❖ Module _8) Advance Python Programming

➢ Printing on Screen:

1.Introduction to the print() function in Python.

Ans:

- The print() function is one of the most commonly used functions in Python.
- It is used to **display output on the screen**.
- print() automatically adds a newline at the end unless you change end.
- You can print numbers, strings, variables, or even expressions.
- Multiple items can be printed in one statement by separating them with commas.

Examples:

1. **Simple Text Output**

```
print("Hello, Python!")
```

Output:

Hello, Python!

2. **Printing Multiple Values**

```
print("My name is", "Hensi")
```

Output:

My name is Hensi

Created by:Hensi Vaghela

2.Formatting outputs using f-strings and format().

Ans:

- When displaying output, you often want to make it **neat and structured**.
- Python provides two popular ways to format strings:
    1. **Using f-strings (formatted string literals)**
    2. **Using the .format() method**

1.Using f-Strings:

- f-Strings allow you to **embed variables or expressions directly inside a string** by prefixing the string with f or F and using {} for placeholders.

Example:

```
name = "Hensi"
age = 21
print(f"My name is {name} and I am {age} years old.")
```

Output:

```
My name is Hensi and I am 21 years old.
```

2.Using .format() Method:

Before f-strings, .format() was widely used for string formatting.
You place {} in the string and pass the values to .format().

Example:

```
name = "Hensi"
age = 21
print("My name is {} and I am {} years old.".format(name, age))
```

Output:

```
My name is Hensi and I am 21 years old.
```

➢ Reading Data from Keyboard:

1.Using the input() function to read user input from the keyboard.

Ans:

- Input means taking data from user.
- In python,input is taken using the input() function.
- input() **pauses the program** until the user presses Enter.
- It **always returns a string** → use int() or float() for numeric input.
- You can use .split() to accept multiple values in a single line.

Basic Syntax:

```
variable = input("Your message: ")
```

Example:

```
age = int(input("Enter your age: "))
print("Next year you will be", age + 1)
```

Output:

```
Enter your age: 21
Next year you will be 22
```


Created by:Hensi Vaghela

2.Converting user input into different data types (e.g., int, float, etc.).

Ans:

- By default, the input() function **returns data as a string**, even if the user types numbers.
- To perform calculations or specific operations, you need to **convert** this string into the required data type.

Common Conversions:

Conversion	Function	Example Input	Resulting Type
Integer	int()	"10"	10 (int)
Float	float()	"10.5"	10.5 (float)
String	str()	10	"10" (str)
Boolean	bool()	"True" / non-empty	True / False

- **Why conversion is needed:**

  Input is always string → without conversion, math operations will cause errors.

- **Type Casting:**

  The process of converting one data type to another using functions like int(), float(), or str() is called **type casting**.

- **Invalid input:**

  If the user types something that can't be converted (e.g., int("abc")), Python will raise a ValueError.

➢ Opening and Closing Files:

1.Opening files in different modes ('r', 'w', 'a', 'r+', 'w+').

Ans:

- To work with files, you use the built-in **open()** function:

  file = open("filename.txt", "mode")
- **filename.txt**:- The name of the file.
- **Mode**:- Tells Python what action to perform (read, write, append, etc.).

Common File Modes:

Mode	Meaning	File Required	Description
'r'	Read (default)	Yes	Opens file for reading. Error if file doesn't exist.
'w'	Write	No	Creates new file or overwrites existing content.
'a'	Append	No	Opens file for appending data at the end without deleting old content.
'r+'	Read and Write	Yes	Reads and updates existing file. Error if file doesn't exist.
'w+'	Write and Read	No	Creates a new file (or overwrites) and allows reading/writing.

2. Using the open() function to create and access files.

Ans:

- The **open()** function is used to **create, read, or modify files**.
- It returns a **file object**, which allows you to perform operations like reading, writing, or appending data.

