Module 8 Advance Python Programming

1.Printing on Screen

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

- The print() function in Python is used to show output on the screen. It displays text, numbers, or the value of variables.

- The **print() function** is one of the most commonly used built-in functions in Python.

Examples of Using print()

1. Printing a simple message:

print("Hello, Python!")

Output:

Hello, Python!

2. Printing multiple items:

print("Name:", "Alice", "Age:", 20)

Output:

Name: Alice Age: 20

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

Using the f-strings

- f-strings are the simplest and fastest way to format output.

- They allow you to write variables directly inside a string by using {}.

Syntax:

print(f"Text {variable}")

- Prefix the string with f or F.
- Place variables or expressions inside {}.

Using the format() Method

- The format() method is an older but still powerful way to format output.
- It uses **placeholders {}** inside a string and fills them with values in order or by name.

Syntax:

print("Text {}".format(value))

Example 1: Basic usage name = "Dhiraj" age = 17 print("My

name is {} and I am {} years old.".format(name, age))

Output:

My name is Dhiraj and I am 17 years old.

Example 2: Using index numbers

print("{1} is older than {0}".format("Ravi", "Aman"))

Output:

Aman is older than Ravi

--{1} → 2nd value, {0} → 1st value.

2. Reading Data from Keyboard

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

- The input() function is a built-in function in Python used to take input from the keyboard.
- It allows the program to interact with the user by asking for information while the program is running.

Syntax:

variable = input("message")

- message – (optional) A text prompt shown to the user.
- variable – The input value entered by the user is stored in this variable as a string.

Example 1: Basic input name =

input("Enter your name: ")

print("Hello,", name) Output:

Enter your name: Dhiraj

Hello, Dhiraj

Important: input() always returns a string

Even if the user types a number, it is stored as a string.
If you want it as a number (int or float), you must convert it

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

- By default, the input() function always returns user input as a string (str), even if the user enters a number.
- To use the input as a number (for calculations), we must convert it into another data type like int or float.

Syntax:

variable = datatype(input("Message"))

- datatype – The data type you want to convert to (int, float, etc.)
- input("Message") – Reads the user input from the keyboard.

1. **Converting Input to Integer (int)**

- Use int() when you want to convert input into a whole number (integer).

  age = int(input("Enter your age: ")) print("Next

  year you will be", age + 1)

  Output:

  Enter your age: 17

  Next year you will be 18

2. **Converting Input to Float (float)**

- Use float() when you want decimal numbers.

  height = float(input("Enter your height in meters: ")) print("Your height is", height, "meters")

  Output:

  Enter your height in meters: 1.72

  Your height is 1.72 meters

3. Opening and Closing Files Theory:

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

- In Python, files are opened using the built-in open() function.
- It allows you to read from, write to, or append to files by specifying a mode.

Syntax:

file_object = open("filename", "mode")

- filename – name of the file (e.g., "data.txt")
- mode – how you want to open the file (read, write, append, etc.)

Mode	Description	Creates File if Not Exists	Deletes Old Data
'r'	Read only	No	No
'w'	Write only	Yes	Yes
'a'	Append only	Yes	No
'r+'	Read + Write	Yes (must exist)	No
'w+'	Write + Read	Yes	Yes

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

- **Open() function** is used to **create, open, read, and write files**.
- It is one of the most important functions for file handling and allows your program to **store data permanently** on the disk.

Syntax:

file_object = open("filename", "mode")

- **filename** – name of the file you want to open or create (e.g., "data.txt")
- **mode** – tells Python what you want to do with the file: read ('r'), write ('w'), append ('a'), etc.

Steps to Work with Files:

1. Open the file using open()
2. Perform read or write operations
3. Close the file using close()

1. Creating a File (using open() with 'w' or 'a')

- If the file **does not exist**, Python will **create it automatically** when you open it in **write** ('w') or **append** ('a') mode.

