

* upper() – Converts to uppercase
* lower() – Converts to lowercase
* capitalize() – Capitalizes first letter
* join() – Joins elements of a list with a string
* split() – Splits string into a list
* strip() – Removes leading/trailing whitespace
* startswith() – Checks if string starts with given substring
* endswith() – Checks if string ends with given substring

s = " hello world "

print(s.strip().capitalize())

**10. List Functions**

Essential list operations for manipulation:

* append(x) – Adds element to end
* pop() – Removes and returns last element
* copy() – Returns a shallow copy
* reverse() – Reverses the list
* insert(index, x) – Inserts x at index
* remove(x) – Removes first occurrence of x

l = [1, 2, 3]

l.append(4)

l.insert(1, 99)

l.remove(2)

l.reverse()

Day – 5  
Date – 13th June 2025 (Absent)

Worksheet -2 was provided

Day – 6

Date – 14th June 2025

**1. First-Class Functions**

In Python, functions are treated as **first-class citizens**, meaning:

* Functions can be **assigned to variables**
* **Passed as arguments**
* **Returned from other functions**

def greet():

return "Hello"

say\_hello = greet

print(say\_hello()) # Output: Hello

**2. Passing Functions as Arguments**

Functions can be passed to other functions just like variables.

def shout(text):

return text.upper()

def speak(func):

print(func("hello"))

speak(shout) # Output: HELLO

**3. Returning Function Addresses**

A function can return another function’s reference.

def outer():

def inner():

return "Inside Inner"

return inner

func = outer()

print(func()) # Output: Inside Inner

**4. Both Passing and Returning Functions**

Functions can be both passed and returned dynamically.

def outer(func):

def inner():

return func().upper()

return inner

def message():

return "hello"

new\_func = outer(message)

print(new\_func()) # Output: HELLO

**5. Decorators**

Decorators allow us to **modify the behavior** of a function without changing its actual code.

def decorator(func):

def wrapper():

print("Before function call")

func()

print("After function call")

return wrapper

@decorator

def greet():

print("Hello")

greet()

**6. Concept of Closure**

A **closure** occurs when a nested function remembers the values of variables in its enclosing scopes, even after the outer function has finished execution.

def outer(x):

def inner():

return x

return inner

closure\_func = outer(10)

print(closure\_func()) # Output: 10

**7. Git and GitHub**

* **Git**: A **version control system** that helps track changes in source code.
* **GitHub**: A **remote platform** for hosting Git repositories.

**8. Version Control Tools**

Tools like **Git**, **GitHub**, **Bitbucket**, and **GitLab** allow developers to manage versions, collaborate, and track code history.

**9. Git Architecture: Working Space, Staging Area, Commit Area**

* **Working Directory**: Where you write/edit files
* **Staging Area**: Temporarily stores changes using git add
* **Commit Area (Repository)**: Stores final version using git commit

**10. Git Basic Commands**

* git init: Initializes a Git repo
* git status: Shows current file status
* git add <file>: Stages file
* git rm <file>: Removes a file
* git commit -m "message": Saves changes
* git remote add origin <URL>: Links to GitHub
* git push origin main: Uploads code
* git pull origin main: Fetches and merges remote changes
* git fetch: Only fetches, doesn’t merge

**11. Importing Files from the Same Folder**

import myfile # Imports myfile.py from same directory

**12. Importing Functions from the Same Folder**

from myfile import my\_function

my\_function()

**13. Importing Files from Different Folder**

To import from another folder, use:

import sys

sys.path.append("path\_to\_folder")

import myfile

**14. Concept of Packages**

A **package** is a directory containing \_\_init\_\_.py and multiple Python modules. It helps in organizing code.

mypackage/

\_\_init\_\_.py

module1.py

module2.py

Usage:

from mypackage import module1

Day – 7

Date – 17th June 2025

**File Importing in Python**

1. **Importing from the Same Folder:**
   * **Single Function:**  
     Use from filename import function\_name to import a specific function from a Python file in the same directory.
   * **Whole File:**  
     Use import filename to import the entire Python file and access functions using filename.function\_name().
2. **Importing from a Different Folder:**
   * Use Python’s sys.path.append("path/to/folder") or convert the folder into a package using \_\_init\_\_.py and then use the import statement accordingly.

**Modules, Packages, and Libraries**

* **Module:**  
  A single .py file containing Python definitions and functions.
* **Package:**  
  A directory containing a group of related Python modules and an \_\_init\_\_.py file.
* **Library:**  
  A collection of modules and packages offering ready-to-use functionalities for various tasks like web development, data science, AI, etc.

**Package Managers in Python**

* **pip:**  
  The default Python package manager.
  + pip install package-name – installs a package
  + pip list – shows installed packages
  + pip show package-name – displays package details
* **conda:**  
  A package and environment manager often used in data science.
  + Used to manage both Python packages and dependencies for other languages.
* **uv:**  
  A new fast Python package manager.
  + Offers better dependency resolution.
  + Install via: pip install uv
  + Usage includes uv init, uv venv, uv add, uv sync, etc.
* **Poetry:**  
  Used for dependency management and packaging in Python projects.