Common modes:

- 'r' → Read
- 'w' → Write (create or overwrite)
- 'a' → Append
- 'r+' → Read & Write
- 'w+' → Write & Read

1.Creating a File (Write Mode)

Using 'w' mode will **create the file** if it doesn't exist, or **overwrite** it if it does.

```
file = open("example.txt", "w")   # Create or overwrite file
file.write("This is a new file.\n")
file.write("File created using open() in write mode.")
file.close()
print("File created successfully!")
```

2.Accessing a File (Read Mode)

Use 'r' mode to open an existing file and read its content.

```
file = open("example.txt", "r")
content = file.read()      # Read entire file
print("File Content:\n", content)
file.close()
```

3.Closing files using close().

Ans:

- When you work with files using the open() function, it's important to **close them** after finishing.
- This ensures that all data is **properly saved** and **system resources are released**.

Basic Syntax:

```
file = open("filename.txt", "mode")
# perform file operations
file.close()
```

Example:

```
f = open("example.txt", "w")
f.write("This is some data.")
f.close()   # ☑File is now closed properly
```

After calling close(), the file is no longer available for reading or writing unless you open it again.

Why Closing is Important:

- Ensures **data is written** completely to the file (especially in write mode).
- Frees up **system resources**.
- Prevents data corruption and file access errors.

Checking if a File is Closed:

You can check the file's status using the .closed attribute:

```
f = open("example.txt", "r")
print(f.closed)   # False
f.close()
print(f.closed)   # True
```

➢ Reading and Writing Files:

1.Reading from a file using read(), readline(), readlines().

Ans:

- When you open a file in **read mode ('r')**, Python gives you different methods to **read its contents** depending on your needs.

1.read() – Read the Entire File:

- Reads **the whole file content at once** as a single string.
- Useful for small files.

2.readline() – Read One Line at a Time:

- Reads the **next line** from the file each time it's called.
- Useful for processing large files line by line.

3.readlines() – Read All Lines into a List:

- Reads **all lines at once** and returns a **list**, where each element is one line from the file.
- Useful when you need to work with lines individually.

2.Writing to a file using write() and writelines().

Ans:

Python provides two main methods to **write data into files**:

1.write() → Writes a **single string** to the file.

2.writelines() → Writes a **list of strings** to the file.

You must open the file in a **write ('w')** or **append ('a')** mode before writing.

1.Using write():

- write() writes **one string** at a time.
- It does **not** add a newline automatically — you must add \n manually if needed.

```
f = open("example.txt", "w")   # 'w' will create or overwrite the file
f.write("Hello, Python!\n")
f.write("This is a second line.")
f.close()
print("Data written successfully using write()!")
```

2.Using writelines():

- writelines() writes **a list of strings** to the file.
- It does not insert line breaks automatically — include \n in each string if needed.

```
lines = [
    "Line 1: First line\n",
    "Line 2: Second line\n",
    "Line 3: Third line\n"
]

f = open("example.txt", "w")
f.writelines(lines)
f.close()
print("Multiple lines written successfully using writelines()!")
```

➢ Exception Handling:

1.Introduction to exceptions and how to handle them using try, except, and finally.

Ans:

- An **exception** is an error that occurs during the execution of a program.
- If not handled, the program will **stop abruptly**.

Example of an exception:

```
x = 10
y = 0
print(x / y)   #ZeroDivisionError
```

❖ Handling Exceptions with try and except:

Python gives us a way to handle errors using:

- Try:

  Code that might cause an error.

- Except:

  Code that runs if an error occurs.

Example:

```
try:
    x = 10
    y = 0
    print(x / y)       #This will raise an exception
except ZeroDivisionError:
    print("You cannot divide by zero!")
```

Output:

You cannot divide by zero!

Note:The program **doesn't crash**, and the error is handled gracefully.

❖ **Using finally Block:**

- finally is **always executed**, whether an error occurs or not.
- Commonly used for cleanup tasks (like closing files, releasing resources, etc.)

Example:

```
try:
    file = open("test.txt", "r")
    data = file.read()
    print(data)
except FileNotFoundError:
    print("File not found!")
finally:
    print("Execution completed. Closing file (if open).")
```

Output (if file doesn't exist):

```
File not found!
Execution completed. Closing file (if open).
```

❖ **Summary Table**

Keyword	Purpose
try	Write the risky code here
except	Handle specific or general exceptions
finally	Always executes, used for cleanup

2.Understanding multiple exceptions and custom exceptions.

