Module 8) Advanced Python Programming (Theory)

1. Printing on Screen

1. Introduction to the print() function in Python

- print() is a **built-in Python function** used to display output on the screen.
- It can print **strings, numbers, variables, or even complex data structures** like lists and dictionaries.
- By default, print() adds a **newline** (\n) at the end, so each call moves to a new line.

Syntax:

print(object1, object2, ..., sep=' ', end='\n')

- object1, object2... → Values you want to display.
- sep → Defines separator between values (default = space " ").
- end → Defines what to print at the end (default = newline "\n").

---

2. Formatting Outputs using f-strings (Python 3.6+)

- f-strings (formatted string literals) allow embedding variables inside strings with {}.
- Start the string with f or F.

Example:

```
name = "Jay"
age = 22
print(f"My name is {name} and I am {age} years old.")
```

Output:

My name is Jay and I am 22 years old.

---

3. Formatting Outputs using .format() Method

- The str.format() method is an **older but powerful** way to format strings.
- Placeholders {} inside the string are replaced with values passed to .format().

Example:

```
name = "Jay"
age = 22
print("My name is {} and I am {} years old.".format(name, age))
```

Output:

My name is Jay and I am 22 years old.

- You can also use **positional** or **named arguments**:

```
print("My name is {0} and I am {1} years old.".format(name, age))
print("My name is {n} and I am {a} years old.".format(n=name, a=age))
```

---

2. Reading Data from Keyboard

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

- input() is a **built-in function** in Python that allows users to type data from the keyboard.
- It **always returns data as a string**.
- You can display a message inside input() to guide the user.

Syntax:

```
variable = input("Enter something: ")
```

Example:

```
name = input("Enter your name: ")
print("Hello,", name)
```

Output:

```
Enter your name: Jay
Hello, Jay
```

---

2. Converting user input into different data types

Since input() always returns a **string**, we often need to **convert it** into int, float, or other types depending on the requirement.

- **Convert to Integer (int)**

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

Output:

Enter your age: 22

Next year you will be 23

- **Convert to Float (float)**

```
price = float(input("Enter the price: "))
print("Price with tax:", price * 1.18)
```

Output:

Enter the price: 100

Price with tax: 118.0

- **Convert to String (str)** (already default, but can be explicit)

```
roll_no = str(input("Enter Roll Number: "))
print("Roll Number is:", roll_no)
```

---

3. Opening and Closing Files

1. Opening Files in Different Modes

In Python, we use the **open() function** to open a file.

Syntax:

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

- **filename** → Name (or path) of the file (e.g., "data.txt").
- **mode** → Defines how the file will be opened.

Mode	Meaning	Description
'r'	Read	Default mode. Opens file for reading. Error if file doesn't exist.
'w'	Write	Creates a new file (if not exists) or **overwrites** an existing file.
'a'	Append	Opens file for writing. Data is added to the **end** of the file.
'r+'	Read + Write	Opens file for both reading and writing. File must exist.
'w+'	Write + Read	Creates a new file or overwrites existing one. Allows reading and writing.

---

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

Example 1: Opening a file for reading (r)

```
file = open("data.txt", "r")  # open in read mode
content = file.read()        # read whole file
print(content)
file.close()
```

Example 2: Writing to a file (w)

```
file = open("data.txt", "w")   # open in write mode
file.write("Hello, Python!")   # overwrite with new text
file.close()
```

Example 3: Appending to a file (a)

```
file = open("data.txt", "a")   # open in append mode
file.write("\nNew line added.")
file.close()
```

Example 4: Reading and Writing (r+)

```
file = open("data.txt", "r+")
print("Before:", file.read())
file.write("\nExtra text")     # adds new text while keeping old data
file.close()
```

---

3. Closing Files using close()

- After completing file operations, always close the file using **close()**.
- It ensures data is saved properly and resources are freed.