**Virtual Environment**

* **Creating a Virtual Environment:**  
  python -m venv env\_name
* **Activating the Environment:**
  + Windows: env\_name\Scripts\activate
  + Linux/Mac: source env\_name/bin/activate
* **Managing Packages Inside Venv:**
  + pip list – lists installed packages
  + pip install package-name – installs a package in the venv
* **With uv:**
  + uv init – initializes project
  + uv venv – creates a virtual environment
  + uv add package-name – adds dependency
  + uv sync – syncs packages
  + deactivate – exits the virtual environment
  + activate – reactivates the environment

**Package Dependency**

* Ensures that packages installed in a project don’t conflict with each other.
* Tools like pip, uv, and conda help manage these dependencies properly.

**Software Design Approaches**

* **Monolithic Approach:**  
  All components of an application are tightly integrated into a single program.
* **Distributed Approach:**  
  Application components are split across different services or machines, communicating via APIs or network protocols.

**CAP Principle (CAP Theorem)**

* Describes the limitations in distributed systems:
  + **C**onsistency: Every read receives the most recent write.
  + **A**vailability: Every request gets a (non-error) response.
  + **P**artition Tolerance: System continues to work even when communication between parts is lost.

According to the theorem, a distributed system can only guarantee two of the three properties at the same time.

**Exception Handling in Python**

* **try block**: Contains code that might throw an exception.
* **except block**: Handles known exceptions.
* **else block**: Runs if no exception is raised in try.
* **finally block**: Executes no matter what (used for cleanup tasks).

**Known vs Unknown Exceptions:**

* Known Exception: except ZeroDivisionError:
* Unknown Exception: except Exception as e:

**Custom Exceptions**

* Python allows creating user-defined exceptions by extending the base Exception class:

class MyCustomError(Exception):

pass

* Raise with: raise MyCustomError("Custom error message")

Day – 8

Date – 18th June 2025

**1. Exception Handling in Python – raise Keyword**

**What is raise?**

The raise keyword is used to throw exceptions manually when a specific condition occurs, even if Python hasn't raised it automatically.

**Why use raise?**

To:

* Enforce business logic
* Improve code readability
* Create more predictable error handling

**Syntax:**

raise ExceptionType("Optional custom error message")

**Example:**

def check\_age(age):

if age < 18:

raise ValueError("Age must be at least 18 to register.")

return True

check\_age(16) # Raises ValueError

**Common Use Cases:**

* Validating user input
* Preventing illegal operations
* Raising custom exceptions when needed

**2. File Handling in Python**

File handling enables reading and writing data to/from external files — critical for data persistence, logging, and data manipulation tasks.

**File Opening Syntax:**

file = open("filename.txt", "mode")

**Modes of Opening Files:**

| **Mode** | **Description** |
| --- | --- |
| 'r' | Read (default); file must exist |
| 'w' | Write; creates file or truncates existing |
| 'a' | Append to existing file |
| 'x' | Create; fails if file exists |
| 'r+' | Read & write; file must exist |

**Common File Functions:**

* read(size), readline(), readlines()
* write(string), writelines(list)
* seek(), tell() for cursor control
* close() to free system resources

**Safe Way to Handle Files:**

with open("data.txt", "w") as file:

file.write("Python makes file handling easy!")

Using with automatically closes the file after the block is executed, even if an exception occurs.

**3. Introduction to JupyterLab & Jupyter Notebooks**

**What is JupyterLab?**

JupyterLab is a browser-based IDE for working with notebooks, code, and data. It's a powerful tool for data analysis, AI, and machine learning workflows.

**Key Features:**

* Interactive code execution in cells
* Rich output: plots, images, markdown
* Supports multiple kernels (Python, R, etc.)
* Extensible UI: drag & drop, multiple tabs, terminals

**Jupyter Modes:**

* **Edit Mode (Enter):** Type/edit code in a cell
* **Command Mode (Esc):** Use shortcuts to control notebook structure (e.g., A, B, D, M, Y, etc.)

**Workflow Example:**

1. Create a new notebook
2. Write code in cells
3. Execute with Shift + Enter
4. Add markdown notes using M and render with Shift + Enter

**4. Databases and DBMS (Database Management System)**

**What is a Database?**

A structured collection of data, stored electronically and accessed using software (DBMS).

**Types of Databases:**

* **Relational (RDBMS):** MySQL, PostgreSQL, SQLite (uses tables and SQL)
* **NoSQL:**
  + Document-based (MongoDB)
  + Key-Value (Redis)
  + Columnar (Cassandra)
  + Graph (Neo4j)

**What is DBMS?**

DBMS is software that interacts with databases to create, retrieve, update, and manage data.

**5. MySQL Basics and Installation**

**Installation:**

Installed **MySQL** and accessed it using its CLI (Command Line Interface).

**Common MySQL Commands:**

CREATE DATABASE studentdb;

USE studentdb;

CREATE TABLE students (

id INT PRIMARY KEY,

name VARCHAR(50),

age INT

);

DESC students; -- View structure of the table

DROP DATABASE studentdb;

**Explanation of Commands:**

* CREATE DATABASE: Creates a new database
* USE: Switches context to a selected database
* DESC table\_name: Describes table structure (columns, types, constraints)
* DROP: Deletes databases or tables

Day – 9

Date – 19th June 2025

**1 Procedural Programming**

Procedural Programming is one of the oldest and most widely-used programming paradigms. It is based on the concept of procedure calls, also known as routines or functions. A program is divided into a set of procedures or functions, each performing a specific task. It follows a top-down approach and emphasizes a step-by-step set of instructions for solving a problem. All operations are performed using functions, and data is usually passed from one function to another as arguments.