Ans:

❖ **Multiple Exceptions and Custom Exceptions:**

- In Python, a program can face different types of errors.
- We use **multiple except blocks** to handle each error separately.
- For example, we can catch ValueError for wrong input and ZeroDivisionError for dividing by zero.
- We can also catch **multiple exceptions in one block** by using a tuple or use a **generic exception** to catch any error.
- **Custom exceptions** are user-defined errors.
- We create them by making a new class that inherits from Exception.
- Then we use raise to trigger the error.
- Custom exceptions make programs easier to understand and handle specific problems clearly.

Example:

```
class NegativeNumberError(Exception):
    pass

try:
    num = int(input("Enter a number: "))
    if num < 0:
        raise NegativeNumberError("Number cannot be negative!")
    print(10 / num)
except ValueError:
    print("Invalid number!")
except ZeroDivisionError:
    print("Cannot divide by zero!")
except NegativeNumberError as e:
    print(e)
```

➢ Class and Object (OOP Concepts):

1. Understanding the concepts of classes, objects, attributes, and methods in Python.

Ans:

- In Python, a **class** is a blueprint for creating objects.
- An **object** is an instance of a class that holds real data.
- **Attributes** are variables inside a class that store data.
- **methods** are functions inside a class that define its behavior.

Example:

```
class Student:
    def  getdata(self, name, age):
        self.name = name      # Attribute
        self.age = age

    def show(self):          # Method
        print(f"Name: {self.name}, Age: {self.age}")

# Object creation
s1 = Student("Hensi", 21)
s1.show()
```

2. Difference between local and global variables.

Ans:

Local Variable	Global Variable
Declared inside a function	Declared outside all functions
Accessible only within that function	Accessible throughout the program
Created when the function starts	Created when the program starts
Destroyed when the function ends	Destroyed when the program ends
Used for temporary calculations	Used for shared data across functions

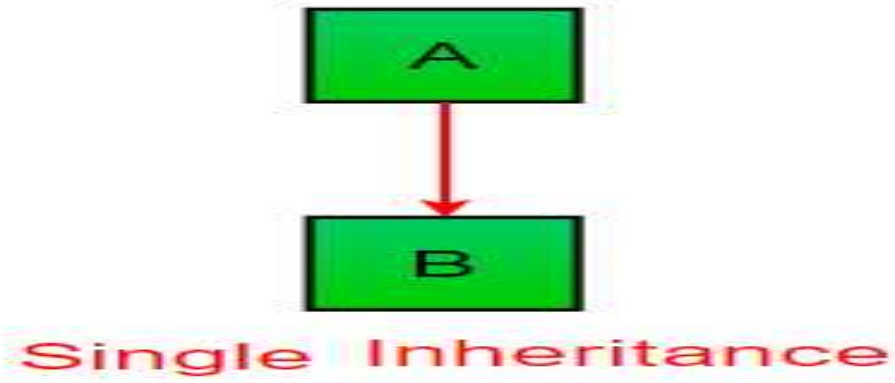
➢ Inheritance:

1.Single, Multilevel, Multiple, Hierarchical, and Hybrid inheritance in Python.

Ans:

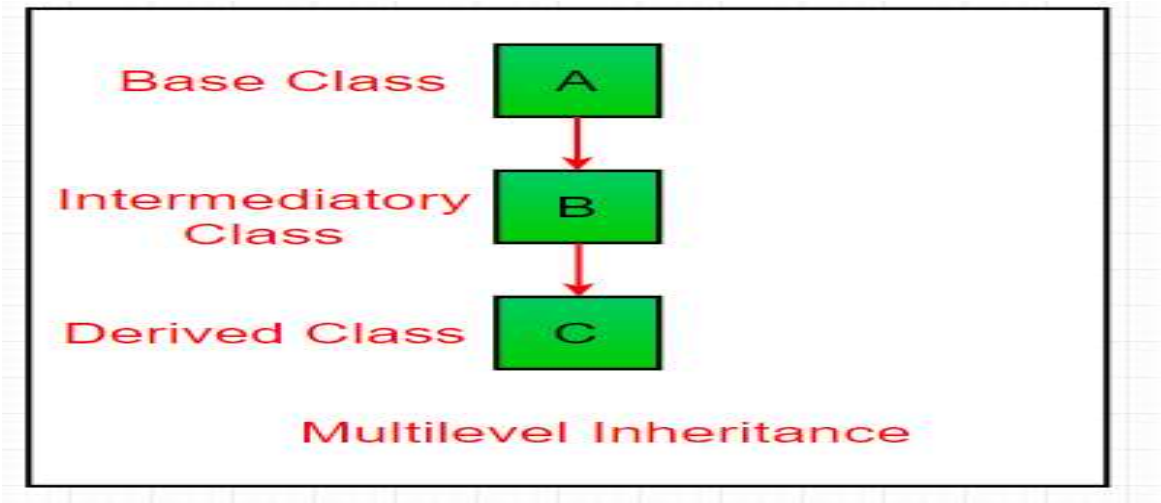
1. Single Inheritance:

- In single inheritance, a child class inherits from only one parent class. It helps in **code reusability** and **simplifies the class hierarchy**.



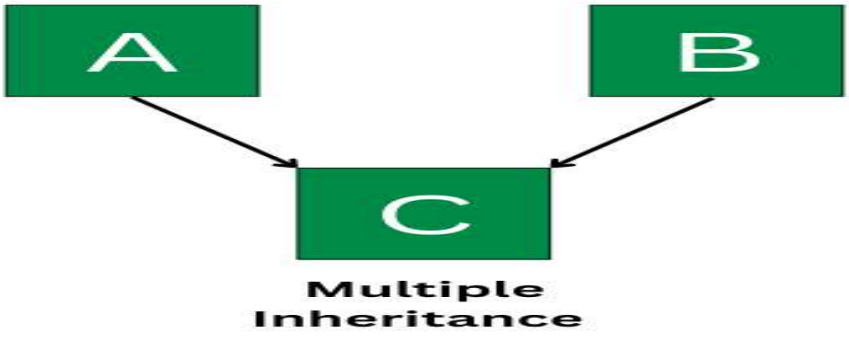
2. Multilevel Inheritance:

- In multilevel inheritance, a class is derived from another derived class — like a **chain**.



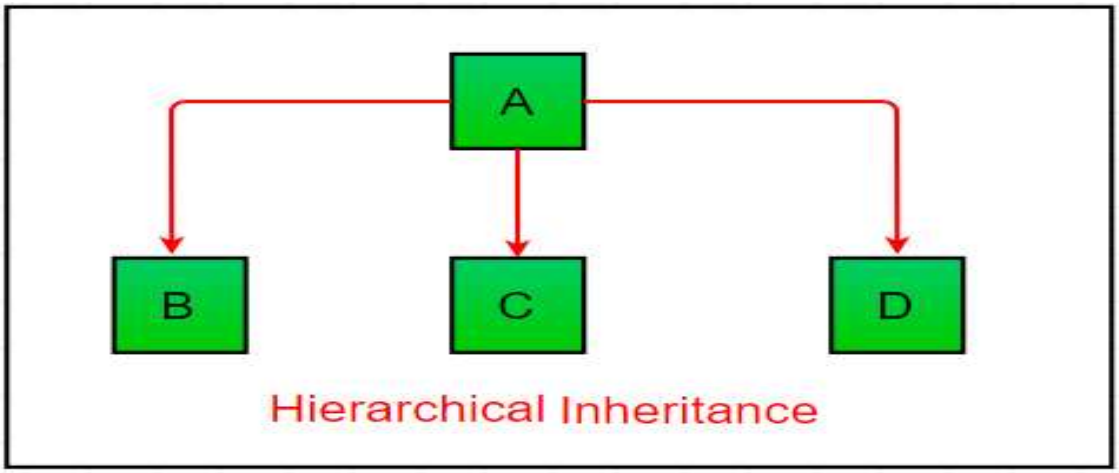
3. Multiple Inheritance:

- In multiple inheritance, a child class inherits from **more than one parent**.It helps to **combine features** of multiple classes.



