Module - 8: Advance Python Programming

1. Printing on Screen

Q1] Introduction to the print() function in Python.

- The print() function in Python is used to **display output to the console**. It can print text, variables, numbers, or the result of expressions.
- Optional Parameters:
  - sep: Specifies how multiple objects are separated (default is a space).
  - end: Specifies what is printed at the end (default is newline \n).

```
print("A", "B", "C", sep="-")  # Output: A-B-C

print("Hello", end="!")       # Output: Hello!
```

- **Syntax:** print(object1, object2, ..., sep=' ', end='\n')
- **Examples:**

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

x = 5

print("The value of x is", x)

print(3 + 4)
```

Q2] Formatting outputs using f-strings and format().

- Python provides multiple ways to format strings for cleaner output, especially when inserting variables.

1. f-strings:

- Use ‘f’ before the string and insert variables or expressions in {}.
- Example:

```
name = "Alice"

age = 30

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

- You can also perform operations inside the placeholders:

```
print(f" In 5 years, I will be {age + 5} years old.")
```

2. str.format() method

- Use {} as placeholders and call .format() with the variables.

```
fname = "Alice"
```

```
lname=”John”
age = 30
print("My name is {} and I am {} years old.".format(name,lname, age))
```

- You can also use index numbers or named arguments:

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

2. Reading Data from Keyboard

Q3] Using the input() function to read user input from the keyboard.

- The input() function is used in Python to get input from the user via the keyboard. It always returns the input as a **string** (text), even if the user types a number.
- **Syntax:** variable = input("Prompt message")
- **Example:**

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

Q4] Converting user input into different data types (e.g., int, float, etc.).

- Since input() returns a string, you often need to convert it into another data type like int, float, etc., for calculations or logic.
- **Examples:**

```
# Integer input
age = int(input("Enter your age: "))
print("Next year, you will be", age + 1)
# Float input
price = float(input("Enter the price: "))
print("Price with tax:", price * 1.1)
# Boolean input (basic)
is_student = input("Are you a student? (yes/no): ")
if is_student.lower() == "yes":
    print("You are eligible for a discount.")
```

3. Opening and Closing Files

Q5] Opening files in different modes ('r', 'w', 'a', 'r+', 'w+').

- When working with files in Python, the open() function is used along with a mode that tells Python what you want to do with the file.

Mode	Description
'r'	Read mode (default). Opens the file for reading. File must exist.
'w'	Write mode. Creates a new file or overwrites the file if it exists.
'a'	Append mode. Opens the file for writing, adds content at the end. Creates the file if it doesn’t exist.
'r+'	Read and write. File must exist. Allows both reading and writing.
'w+'	Write and read. Overwrites existing file or creates a new one.

Examples:

```
# 'r' - read only

f = open('file.txt', 'r')

# 'w' - write only, creates or overwrites

f = open('file.txt', 'w')

# 'a' - append only

f = open('file.txt', 'a')

# 'r+' - read and write (no overwrite)

f = open('file.txt', 'r+')

# 'w+' - write and read (with overwrite)

f = open('file.txt', 'w+')
```

Q6] Using the open() function to create and access files.

- The open() function in Python is used to create, read, write, or append to a file. Its basic syntax is:
- file = open('filename', 'mode')
- After opening a file, you can use:
  - .read() – to read content,
  - .write() – to write to the file,
  - .close() – to close the file.

Example:

```
# Create and write to a file

file = open('example.txt', 'w')

file.write("Hello, this is a test file.")

file.close()
```

Example:

```
file = open('example.txt', 'r')

content = file.read()

print(content)

file.close()
```

Example:

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

    print(file.read())

# Automatically closes the file
```

Q7] Closing Files Using close()

- When working with files in Python, it's important to **close** the file after you're done using it. This ensures that:
- Resources are released properly.
- Data is saved (especially in write/append modes).
- No file corruption or memory leaks occur.

➢ **Why is close() important?**

When a file is open:

- It uses system resources.
- If you're writing to it, data may be held in a **buffer** (not saved yet).
- If you don't close it, changes might not be written properly, and the file may become **locked**.

➢ **Syntax:**

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

# do something with the file

file.close()
```

➢ **Use with Statement (Auto-Close)**

- Instead of manually calling close(), you can use a with block, which **automatically closes the file**, even if an error occurs.