This paradigm focuses on:

* A linear flow of control
* Reuse of code via procedures
* Easier debugging for small projects

Languages like C, Fortran, and Pascal are classic examples of procedural programming languages.

**2 Functional Programming**

Functional Programming is a paradigm where computation is treated as the evaluation of mathematical functions and avoids changing-state and mutable data. It encourages writing programs by composing pure functions and avoids shared state or side effects. Unlike procedural programming, functional programming does not rely on a sequence of instructions to perform operations.

Key features of functional programming include:

* First-class and higher-order functions
* Immutability (variables, once assigned, cannot be changed)
* Pure functions (output depends only on input, without side effects)
* Recursion in place of loops

Python supports functional programming concepts through constructs like lambda functions, map(), filter(), and reduce().

**3 Object-Oriented Programming (OOP)**

Object-Oriented Programming is a modern and widely-used programming paradigm that organizes a program into objects, which are instances of classes. OOP models real-world entities such as students, employees, and vehicles. It enables the bundling of data and functions that operate on that data into a single unit — the object.

The four core principles of OOP are:

* **Encapsulation:** Wrapping data and the methods that manipulate the data into a single unit. This prevents outside interference and misuse.
* **Inheritance:** One class can inherit the properties and behaviors of another class. This promotes code reuse and reduces redundancy.
* **Polymorphism:** The same function name can be used for different types or classes.
* **Abstraction:** Hiding the internal implementation details and showing only the essential features.

OOP helps in building reusable and scalable applications and is widely used in modern software development.

**4 Fully Object-Oriented vs Pure Object-Oriented Programming Languages**

This part of the training highlighted the distinction between languages that are **fully object-oriented** and those that are **pure object-oriented**.

* **Fully Object-Oriented Languages:** These languages support most or all OOP principles but still allow certain primitive types or constructs outside of the object system. Python is a fully object-oriented language because almost everything is treated as an object, but it still allows the use of primitive types like integers and strings directly.
* **Pure Object-Oriented Languages:** In these languages, everything is considered an object — including primitive types and even control structures. There is no escape from object orientation. An example is Smalltalk.

Understanding this distinction helps in evaluating the level of object-orientation a language supports.

**5 Creating Classes in Python**

In Python, classes are defined using the class keyword. A class is a blueprint for creating objects and can contain attributes (variables) and methods (functions).

**Basic Syntax:**

class Student:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def show\_details(self):

print(f"Name: {self.name}, Age: {self.age}")

In this syntax:

* \_\_init\_\_ is a special method known as a constructor. It is automatically called when a new object is created.
* self is a reference to the current instance of the class. It allows access to the instance variables and methods.

**Creating and Using an Object:**

s1 = Student("Harnoor", 20)

s1.show\_details()

This code defines an object s1 of the class Student and calls its method.

**6 How Python Classes Automatically Pass the Object Address (self)**

When a method is called using an object (e.g., s1.show\_details()), Python automatically passes the memory address of the object as the first parameter to the method, which is conventionally named self. This allows the method to access the attributes and other methods of the object it was called on. Although we don’t explicitly pass self, Python does it behind the scenes.

**7 Database Topics**

We were also introduced to fundamental operations of Structured Query Language (SQL) used in relational database systems.

**7.1 INSERT Command**

The INSERT command is used to add new records into a table.

**Syntax:**

INSERT INTO table\_name (column1, column2) VALUES (value1, value2);

**Example:**

INSERT INTO Students (name, age) VALUES ('Harnoor', 20);

This command adds a new student record to the Students table.

**7.2 SELECT Command**

The SELECT command is used to retrieve records from a table.

**Syntax:**

SELECT column1, column2 FROM table\_name;

To fetch all columns, we use:

SELECT \* FROM Students;

This retrieves all the data stored in the Students table.

**7.3 SQL Constraints**

SQL constraints are used to specify rules for the data in a table. These constraints ensure the integrity, accuracy, and reliability of the data.

* **NOT NULL:** Ensures that a column cannot have a NULL value.
* **UNIQUE:** Ensures all values in a column are different.
* **PRIMARY KEY:** Uniquely identifies each record in a table.
* **FOREIGN KEY:** Maintains referential integrity by linking to the primary key in another table.
* **CHECK:** Ensures that values in a column satisfy a specific condition.
* **DEFAULT:** Provides a default value for a column when none is specified.

**Example with Constraints:**

CREATE TABLE Students (

id INT PRIMARY KEY,

name VARCHAR(50) NOT NULL,

age INT CHECK (age >= 18)

);

**7.4 Relationships in Relational Databases**

Relationships define how tables are related to each other. These are essential for maintaining data normalization and integrity.

Types of Relationships:

* **One-to-One:** Each record in Table A is linked to one record in Table B.
* **One-to-Many:** A record in Table A can relate to multiple records in Table B.
* **Many-to-Many:** Multiple records in Table A can relate to multiple records in Table B, typically implemented via a junction table.

Foreign keys are used to establish these relationships.