Create a new file or overwrite if it exists file

= open("myfile.txt", "w") file.write("Hello,

this is my first file!") file.close()

print("File created and data written successfully!")

Explanation:

- "w" – opens the file for writing (creates new or overwrites existing).
- write() – writes data into the file.

- close() – closes the file (important to save changes).

2. Appending Data to a File

- To **add new content without erasing existing data**, use **append mode 'a'**.

file = open("myfile.txt", "a") file.write("\nThis

line is added later.") file.close()

print("New data added successfully!")

Explanation:

- "a" – opens the file for appending data at the end.
- Does **not erase existing data**.

3.Closing files using close().

- After working with a file in Python, we use the **close()** method to **close it properly**.
- This makes sure all data is saved and frees up memory.

Syntax : file_object.close()

Example:

```
f = open("data.txt", "w")
f.write("Hello, Python!")
f.close()
```

Important Points:

- Always close a file after reading or writing.
- If you don't close it, data might not save correctly.
- with open() closes the file automatically.

4. Reading and Writing Files

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

Python provides three main methods to read data from a file:

1. read() – Reads the entire content of the file.
2. readline() – Reads one line at a time.
3. readlines() – Reads all lines and returns them as a list.

1. Using read()

- Reads the whole file as a single string.

Example:

```
f = open("data.txt",
"r") content = f.read()
print(content)
f.close()
```

Output:

```
Hello, Python!
Welcome to file handling.
```

2. Using readline()

- Reads one line at a time.
- Each call to readline() moves the cursor to the next line.

Example:

```
f = open("data.txt",
"r") line1 = f.readline() line2
= f.readline() print(line1)
print(line2)
f.close()
```

Output:

Hello, Python!

Welcome to file handling.

3. Using readlines()

- Reads all lines and returns a list of lines.

f = open("data.txt", "r") lines

= f.readlines() print(lines)

f.close()

Output:

['Hello, Python!\n', 'Welcome to file handling.\n']

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

1. Writing using write()

The write() method is used to **write a string to a file**.

Syntax:

file_object.write(string)

- If the file does not exist, Python will create it.
- Using mode 'w' overwrites the existing file. •

  Using mode 'a' appends to the existing file.

Example 1: Writing a single line

Open file in write mode file =

open("example.txt", "w")

Write a single line file.write("Hello,

this is the first line.\n")

Close the file file.close()

Example 2: Writing multiple lines using write() file

= open("example.txt", "w")

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

for line in lines:

file.write(line)

file.close()

2. Writing using writelines()

The writelines() method **writes a list of strings to a file**. It does **not add newlines automatically**, so you must include \n if you want lines separated.

Syntax:

file_object.writelines(list_of_strings)

Example 1: Using writelines() file

= open("example.txt", "w")

lines = ["First line\n", "Second line\n", "Third line\n"]

file.writelines(lines)

file.close()

5. Exception Handling

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

What is an Exception?

- An **exception** is an error that occurs during program execution, which **disrupts the normal flow** of the program.

  For example:

```
a = 10 b = 0 print(a / b) # ZeroDivisionError
```

  Here, dividing by zero causes an exception — ZeroDivisionError.

Why Handle Exceptions?

- If exceptions are not handled, the program **stops abruptly**.
- Exception handling lets you **manage errors gracefully** and keep the program running.

Syntax of try–except–finally

```
try:
  # Code that may cause an error except:
  # Code to handle the error finally:
  # Code that will run no matter what
```

Example : Basic try–except

```
try:
 a = int(input("Enter a number: "))  b = int(input("Enter another number: "))
  print("Result:", a / b) except:
  print("Something went wrong!")
```

2. Understanding multiple exceptions and custom exceptions.

What are Multiple Exceptions?

- Sometimes, a program can cause **different types of errors**.
- Instead of writing one except for all, you can **handle each error separately**.