4. Hierarchical Inheritance:

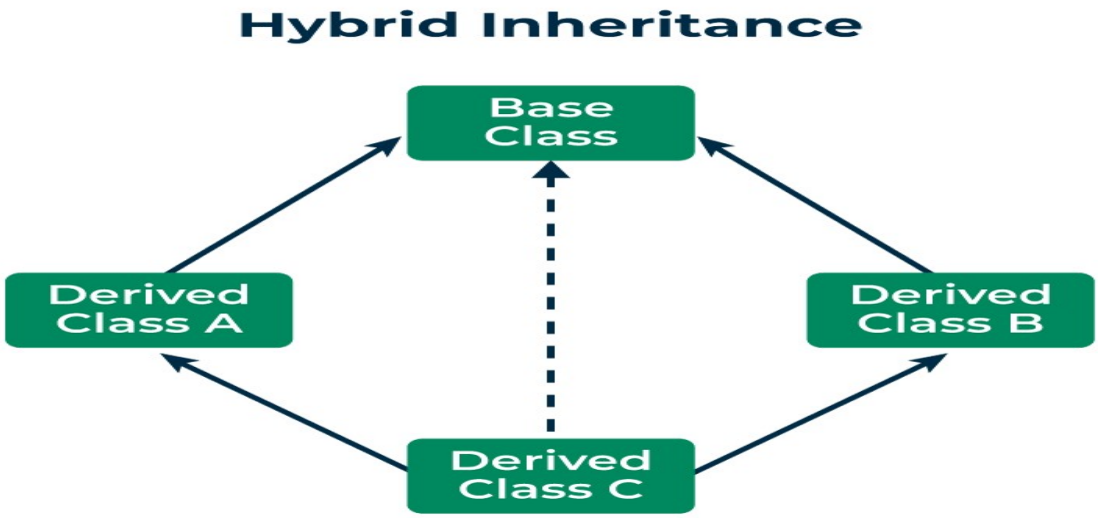
- In hierarchical inheritance, **multiple child classes inherit from the same parent class**.



5. Hybrid Inheritance:

- Hybrid inheritance is a **combination of two or more types** of inheritance.

  Example: Combining multiple + multilevel.



2.Using the super() function to access properties of the parent class.

Ans:

- The super() function is used inside a child class to **access methods and properties of its parent class**.
- It helps in **code reusability**, avoids repeating the parent class name, and is useful in **multiple inheritance** too.

Example:

```
class Parent:
    def display(self):
        print("This is Parent class")
class Child(Parent):
    def display(self):
        super().display()   # Access parent method
        print("This is Child class")
# Create object
obj = Child()
obj.display()
```

Why use super():

- Easier to maintain and update parent class.
- Avoids hardcoding parent class names.
- Works well with multiple inheritance (respects method resolution order).

➢ Method Overloading and Overriding:

1.Method overloading: defining multiple methods with the same name but different parameters.

Ans:

- **Method Overloading** means defining **multiple methods with the same name but different parameters** (number or type of arguments).
- In Python, true method overloading is **not supported**, but we can achieve similar behavior using **default parameters** or ***args**.

Example:

```
class Calculator:

    def add(self, a=0, b=0, c=0):
        return a + b + c

obj = Calculator()

print(obj.add())          # Output: 0
print(obj.add(5))         # Output: 5
print(obj.add(5, 10))     # Output: 15
print(obj.add(5, 10, 15)) # Output: 30
```

Explanation:

Here, the add() method behaves differently depending on how many arguments are passed.

This shows **method overloading** in a Python way.

2.Method overriding: redefining a parent class method in the child class.

Ans:

- **Method Overriding** means **redefining a parent class method in the child class** with the **same name** and **same parameters**, to change or extend its behavior.

Example:

```
class Animal:

    def sound(self):
        print("Animals make sounds")

class Dog(Animal):

    def sound(self):
        print("Dog barks")

obj = Dog()

obj.sound()   # Output: Dog barks
```

Explanation:

Here, the sound() method of the **Dog** class **overrides** the sound() method of the **Animal** class.

This is called **method overriding** — when a child class gives its own version of a parent class method.

➢ SQLite3 and PyMySQL (Database Connectors):

1.Introduction to SQLite3 and PyMySQL for database connectivity.

Ans:

- **SQLite3** is a built-in database module in Python.
- It helps you create and manage a **lightweight, file-based database** without needing a separate database server.