Day – 10

Date – 20th June 2025

**1 Understanding Normalization**

Normalization is a technique used to organize data in a relational database to minimize redundancy and improve data integrity. It involves dividing large tables into smaller ones and defining relationships among them. The primary goals are:

* Eliminate redundant data
* Ensure data dependencies make sense
* Maintain database consistency

We reviewed common normal forms (1NF, 2NF, 3NF) and discussed how each step improves structure.

**2 Armstrong’s Axioms**

Armstrong’s Axioms are a set of inference rules used to derive all the **functional dependencies** from a given set. They are critical in database normalization and dependency preservation.

The core rules include:

* **Reflexivity:** If Y is a subset of X, then X → Y
* **Augmentation:** If X → Y, then XZ → YZ
* **Transitivity:** If X → Y and Y → Z, then X → Z

These axioms are foundational to deriving closures and identifying candidate keys.

**3 Closures**

The **closure** of an attribute set (denoted as X⁺) is the complete set of attributes functionally determined by X using a given set of functional dependencies. Closures help in:

* Identifying candidate keys
* Verifying dependency preservation
* Guiding table decomposition during normalization

**4 Candidate Keys**

A **candidate key** is a minimal set of attributes that can uniquely identify a tuple (record) in a relation. A table can have multiple candidate keys, but only one of them becomes the **primary key**. Determining candidate keys involves calculating closures and analyzing functional dependencies.

**SQL Operations Practiced**

**5 Modifying Table Structure – ALTER Command**

The ALTER statement is used to change an existing table structure.

**Examples:**

ALTER TABLE Employees ADD salary INT;

ALTER TABLE Employees MODIFY salary FLOAT;

ALTER TABLE Employees DROP COLUMN salary;

**Updating Data – UPDATE Command**

Used to modify existing records in a table.

**Example:**

UPDATE Employees SET salary = 50000 WHERE emp\_id = 101;

**6. Deleting Data – DELETE Command**

Deletes specific rows based on a condition.

**Example:**

DELETE FROM Employees WHERE emp\_id = 101;

If no WHERE clause is provided, all rows will be deleted.

**7. Filtering Results – WHERE Clause**

Used in combination with other commands to filter rows based on conditions.

**Example:**

SELECT \* FROM Employees WHERE department = 'HR';

**User Management in SQL**

**8. Creating a New User**

**Syntax:**

CREATE USER 'username'@'localhost' IDENTIFIED BY 'password';

**Example:**

CREATE USER 'devuser'@'localhost' IDENTIFIED BY 'dev123';

**9. GRANT Command**

Gives specific privileges to a user.

**Example:**

GRANT SELECT, INSERT ON training\_db.\* TO 'devuser'@'localhost';

**10. REVOKE Command**

Removes previously granted privileges.

**Example:**

REVOKE INSERT ON training\_db.\* FROM 'devuser'@'localhost';

**11. Dropping a User**

**Example:**

DROP USER 'devuser'@'localhost';

This permanently deletes the user from the database.

**12. Table Creation Practice**

We created two related tables as part of hands-on practice:

**Department Table**

CREATE TABLE Department (

dept\_id INT PRIMARY KEY,

dept\_name VARCHAR(50) NOT NULL

);

**Employee Table**

CREATE TABLE Employees (

emp\_id INT PRIMARY KEY,

emp\_name VARCHAR(50),

dept\_id INT,

salary FLOAT,

FOREIGN KEY (dept\_id) REFERENCES Department(dept\_id)

);

This example illustrated the use of **foreign key relationships** to maintain referential integrity between related tables.

Day – 11

Date – 23rd June 2025

**Aggregate Functions**

Aggregate functions perform a **calculation on a set of values** and return a **single value**. These are used in combination with SELECT statements to summarize data.

**1. COUNT()**

Returns the number of records in a column (excluding NULL values).

**Syntax:**

SELECT COUNT(column\_name) FROM table\_name;

**Example:**

SELECT COUNT(emp\_id) FROM Employees;

**2. SUM()**

Returns the total sum of a numeric column.

**Syntax:**

SELECT SUM(column\_name) FROM table\_name;

**Example:**

SELECT SUM(salary) FROM Employees;

**3. AVG()**

Returns the average value of a numeric column.

**Syntax:**

SELECT AVG(column\_name) FROM table\_name;

**Example:**

SELECT AVG(salary) FROM Employees;

**4. MIN()**

Returns the smallest value in a column.

**Syntax:**

SELECT MIN(column\_name) FROM table\_name;

**Example:**

SELECT MIN(salary) FROM Employees;

**5. MAX()**

Returns the largest value in a column.

**Syntax:**

SELECT MAX(column\_name) FROM table\_name;

**Example:**

SELECT MAX(salary) FROM Employees;

**GROUP BY Clause**

The GROUP BY clause is used to **group rows** that have the same values in specified columns and allows aggregate functions to be applied to each group.

**Syntax:**

SELECT column\_name, AGGREGATE\_FUNCTION(column\_name)

FROM table\_name

GROUP BY column\_name;

**Example:**

SELECT department, COUNT(\*) FROM Employees GROUP BY department;

This groups employees by department and counts how many are in each.

**HAVING Clause**

The HAVING clause is used to **filter records after grouping**, similar to WHERE but used specifically with aggregate functions.