Example 1: Handling Multiple Exceptions Separately

```
try:
   a = int(input("Enter number 1: "))
b = int(input("Enter number 2: "))
   result = a / b
print("Result:", result)
except ZeroDivisionError:
   print("Error: You cannot divide by zero!") except
ValueError:
   print("Error: Please enter valid numbers only!") except
Exception as e:
   print("Some other error occurred:", e)
```

Example 2: Handling Multiple Exceptions Together

- If you want to handle **different exceptions in the same way**,you can group them inside **a tuple**.

```
try:
   num = int(input("Enter a number: "))
print(10 / num) except
(ZeroDivisionError, ValueError):
   print("Error: Invalid input or division by zero!")
```

What are Custom Exceptions?

- Sometimes, you may want to create **your own type of error**. This is called a **custom exception**.

Example 3: Creating a Custom Exception

```
# Define custom exception class
AgeError(Exception):
    pass


# Use custom exception
try:
    age = int(input("Enter your age: "))
    if age < 18:
        raise AgeError("You must be 18 or older!")
    else:
        print("Welcome! You are eligible.") except
AgeError as e:
    print("Custom Exception:", e) except
ValueError:
    print("Please enter a valid number.")
```

6. Class and Object (OOP Concepts)

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

1.Class

- A class is a blueprint or template.
- A class is collection of data member and member function.

Example analogy: A car design is a class.

2.Object

- An object is a real thing made from a class.
- It has its own data but uses the class's template.

Example analogy: A real car you drive is an object.

3.Attributes

- Attributes are the properties or details of an object.
- Example: color, name, age, model.

Example analogy:

- A car's color or speed is an attribute.

4.Methods

- Methods are actions or functions of an object.
- They define what an object can do.

Example analogy:

- Car can start, stop, honk → these are methods.

2.Difference between local and global variables.

Feature	**Local Variable**	**Global Variable**
Defined	Inside a function	Outside all functions
Scope	Only inside the function	Anywhere in the program

Lifetime	Temporary (only during function)	Until the program ends
Access	Not accessible outside function	Accessible inside and outside functions

7. Inheritance

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

- **Single Inheritance:**
  When a child class inherits properties and methods from only one parent class.
- **Multilevel Inheritance:**
  When a child class acts as a parent for another class, forming a chain of inheritance.
- **Multiple Inheritance:**
  When a single child class inherits features from more than one parent class.
- **Hierarchical Inheritance:**
  When multiple child classes inherit from the same parent class.
- **Hybrid Inheritance:**
  A combination of two or more types of inheritance in a single program.

Inheritance Type	**Description**	**Example**
Single	One parent → One child	class B(A)
Multilevel	Grandparent → Parent → Child	A → B → C
Multiple	One child → Multiple parents	class C(A, B)
Hierarchical	One parent → Multiple children	A → B, A → C
Hybrid	Combination of above types	e.g., Multiple + Multilevel

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

- **Using the super() function:**

  The super() function in Python is used inside a child class to call methods or access properties of its parent class without explicitly naming the parent. It helps in reusing code and maintaining cleaner class hierarchies, especially in multiple or multilevel inheritance.

- **Purpose:**

  It is mainly used to call the **constructor (__init__)** or other methods of the parent class from the child class.

- **Syntax:** super().method_name()

  Here, super() automatically refers to the parent class.

8. Method Overloading and Overriding Theory:

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

- Method Overloading in Python means defining multiple methods with the same name but different numbers or types of parameters.
- However, Python doesn't support traditional method overloading like some other languages (e.g., Java or C++).
- Instead, the latest defined method with the same name overrides the previous ones.

Example of Conceptual Overloading (using default arguments):

```
class Example:          def show(self,
a=None, b=None):        if a is not None
and b is not None:              print("Two
```

```
arguments:", a, b)          elif a is not
None:
        print("One argument:", a)
    else:
        print("No argument")

obj = Example()
obj.show() obj.show(10)
obj.show(10, 20)
```

Explanation:

- Only one method show() is defined.
- It handles different argument patterns using default parameters and if conditions.
- This approach simulates method overloading in Python.