Example:

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

    data = file.read()

# file is automatically closed here
```

4. Reading and Writing Files

Q8] Reading from a file using read(), readline(), readlines().

- In Python, files can be in read from or written to using the built-in open() function along with various methods:
  - To read: Use modes 'r', 'r+'
  - To write: Use modes 'w', 'a', 'w+', 'r+'
- After opening a file in read mode ('r'), you can use these methods:
- read(), readline(), readlines().
- Use readline() in loops for line-by-line processing.
  Use readlines() when you want all lines at once, e.g., for iteration.

1. read() – Reads the entire file as a single string.

```
with open('sample.txt', 'r') as f:

    data = f.read()

    print(data)
```

2. readline() – Reads one line at a time.

```
with open('sample.txt', 'r') as f:

    line1 = f.readline()

    line2 = f.readline()

    print(line1)

    print(line2)
```

3. readlines() – Reads all lines and returns them as a list of strings.

```
with open('sample.txt', 'r') as f:

    lines = f.readlines()

    print(lines)
```

Q9] Writing to a file using write() and writelines().

- To write to a file, open it in 'w', 'a', or 'w+' mode.

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

```
with open('output.txt', 'w') as f:

    f.write("This is the first line.\n")

    f.write("This is the second line.\n")
```

2. writelines() – Writes a list of strings. It does not add newline characters automatically — you must include them (\n) in your strings.

Example:

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

with open('output.txt', 'w') as f:

    f.writelines(lines)
```

SUMMARY TABLE:

Method	**Purpose**	**Returns**
read()	Read whole file	str
readline()	Read next line	str
readlines()	Read all lines	list of str
write()	Write one string	int (chars written)
writelines()	Write list of strings	None

5. Exception Handling

Q10] Introduction to exceptions and how to handle them using try, except, and finally.

Syntax:

```
try:

    # risky code

except ErrorType:

    # handle error

finally:

    # always runs
```

Example:

```
try:

    x = 5 / 0

except ZeroDivisionError:

    print("Can't divide by zero!")

finally:

    print("Done!")
```

Q11. Understanding multiple exceptions and custom exceptions.

- **Handling Multiple Exceptions:**

```
try:
    value = int("abc")
except ValueError:
    print("Value error!")
except TypeError:
    print("Type error!")
```

- **Catching Multiple Errors Together:**

```
except (ValueError, TypeError):
    print("Either a ValueError or TypeError occurred.")
```

- **Custom Exception:**

```
class TooSmallError(Exception):
    pass
def check(n):
    if n < 10:
        raise TooSmallError("Number too small")
try:
    check(5)
except TooSmallError as e:
    print("Error:", e)
```

6. Class and Object (OOP Concepts)

Q12] Understanding the concepts of classes, objects, attributes, and methods in Python.

(I) Class

- A **class** is a blueprint or template for creating objects. It defines **attributes** (variables) and **methods** (functions).

Example:

```
class Car:
    color = "prusian blue"    #attribute

    def drive(self):      #method
```

```
        print("The car is driving.")
```

(II) Object

- An object is an instance of a class. It is created using the class and can access its attributes and methods.

Example:

```
my_car = Car()       # object created from class Car

print(my_car.color)  # accessing attribute

my_car.drive()       # calling method
```

(III) Attributes

- Attributes are **variables inside a class**. They store the state or data of an object.
  - You can have class attributes (shared by all objects)
  - Or instance attributes **(**unique to each object)

Example:

```
class Dog:

    def __init__(self, name):

        self.name = name             # instance attribute


my_dog = Dog("Buddy")

print(my_dog.name)                  # Output: Buddy
```

(IV) Methods

Methods are **functions defined inside a class**. They can use and modify attributes and define object behavior.

Example:

```
class Calculator:

    def add(self, a, b):

        return a + b


calc = Calculator()

print(calc.add(3, 5))            # Output: 8
```

Q13] Difference between local and global variables.