**Syntax:**

SELECT column\_name, AGGREGATE\_FUNCTION(column\_name)

FROM table\_name

GROUP BY column\_name

HAVING condition;

**Example:**

SELECT department, COUNT(\*) FROM Employees

GROUP BY department

HAVING COUNT(\*) > 5;

This returns only departments that have more than 5 employees.

**ORDER BY Clause**

The ORDER BY clause is used to **sort the result-set** in ascending or descending order.

**Syntax:**

SELECT column1, column2 FROM table\_name ORDER BY column1 [ASC|DESC];

**Example:**

SELECT emp\_name, salary FROM Employees ORDER BY salary DESC;

This returns the list of employees sorted by salary in descending order.

**Subqueries**

A subquery (also known as an inner query or nested query) is a query **within another SQL query**. It is used to perform operations that require multiple steps of filtering or condition checking.

**Syntax:**

SELECT column\_name

FROM table\_name

WHERE column\_name IN (SELECT column\_name FROM another\_table WHERE condition);

**Example:**

SELECT emp\_name FROM Employees

WHERE salary > (SELECT AVG(salary) FROM Employees);

This returns names of employees whose salary is above the average salary.

Day – 12

Date – 24th June 2025

**LIMIT Keyword**

* **Purpose**: Used to specify the number of records to return.
* **Syntax**:

SELECT \* FROM employees LIMIT 5;

* **Output**: Returns the first 5 rows from the employees table.

**OFFSET Keyword**

* **Purpose**: Used to skip a specific number of rows before starting to return rows.
* **Syntax**:

SELECT \* FROM employees LIMIT 5 OFFSET 3;

* **Output**: Skips the first 3 rows and returns the next 5 rows.

**LIKE Keyword**

* **Purpose**: Used for pattern matching in WHERE clause.
* **Syntax**:

SELECT \* FROM customers WHERE name LIKE 'A%';

* **Output**: Fetches customers whose name starts with the letter 'A'.

**Subqueries / Nested Subqueries**

* **Definition**: A query nested inside another query.

**Types of Subqueries:**

1. **Single Row Subquery**
   * Returns only one row.
   * **Example**:

SELECT name FROM employees WHERE salary = (SELECT MAX(salary) FROM employees);

1. **Multiple Row Subquery**
   * Returns more than one row.
   * **Example**:

SELECT name FROM employees WHERE department\_id IN (SELECT department\_id FROM departments WHERE location\_id = 100);

**ANY Keyword**

* **Purpose**: Compares a value to each value returned by the subquery and returns true if **any one** matches.
* **Example**:

SELECT name FROM employees WHERE salary > ANY (SELECT salary FROM employees WHERE department\_id = 30);

**ALL Keyword**

* **Purpose**: Compares a value to all values returned by the subquery. True only if the condition holds for **all**.
* **Example**:

SELECT name FROM employees WHERE salary > ALL (SELECT salary FROM employees WHERE department\_id = 40);

**Joins in SQL**

**1. INNER JOIN**

* **Returns**: Only matching rows from both tables.
* **Example**:

SELECT employees.name, departments.dept\_name

FROM employees

INNER JOIN departments ON employees.dept\_id = departments.id;

**2. LEFT OUTER JOIN**

* **Returns**: All rows from the left table and matching rows from the right table (NULL where no match).
* **Example**:

SELECT customers.name, orders.order\_id

FROM customers

LEFT JOIN orders ON customers.id = orders.customer\_id;

**3. RIGHT OUTER JOIN**

* **Returns**: All rows from the right table and matching rows from the left table (NULL where no match).
* **Example**:

SELECT orders.order\_id, customers.name

FROM orders

RIGHT JOIN customers ON orders.customer\_id = customers.id;

**4. SELF JOIN**

* **Purpose**: A table joins with itself.
* **Example**:

SELECT A.name AS Employee, B.name AS Manager

FROM employees A

JOIN employees B ON A.manager\_id = B.id;

Day – 13

Date – 25th June 2025

**1. Concept of Database API**

* **DB API** is a standard interface provided by Python for accessing relational databases.
* It enables interaction between Python code and different databases using a common structure.
* Modules that follow Python DB API (PEP 249) include: sqlite3, pymysql, psycopg2, etc.

**2. What is an API?**

* **API (Application Programming Interface)** is a set of protocols and tools that allows two software applications to communicate.
* In databases, APIs allow Python to talk to databases like MySQL, SQLite, PostgreSQL.

**3. Introduction to SQLAlchemy**

* **SQLAlchemy** is a powerful SQL toolkit and Object Relational Mapper (ORM) for Python.
* Allows you to interact with databases using both SQL and Pythonic classes/objects.

**4. Popular DBAPI Libraries**

* **pymysql**: Used for connecting Python with **MySQL**.
* **psycopg2**: Used for connecting Python with **PostgreSQL**.
* Both support parameterized queries and transactions.