Features:

- Comes **pre-installed** with Python.
- Stores data in a single .db or .sqlite file.
- Best for **small applications** or **testing**.

Example:

```
import sqlite3

# Connect to database (or create if not exists)
conn = sqlite3.connect("student.db")

# Create cursor
cursor = conn.cursor()

# Create table
cursor.execute("CREATE TABLE IF NOT EXISTS student (id INTEGER, name
TEXT)")

# Insert data
cursor.execute("INSERT INTO student VALUES (1, 'Hensi')")

# Save changes
conn.commit()

# Fetch data
cursor.execute("SELECT * FROM student")
print(cursor.fetchall())

# Close connection
conn.close()Introduction to PyMySQL
```

- **PyMySQL** is a third-party Python library used to connect Python with a **MySQL database server**.

Features:

- Used for **web applications** and **large databases**.
- Requires **MySQL server** installed on your system.
- You need to install the library:
- pip install pymysql

Example:

```
import pymysql

# Connect to MySQL server
conn = pymysql.connect(host="localhost", user="root", password="",
database="school")

# Create cursor
cursor = conn.cursor()

# Create table
cursor.execute("CREATE TABLE IF NOT EXISTS student (id INT, name
VARCHAR(50))")

# Insert data
cursor.execute("INSERT INTO student VALUES (1, 'Hensi')")

# Save changes
conn.commit()

# Fetch data
cursor.execute("SELECT * FROM student")
print(cursor.fetchall())

# Close connection
conn.close()
```

2.Creating and executing SQL queries from Python using these connectors.

Ans:

❖ **Using SQLite3:**

SQLite3 is built into Python and used for local (file-based) databases.

Example:

```
import sqlite3

# Connect to database (or create one)
conn = sqlite3.connect("student.db")
cursor = conn.cursor()

# Create table
cursor.execute("CREATE TABLE IF NOT EXISTS student (id INTEGER, name
TEXT)")

# Insert data
cursor.execute("INSERT INTO student VALUES (1, 'Hensi')")

# Select data
cursor.execute("SELECT * FROM student")
data = cursor.fetchall()
print(data)

# Update data
cursor.execute("UPDATE student SET name='Riya' WHERE id=1")

# Delete data
cursor.execute("DELETE FROM student WHERE id=1")

# Save changes
conn.commit()
conn.close()
```

❖ Using PyMySQL:

PyMySQL connects Python with the MySQL database server.

Example:

```
import pymysql

# Connect to MySQL
conn = pymysql.connect(host="localhost", user="root", password="",
database="school")
cursor = conn.cursor()

# Create table
cursor.execute("CREATE TABLE IF NOT EXISTS student (id INT, name
VARCHAR(50))")

# Insert data
cursor.execute("INSERT INTO student VALUES (1, 'Hensi')")

# Select data
cursor.execute("SELECT * FROM student")
data = cursor.fetchall()
print(data)

# Update data
cursor.execute("UPDATE student SET name='Riya' WHERE id=1")

# Delete data
cursor.execute("DELETE FROM student WHERE id=1")

# Save changes
conn.commit()
conn.close()
```

➢ Search and Match Functions:

1.Using re.search() and re.match() functions in Python's re module for pattern matching.

Ans:

In Python, the **re module** is used for pattern matching with regular expressions.

- **re.match()** checks for a match **only at the beginning** of a string.
- **re.search()** checks for a match **anywhere** in the string.

Example:

```
import re

text = "Hello Python"

# re.match() – checks from start
m1 = re.match("Hello", text)
print(m1)  # Match found

# re.search() – checks anywhere
m2 = re.search("Python", text)
print(m2)  # Match found
```

2.Difference between search and match.

Ans:

Point	re.match()	re.search()
1	Checks pattern **only at the beginning** of the string	Checks pattern **anywhere** in the string
2	Returns match only if found at **start**	Returns match if found **anywhere**
3	Returns None if pattern not at the beginning	Returns None only if pattern not found at all
4	Used for **strict start matching**	Used for **general pattern searching**
5	Faster for start-only checks	Scans the entire string
6	Example: re.match("Hi", "Hi Python") ☑	Example: re.search("Python", "Hi Python") ☑