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

- **Method Overriding** in Python means **redefining a method of the parent class inside the child class** with the **same name** and **same parameters**.
- It allows the **child class to provide its own version** of a method that already exists in the parent class.

Example:

```
class Parent:    def
display(self):
    print("This is the parent class method.")

class    Child(Parent):
def display(self):
    print("This is the child class method (overridden).")
```

```
obj = Child()
obj.display()
```

Output:

This is the child class method (overridden).

9. SQLite3 and PyMySQL (Database Connectors)

1.Introduction to SQLite3 and PyMySQL for database connectivity.

1. SQLite3

- **SQLite3** is a **built-in database module** in Python — no separate installation is needed.
- It stores data in a **local file (e.g., database.db)** on your computer.
- It is ideal for **small applications, testing, or local storage**.
- It uses **SQL (Structured Query Language)** to manage and query data.

2. PyMySQL

- **PyMySQL** is a **third-party library** used to connect Python with a **MySQL database**.
- It allows you to perform **database operations on a MySQL server** (local or remote).
- You need to install it first using:
- pip install PyMySQL

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

1. Using SQLite3 Example:

```
import sqlite3

# Step 1: Connect to SQLite database conn
= sqlite3.connect("student.db")

# Step 2: Create a cursor object cur
= conn.cursor()

# Step 3: Execute SQL queries
cur.execute("CREATE TABLE IF NOT EXISTS students (id INTEGER PRIMARY KEY, name TEXT, age INTEGER)") cur.execute("INSERT INTO students (name, age) VALUES ('John', 20)") cur.execute("INSERT INTO students (name, age) VALUES ('Alice', 22)")

# Step 4: Fetch data cur.execute("SELECT
* FROM students")
rows = cur.fetchall() for
row in rows:
print(row)

# Step 5: Save and close conn.commit()
conn.close()
```

- **2. Using PyMySQL**

Example:

```
import pymysql
```

```
# Step 1: Connect to MySQL database conn
= pymysql.connect(
   host="localhost",
user="root",
   password="yourpassword",
database="school"
)
# Step 2: Create cursor cur
= conn.cursor()

# Step 3: Execute SQL queries
cur.execute("CREATE TABLE IF NOT EXISTS students (id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(50), age INT)") cur.execute("INSERT INTO students (name, age) VALUES ('David', 23)") cur.execute("INSERT INTO students (name, age) VALUES ('Emma', 21)")

# Step 4: Fetch data cur.execute("SELECT
* FROM students")
data = cur.fetchall() for
record in data:
print(record)

# Step 5: Save and close
conn.commit() conn.close()
```

10. Search and Match Functions

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

1. re.match() Function

- The re.match() function **checks for a match only at the beginning** of a string.

- If the pattern is found at the **start**, it returns a **match object**; otherwise, it returns **None**.

Example: import re text =

"Hello World" result =

re.match("Hello", text)

if result: print("Match found:",

result.group()) else:

print("No match")

Output:

Match found: Hello

2. re.search() Function

- The re.search() function **scans the entire string** to find the **first occurrence** of the pattern.
- It returns a **match object** if found anywhere in the string; otherwise, **None**.

Example: import

re text =

"Welcome to

Python

programming"

result = re.search("Python", text)

if result:

print("Pattern found:", result.group())

else: print("Pattern not found")

Output:

Pattern found: Python

2.Difference between search and match.

Feature	**re.match()**	**re.search()**
Search Location	Looks only at the beginning of the string.	Searches anywhere in the entire string.
Return Value	Returns a match object only if the pattern is at the start.	Returns a match object if the pattern is found anywhere.
If Not Found	Returns None	Returns None
Use Case	When you want to check whether a string starts with a certain pattern.	When you want to find a pattern anywhere in the text.