**5. Connecting SQLite Database using SQLAlchemy**

from sqlalchemy import create\_engine

# Create an engine for SQLite (local file-based DB)

engine = create\_engine('sqlite:///students.db')

# Connect and execute SQL

with engine.connect() as conn:

result = conn.execute("SELECT sqlite\_version();")

for row in result:

print("SQLite version:", row[0])

**6. Connecting MySQL Database using SQLAlchemy**

from sqlalchemy import create\_engine

# Format: mysql+pymysql://<username>:<password>@<host>/<dbname>

engine = create\_engine('mysql+pymysql://root:password@localhost/mydatabase')

with engine.connect() as conn:

result = conn.execute("SHOW TABLES;")

for row in result:

print(row)

Make sure pymysql is installed using: pip install pymysql

**7. Context Manager using with Keyword**

* Context managers ensure proper **resource management**, especially with file or DB operations.
* Using with ensures that connections are **automatically closed** even if an error occurs.

**8. Example – Using with for File**

with open('data.txt', 'r') as file:

content = file.read()

print(content)

# file is automatically closed after the block

**Example – Using with for DB Connection**

from sqlalchemy import create\_engine

engine = create\_engine('sqlite:///test.db')

with engine.connect() as conn:

result = conn.execute("SELECT \* FROM users")

for row in result:

print(row)

# connection is auto-closed

Day – 14

Date – 27th June 2025

**1. Command Line Based Library Management System**

* Developed a simple yet functional **Library Management System** using **Python OOP concepts**.
* Implemented features:
  + **Add new books**
  + **Display available books**
  + **Borrow books by customers**

**2. Object-Oriented Programming Concepts Used**

* **Classes and Objects:**
  + Defined a Library class and created objects for usage.
* **Class Variables vs Instance Variables:**
  + **Class Variables:** Shared across all instances (e.g., list of books).
  + **Instance Variables:** Unique to each object (e.g., borrowed book by a specific customer).
* **Class Attributes vs Object Attributes:**
  + **Class Attributes:** Define properties common to all instances.
  + **Object Attributes:** Represent data tied to a specific object.

**3. Sample Code Highlight**

class Library:

books = ["Python Basics", "Data Science", "AI Fundamentals"]

def \_\_init\_\_(self):

self.borrowed\_books = []

def display\_books(self):

for book in Library.books:

print(book)

def borrow\_book(self, book\_name):

if book\_name in Library.books:

Library.books.remove(book\_name)

self.borrowed\_books.append(book\_name)

print(f"You have borrowed '{book\_name}'")

else:

print("Book not available")

**4. SOLID Principles (Clean Code Practices)**

We learned the SOLID principles for designing maintainable, extensible software:

* **S - Single Responsibility Principle**
  + Each class should have only one reason to change.
* **O - Open/Closed Principle**
  + Classes should be open for extension, but closed for modification.
* **L - Liskov Substitution Principle**
  + Subclasses should be substitutable for their base classes.
* **I - Interface Segregation Principle**
  + No client should be forced to depend on methods it does not use.
* **D - Dependency Inversion Principle**
  + Depend on abstractions, not concrete implementations.

**What are SOLID Principles?**

**SOLID** is an acronym for **five design principles** that help developers design robust object-oriented systems.  
They were introduced by **Robert C. Martin (Uncle Bob)** and are widely followed in software engineering.

**1. S – Single Responsibility Principle (SRP)**

**Definition:**

A class should **have only one reason to change**, i.e., it should **perform only one responsibility**.

Each class should only handle one job or task.

***Bad Example:***

class Report:

def generate(self):

# logic to generate report

pass

def save\_to\_file(self):

# logic to save to disk

pass

Here, the class is **responsible for both** generating and saving the report – two different concerns.

***Good Example:***

class Report:

def generate(self):

# only report generation logic

pass

class ReportSaver:

def save\_to\_file(self, report):

# only saving logic

pass

Now, **each class has one responsibility**, and future changes to one won’t affect the other.

**2. O – Open/Closed Principle (OCP)**

**Definition:**

**Software entities (classes, functions, modules)** should be **open for extension but closed for modification**.

You should be able to add new functionality **without changing existing code**.

***Bad Example:***

class PaymentProcessor:

def pay(self, method):

if method == "credit":

print("Paid using credit card")

elif method == "paypal":

print("Paid using PayPal")

You have to **modify the code every time** a new payment method is added.

***Good Example (Using Polymorphism):***

class PaymentMethod:

def pay(self):

pass

class CreditCard(PaymentMethod):

def pay(self):

print("Paid using credit card")

class PayPal(PaymentMethod):

def pay(self):

print("Paid using PayPal")

def process\_payment(payment\_method: PaymentMethod):

payment\_method.pay()

You can now add new payment methods **without changing process\_payment()** – just create a new subclass!

**3. L – Liskov Substitution Principle (LSP)**

**Definition:**

**Subclasses should be replaceable with their parent class** without altering the correctness of the program.

If a class S is a subclass of class T, then objects of type T should be replaceable with objects of type S.

***Bad Example:***

class Bird:

def fly(self):

print("Flying")

class Ostrich(Bird):

def fly(self):

raise Exception("I can't fly!")

Replacing Bird with Ostrich will break the application, which **violates LSP**.

***Good Example:***

class Bird:

def make\_sound(self):

print("Chirp!")

class FlyingBird(Bird):

def fly(self):

print("Flying high!")

class Ostrich(Bird):

def walk(self):

print("Walking on ground")

Now, we separated flying behavior so that subclasses follow only applicable behavior.

**4. I – Interface Segregation Principle (ISP)**

**Definition:**

A class should **not be forced to implement methods** it does **not use**.