Example:

```
file = open("data.txt", "w")
file.write("Closing example")
file.close()   # very important
```

- Alternatively, we can use **with statement** (best practice):

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

```
    content = file.read()
    print(content)
# file closes automatically after block ends
```

---

4. Reading and Writing Files

1. Reading from a File

After opening a file in **read mode ("r")**, we can use different methods:

(a) read() → Reads the entire file

```
file = open("data.txt", "r")
content = file.read()      # reads full content
print(content)
file.close()
```

Output (if file contains text):

```
Hello, Python!
New line added.
```

(b) readline() → Reads one line at a time

```
file = open("data.txt", "r")
line1 = file.readline()   # reads first line
line2 = file.readline()   # reads second line
print(line1)
print(line2)
file.close()
```

Output:

```
Hello, Python!
New line added.
```

(c) readlines() → Reads all lines into a list

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

lines = file.readlines()

print(lines)

file.close()
```

Output:

```
['Hello, Python!\n', 'New line added.\n']
```

---

2. Writing to a File

After opening a file in **write ("w")** or **append ("a")** mode, we can write data.

(a) write() → Writes a string to a file

```
file = open("data.txt", "w")   # overwrites old content

file.write("This is Python file handling.\n")

file.write("Second line of text.")

file.close()
```

File content after execution:

```
This is Python file handling.

Second line of text.
```

(b) writelines() → Writes a list of strings to a file

```
file = open("data.txt", "a")   # appends data

lines = ["\nLine 1", "\nLine 2", "\nLine 3"]

file.writelines(lines)

file.close()
```

- **File content after execution**:

This is Python file handling.

```
Second line of text.
Line 1
Line 2
Line 3
```

---

5. Exception Handling

1. Introduction to Exceptions

- **Exception** = An error that occurs during program execution, which interrupts the normal flow.
- Common examples:
  - ZeroDivisionError → dividing by zero
  - ValueError → invalid type conversion
  - FileNotFoundError → accessing non-existing file

Without handling, the program **crashes**.
To prevent this, Python provides **exception handling**.

---

2. Handling Exceptions using try, except, and finally

Syntax:

```
try:
    # code that may cause error
except SomeError:
    # code to handle error
finally:
    # optional, always runs (cleanup)
```

Example:

```
try:
    x = int(input("Enter a number: "))
    result = 10 / x
    print("Result:", result)
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
finally:
    print("Execution finished.")
```

Possible Outputs:

```
Enter a number: 2
```

Result: 5.0

Execution finished.

Enter a number: 0

Error: Cannot divide by zero.

Execution finished.

---

3. Handling Multiple Exceptions

You can catch different types of errors separately.

```
try:
    num = int(input("Enter number: "))
    print("10 / num =", 10/num)
except ValueError:
    print("Invalid input! Please enter a number.")
except ZeroDivisionError:
    print("Division by zero is not allowed.")
```

Outputs:

- **Input "abc" → Invalid input! Please enter a number.**
- **Input 0 → Division by zero is not allowed.**

---

4. Custom Exceptions

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

```
class NegativeNumberError(Exception):
    pass


try:
    num = int(input("Enter positive number: "))
    if num < 0:
        raise NegativeNumberError("Negative numbers are not allowed!")
    print("You entered:", num)
```

```
except NegativeNumberError as e:
    print("Custom Exception:", e)
```

Output (if user enters -5):

```
Custom Exception: Negative numbers are not allowed!
```

---

6. Class and Object (OOP Concepts)

1. Understanding Classes and Objects

- **Class**
  - A class is a **blueprint** or template for creating objects.
  - It defines **attributes** (variables) and **methods** (functions) that an object will have.
  - Created using the class keyword.
- **Object**
  - An object is an **instance** of a class.
  - Objects have their own **data** and can use methods defined in the class.

---

2. Attributes and Methods

- **Attributes** → Variables inside a class (represent properties of an object).
- **Methods** → Functions inside a class (represent actions/behaviors).

Example:

```
class Student:
    # attribute
    college = "ABC College"   # class attribute (shared)

    # constructor method
    def __init__(self, name, age):
        self.name = name      # instance attribute
        self.age = age
```

```
    # method
    def show_info(self):
        print(f"Name: {self.name}, Age: {self.age}, College: {Student.college}")

# creating objects
s1 = Student("Jay", 22)
s2 = Student("Priya", 21)

# calling method
s1.show_info()
s2.show_info()
```

Output:

```
Name: Jay, Age: 22, College: ABC College
Name: Priya, Age: 21, College: ABC College
```

---

3. Difference between Local and Global Variables

- **Global Variable**
  - Declared **outside any function/class**.
  - Can be accessed anywhere in the program (but should be used carefully).
- **Local Variable**
  - Declared **inside a function or method**.
  - Can only be accessed inside that function/method.

Example:

```
x = 100   # global variable

class Example:
    def show(self):
        y = 50   # local variable
        print("Local y:", y)
```

```
    print("Global x:", x)


obj = Example()
obj.show()
```

Output:

```
Local y: 50
Global x: 100
```

---

7. Inheritance

1. What is Inheritance?

- **Inheritance** is an OOP concept where a new class (child/derived class) can use the **properties and methods** of an existing class (parent/base class).
- Helps in **code reusability** and **extensibility**.

---

2. Types of Inheritance in Python

1. **Single Inheritance**
    - One child class inherits from one parent class.