Keep interfaces **small and specific** rather than fat and general.

***Bad Example:***

class Machine:

def print(self):

pass

def scan(self):

pass

def fax(self):

pass

class OldPrinter(Machine):

def print(self):

print("Printing")

def scan(self):

raise NotImplementedError()

def fax(self):

raise NotImplementedError()

This class is forced to implement methods it doesn’t support.

***Good Example:***

class Printer:

def print(self):

pass

class Scanner:

def scan(self):

pass

class Fax:

def fax(self):

pass

class OldPrinter(Printer):

def print(self):

print("Printing only")

Now classes **only implement the interfaces they need**.

**5. D – Dependency Inversion Principle (DIP)**

**Definition:**

**High-level modules should not depend on low-level modules.** Both should depend on **abstractions** (e.g., interfaces or abstract classes).

Code should depend on **interfaces**, not concrete classes.

***Bad Example:***

class MySQLDatabase:

def connect(self):

print("Connecting to MySQL")

class App:

def \_\_init\_\_(self):

self.db = MySQLDatabase()

Now, the App class is tightly coupled to MySQLDatabase.

***Good Example:***

class Database:

def connect(self):

pass

class MySQLDatabase(Database):

def connect(self):

print("Connected to MySQL")

class App:

def \_\_init\_\_(self, db: Database):

self.db = db

self.db.connect()

Now App **depends on abstraction**, and we can switch databases (e.g., PostgreSQL) without changing the App.

**Summary Table**

| **Principle** | **Full Form** | **Key Idea** |
| --- | --- | --- |
| S | Single Responsibility | One class → One responsibility |
| O | Open/Closed | Extend behavior without modifying code |
| L | Liskov Substitution | Subclasses can replace their parents |
| I | Interface Segregation | Don’t force unnecessary method implementations |
| D | Dependency Inversion | Depend on abstractions, not concrete implementations |

Day – 15

Date – 28th June 2025

**1. Encapsulation**

Encapsulation is the process of wrapping data (variables) and methods into a single unit (class). It restricts direct access to variables.

class Student:

def \_\_init\_\_(self):

self.\_\_name = "Harnoor" # private variable

def get\_name(self):

return self.\_\_name

def set\_name(self, new\_name):

self.\_\_name = new\_name

s = Student()

print(s.get\_name()) # Harnoor

s.set\_name("Kaur")

print(s.get\_name()) # Kaur

**2. Python Compromises Security using dir()**

The dir() function can still access "private" members due to Python's name-mangling, showing that encapsulation is not truly enforced.

print(dir(s))

# Shows: '\_Student\_\_name', which can still be accessed like s.\_Student\_\_name

**3. Inheritance**

Inheritance allows one class to derive the properties of another.

**Types of Inheritance in Python:**

1. **Single Inheritance**
2. **Multiple Inheritance**
3. **Multilevel Inheritance**
4. **Hierarchical Inheritance**
5. **Hybrid Inheritance**

**Example:**

# Single Inheritance

class Parent:

def greet(self):

print("Hello from Parent")

class Child(Parent):

pass

c = Child()

c.greet()

**4. MRO (Method Resolution Order)**

Python uses MRO to determine the method to invoke in multiple inheritance.

class A: pass

class B(A): pass

class C(A): pass

class D(B, C): pass

print(D.mro())

# Output: [D, B, C, A, object]

**5. super() Function**

super() is used to call the constructor or method of a parent class.

class A:

def \_\_init\_\_(self):

print("A init")

class B(A):

def \_\_init\_\_(self):

super().\_\_init\_\_()

print("B init")

b = B()

**6. Polymorphism**

Polymorphism allows the same method to perform different tasks.

class Bird:

def fly(self):

print("Bird can fly")

class Parrot(Bird):

def fly(self):

print("Parrot flies high")

obj = Parrot()

obj.fly()

**7. Dunder/Magic Methods**

These are special methods in Python that start and end with double underscores.

class Employee:

def \_\_init\_\_(self, name):

self.name = name

def \_\_str\_\_(self):

return f"Employee Name: {self.name}"

e = Employee("Harnoor")

print(e) # Uses \_\_str\_\_

**8. Function Overloading**

Python does not support traditional function overloading, but you can simulate it using default parameters or \*args.

def greet(name=None):

if name:

print("Hello", name)

else:

print("Hello")

greet()

greet("Harnoor")

**9. Function Overriding**

Subclasses can override methods of the parent class.

class Parent:

def show(self):

print("Parent")

class Child(Parent):

def show(self):

print("Child")

c = Child()

c.show()

**10. Abstract Class**

An abstract class is a class that contains one or more abstract methods and cannot be instantiated.

from abc import ABC, abstractmethod

class Shape(ABC):

@abstractmethod

def area(self):

pass

**11. Abstract Method**

A method declared in the abstract class without implementation. It must be implemented in a subclass.

class Circle(Shape):

def area(self):

return 3.14 \* 5 \* 5

**12. Concrete Class**

A class that implements all abstract methods of its abstract parent and can be instantiated.

c = Circle()

print(c.area()) # Output: 78.5

Day – 16

Date – 30th June 2025

**1. MRO (Method Resolution Order)**

MRO defines the order in which base classes are searched when executing a method.

Useful especially in **multiple inheritance** to avoid ambiguity.

class A:

def show(self): print("A")

class B(A):

def show(self): print("B")

class C(A):

def show(self): print("C")

class D(B, C):

pass

d = D()

d.show() # Output: B (because B appears before C in D's inheritance)

print(D.mro())

# Output: [<class '\_\_main\_\_.D'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class 'object'>]

**2. Difference Between \_\_str\_\_() and \_\_repr\_\_()**

|  |  |  |
| --- | --- | --- |
| **Feature** | **\_\_str\_\_()** | **\_\_repr\_\_()** |
| Purpose | For end users | For developers/debugging |
| Usage | Used by print() | Used in interpreter or repr() |
| Goal | Readable, human-friendly output | Precise, unambiguous (can recreate object) |
| Fallback | repr() used if str() is not defined | str() not used if repr() is defined |

class Book:

def \_\_init\_\_(self, title):

self.title = title

def \_\_str\_\_(self):

return f"Book: {self.title}"

def \_\_repr\_\_(self):

return f"Book('{self.title}')"

b = Book("Python 101")

print(str(b)) # Book: Python 101

print(repr(b)) # Book('Python 101')

**3. ORM (Object Relational Mapping)**

ORM is a technique that allows you to interact with a database using Python classes instead of SQL queries.

**Popular Python ORM:** SQLAlchemy, Django ORM

# Using SQLAlchemy ORM

from sqlalchemy import Column, Integer, String, create\_engine

from sqlalchemy.orm import declarative\_base, sessionmaker

Base = declarative\_base()

class Student(Base):

\_\_tablename\_\_ = 'students'

id = Column(Integer, primary\_key=True)

name = Column(String)

engine = create\_engine('sqlite:///school.db')

Base.metadata.create\_all(engine)

**4. @property Decorator**

It allows you to define a method that acts like an attribute. It improves encapsulation and adds getter/setter behavior.

class Student:

def \_\_init\_\_(self, marks):

self.\_marks = marks

@property

def marks(self):

return self.\_marks

@marks.setter

def marks(self, value):

if value < 0:

raise ValueError("Marks cannot be negative")

self.\_marks = value

s = Student(90)

print(s.marks)

s.marks = 95

**5. How to Write Clean Code Using @property and Classes**

✅ Benefits of using @property:

* Hides internal implementation
* Provides getter/setter without changing interface
* Adds validation logic cleanly

**Good Code Example:**

class Account:

def \_\_init\_\_(self, balance):

self.\_balance = balance

@property

def balance(self):

return self.\_balance

@balance.setter

def balance(self, amount):

if amount < 0:

raise ValueError("Balance can't be negative")

self.\_balance = amount

acc = Account(1000)

acc.balance = 1200 # sets balance

print(acc.balance) # gets balance

**6. Mini Project: Library Management System Using OOPs**

**Features Covered in Project:**

* Class-based design
* Book inventory
* Issue/Return functionality
* Use of encapsulation and @property

class Book:

def \_\_init\_\_(self, title, author):

self.title = title

self.author = author

self.\_available = True

@property

def available(self):

return self.\_available

def issue(self):

if self.\_available:

self.\_available = False

return True

return False

def return\_book(self):

self.\_available = True

class Library:

def \_\_init\_\_(self):

self.books = []

def add\_book(self, book):

self.books.append(book)

def display\_books(self):

for book in self.books:

status = "Available" if book.available else "Issued"

print(f"{book.title} by {book.author} - {status}")

lib = Library()

book1 = Book("Python Basics", "Guido")

book2 = Book("AI 101", "Andrew Ng")

lib.add\_book(book1)

lib.add\_book(book2)

book1.issue()

lib.display\_books()

Day – 17

Date – 30th June 2025

**1. Virtual Environment in Python**

A **virtual environment** is an isolated environment used to manage dependencies for Python projects without affecting global packages.

**Why it's important:**

* Avoids package conflicts between projects
* Keeps your project lightweight and clean
* Makes deployment and collaboration easier

**Creating and activating a virtual environment:**

# Create

python -m venv myenv

# Activate (Windows)

myenv\Scripts\activate

# Activate (Linux/Mac)

source myenv/bin/activate

**2. MySQL Database Integration using SQLAlchemy**

We learned how to connect **MySQL** databases in Python using **SQLAlchemy ORM**, which maps Python classes to database tables.

from sqlalchemy import create\_engine

engine = create\_engine("mysql+pymysql://username:password@localhost/dbname")

**3. Creating Tables Using ORM**

Using declarative\_base() and class definitions, we created tables directly from Python code:

from sqlalchemy import Column, Integer, String

from sqlalchemy.orm import declarative\_base

Base = declarative\_base()

class Student(Base):

\_\_tablename\_\_ = 'students'

id = Column(Integer, primary\_key=True)

name = Column(String(100))

age = Column(Integer)

Base.metadata.create\_all(engine) # Creates the table

**4. Inserting Records into the Table**

We learned how to create a session and add records using ORM:

from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind=engine)

session = Session()

new\_student = Student(name="Harnoor", age=19)

session.add(new\_student)

session.commit()

**5. Deleting Records Using ORM**

student\_to\_delete = session.query(Student).filter\_by(name="Harnoor").first()

session.delete(student\_to\_delete)

session.commit()