```
class Parent:
  def show(self):
        print("This is parent class")


  class Child(Parent):
    def display(self):
      print("This is child class")


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

---

2. **Multilevel Inheritance**
    - A chain of inheritance (grandparent → parent → child).

```
class Grandparent:
  def fun1(self):
        print("Grandparent class")


   class Parent(Grandparent):
     def fun2(self):
       print("Parent class")


class Child(Parent):
     def fun3(self):
       print("Child class")


   obj = Child()
   obj.fun1()
   obj.fun2()
   obj.fun3()
```

---

3. **Multiple Inheritance**
    - A class inherits from more than one parent class.

```
class Father:
     def quality1(self):
       print("Father's quality")


   class Mother:
     def quality2(self):
       print("Mother's quality")
```

```
class Child(Father, Mother):
    def quality3(self):
        print("Child's own quality")

obj = Child()
obj.quality1()
obj.quality2()
obj.quality3()
```

---

4. **Hierarchical Inheritance**
    - Multiple child classes inherit from the same parent class.

```
class Parent:
    def feature(self):
        print("Parent feature")

class Child1(Parent):
    def feature1(self):
        print("Child1 feature")

class Child2(Parent):
    def feature2(self):
        print("Child2 feature")

obj1 = Child1()
obj1.feature()
obj1.feature1()

obj2 = Child2()
obj2.feature()
```

```
obj2.feature2()
```

---

5. **Hybrid Inheritance**
    - A combination of two or more types of inheritance.

```
class A:
    def showA(self):
        print("Class A")

class B(A):
    def showB(self):
        print("Class B")

class C(A):
    def showC(self):
        print("Class C")

class D(B, C):   # Hybrid (combination of multiple + hierarchical)
    def showD(self):
        print("Class D")

obj = D()
obj.showA()
obj.showB()
obj.showC()
obj.showD()
```

---

3. Using super() Function

- The **super() function** is used in child class to call methods or constructors of the parent class.

Example with Constructor:

```
class Parent:
    def __init__(self, name):
        self.name = name
        print("Parent constructor called")

class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)   # calling parent constructor
        self.age = age
        print("Child constructor called")

c = Child("Jay", 22)
print(c.name, c.age)
```

Output:

```
Parent constructor called
Child constructor called
Jay 22
```

---

8. Method Overloading and Overriding

1. Method Overloading

- **Definition:** Having multiple methods with the **same name but different parameters**.
- In many languages (like Java, C++), true method overloading is supported.
- In **Python**, method overloading is **not directly supported** because functions are identified by name only, not by parameters.
- However, we can achieve similar behavior by:
    - Using **default arguments**
    - Using **variable-length arguments (*args, **kwargs)**

Example (default arguments):

```
class Calculator:
```

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

calc = Calculator()
print(calc.add(10, 20))       # two arguments
print(calc.add(10, 20, 30))   # three arguments
```

Output:

```
30
60
```

Example (variable-length arguments):

```
class Calculator:
    def add(self, *numbers):
        return sum(numbers)

calc = Calculator()
print(calc.add(5, 10))
print(calc.add(5, 10, 15, 20))
```

Output:

```
15
50
```

---

2. Method Overriding

- **Definition:** Redefining a parent class method in the child class with the **same name and parameters**.
- Used when the child class needs a **different implementation** of the method.

Example:

```
class Animal:
    def sound(self):
        print("Animals make sounds")
```

```
class Dog(Animal):
    def sound(self):   # overriding parent method
        print("Dog barks")

class Cat(Animal):
    def sound(self):   # overriding parent method
        print("Cat meows")

a1 = Animal()
d1 = Dog()
c1 = Cat()

a1.sound()
d1.sound()
c1.sound()
```

Output:

```
Animals make sounds
Dog barks
Cat meows
```

---

3. Using super() in Overriding

- Sometimes we override a method but still want to use the **parent's version**.
- We can call it using super().

Example:

```
class Vehicle:
    def start(self):
        print("Vehicle started")

class Car(Vehicle):
```

```
    def start(self):
        super().start()  # call parent method
        print("Car engine started")

c = Car()
c.start()
```

Output:

```
Vehicle started
Car engine started
```

---

9. SQLite3 and PyMySQL (Database Connectors)

1. Introduction to SQLite3 and PyMySQL

- **SQLite3**
  - A lightweight, **serverless database** built into Python.
  - Stores data in a **single file** (.db or .sqlite).
  - Useful for small/medium projects, testing, or learning database connectivity.
  - Comes **pre-installed** with Python (no need to install separately).
- **PyMySQL**
  - A **Python library** used to connect to a **MySQL database server**.
  - Supports executing SQL queries from Python.
  - Requires installation:

    pip install pymysql

---

2. Connecting to SQLite3

Step 1: Import the module

```
import sqlite3
```

Step 2: Create/Connect to a database

```
conn = sqlite3.connect("student.db")   # creates file student.db
```

```
cursor = conn.cursor()
```

Step 3: Create a table

```
cursor.execute("""
CREATE TABLE IF NOT EXISTS student(
    id INTEGER PRIMARY KEY,
    name TEXT,
    age INTEGER
)
""")
```

Step 4: Insert data

```
cursor.execute("INSERT INTO student(name, age) VALUES (?, ?)", ("Jay", 22))
conn.commit()
```

Step 5: Fetch data

```
cursor.execute("SELECT * FROM student")
rows = cursor.fetchall()
for row in rows:
    print(row)
```

Step 6: Close connection

```
conn.close()
```

---

3. Connecting to MySQL using PyMySQL

Step 1: Import module

```
import pymysql
```

Step 2: Establish connection

```
conn = pymysql.connect(
    host="localhost",
    user="root",
    password="yourpassword",
    database="testdb"
```

```
)
cursor = conn.cursor()
```

Step 3: Create a table

```
cursor.execute("""
CREATE TABLE IF NOT EXISTS student(
   id INT AUTO_INCREMENT PRIMARY KEY,
   name VARCHAR(50),
   age INT
)
""")
```

Step 4: Insert data

```
cursor.execute("INSERT INTO student(name, age) VALUES (%s, %s)", ("Priya", 21))
conn.commit()
```

Step 5: Fetch data

```
cursor.execute("SELECT * FROM student")
for row in cursor.fetchall():
   print(row)
```

Step 6: Close connection

```
conn.close()
```

---

10. Search and Match Functions

Introduction

Python provides the re module (regular expressions) for pattern matching in strings.
Two important functions are:

- **re.match()**
- **re.search()**

Both are used to check if a pattern exists in a string, but their behavior is **different**.

---

1. re.match() Function

- Syntax:

  re.match(pattern, string)

- **Definition:**
  Tries to match the **pattern only at the beginning** of the string.

- Returns:
  - A **match object** if the pattern is found at the start.
  - None if not found.

Example:

```
import re


text = "Python is easy to learn"


result = re.match("Python", text)
if result:
   print("Match found:", result.group())
else:
   print("No match")
```

Output:

```
Match found: Python
```

---

2. re.search() Function

- Syntax:

  re.search(pattern, string)

- **Definition:**
  Scans the **entire string** to find the first occurrence of the pattern (not just at the beginning).

- Returns:
  - A **match object** if the pattern exists anywhere in the string.
  - None if not found.

Example:

```
import re

text = "Python is easy to learn"

result = re.search("easy", text)
if result:
    print("Search found:", result.group())
else:
    print("Not found")
```

Output:

```
Search found: easy
```

---

3. Difference between match() and search()

Feature	re.match()	re.search()
Where it checks	Only at the **beginning** of string	Scans **entire string**
If pattern not at start	Returns None	Can still return a match
Use case	When you want to check the prefix	When you want to check anywhere

---

Example showing the difference:

```
import re

text = "I love Python programming"

match_result = re.match("Python", text)   # Only checks beginning
search_result = re.search("Python", text) # Checks entire string

print("Match result:", match_result)
print("Search result:", search_result)
```

Output:

Match result: None

Search result: <re.Match object; span=(7, 13), match='Python'>