(I)Local Variable

- Defined inside a function
- Accessible only within that function

Example:

```
def greet():
   name = "Alice"  # local variable
   print(name)
```

(II) Global Variable

- Defined outside any function
- Can be accessed anywhere in the program

Example:

```
name = "Bob"          # global variable
def greet():
   print(name)         # can access global variable
greet()
```

➢ **Modifying Global Variables Inside Functions**

- If you want to **change** a global variable inside a function, use the global keyword.

```
x = 10
def update():
   global x
   x = 20
update()
print(x)       # Output: 20
```

7. Inheritance

Q14] Single, Multilevel, Multiple, Hierarchical, and Hybrid inheritance in Python.

➢ Inheritance allows a class (child) to **inherit properties and methods** from another class (parent).

(1) Single Inheritance

➢ One child class inherits from one parent class.

```
class Animal:
    def sound(self):
        print("Animal sound")


class Dog(Animal):
    def bark(self):
        print("Dog barks")
d = Dog()
d.sound()   # Inherited
d.bark()    # Own method
```

(2) Multilevel Inheritance

- A class inherits from a class, which in turn inherits from another class.

Example:

```
class Animal:
    def sound(self):
        print("Animal sound")
class Dog(Animal):
    def bark(self):
        print("Dog barks")
class Puppy(Dog):
    def weep(self):
        print("Puppy weeps")
p = Puppy()
p.sound()          # From Animal
p.bark()           # From Dog
p.weep()           # Own
```

(3) Multiple Inheritance

- A child class inherits from **more than one** parent class.

Example:

```
class Father:
    def skills(self):
        print("Gardening")
class Mother:
    def traits(self):
        print("Cooking")
class Child(Father, Mother):
    pass
c = Child()
c.skills()
c.traits()
```

(4) Hierarchical Inheritance

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

Example:

```
class Vehicle:
    def engine(self):
        print("Engine started")
class Car(Vehicle):
    pass
class Bike(Vehicle):
    pass
c = Car()
b = Bike()
c.engine()
b.engine()
```

(5) Hybrid Inheritance

- A combination of **two or more types** of inheritance.

```
class A:
    def show(self):
        print("Class A")
class B(A):
    pass
```

```
class C(A):
    pass
class D(B, C):  # Inherits from both B and C (multilevel + multiple)
    pass
d = D()
d.show()
```

Q15] Using the super() function to access properties of the parent class.

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

i.Calling Parent Constructor

```
class Animal:
    def __init__(self, name):
        self.name = name
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)   # Call parent __init__
        self.breed = breed
d = Dog("Tommy", "Labrador")
print(d.name, d.breed)
```

ii. Calling Parent Methods

```
class A:
    def greet(self):
        print("Hello from A")

class B(A):
    def greet(self):
        super().greet()  # Call parent method
        print("Hello from B")
b = B()
b.greet()
```

8. Method Overloading and Overriding

Q16] Method overloading: defining multiple methods with the same name but different parameters.

(I)Method Overloading:

Method Overloading means defining multiple methods with the same name but different parameters.

- In some languages like Java, this is done by changing the number/type of arguments.
- But in Python, **true method overloading is not directly supported** because Python methods can accept *any* number of arguments using *args and **kwargs.

This works because Python uses **default arguments** or *args to mimic overloading.

Simulating Method Overloading in Python

```
class Greet:
    def hello(self, name=None):
        if name:
            print(f"Hello, {name}!")
        else:
            print("Hello!")
g = Greet()
g.hello()                 # Output: Hello!
g.hello("Alice")          # Output: Hello, Alice!
```

Q17] Method overriding: redefining a parent class method in the child class.

Method Overridding:

- Method Overriding means redefining a method from the parent class in the child class to provide a different implementation.

Example:

```
class Animal:
    def speak(self):
        print("Animal makes a sound")
class Dog(Animal):
    def speak(self):  # overriding the method
        print("Dog barks")
```

```
d = Dog()
d.speak()  # Output: Dog barks
```

- **Using super() with Overriding**

You can still call the parent version of the overridden method using super():

Example:

```
class Dog(Animal):
    def speak(self):
        super().speak()
        print("Dog barks")
d = Dog()
d.speak()
```

9. SQLite3 and PyMySQL (Database Connectors)

Q18] Introduction to SQLite3 and PyMySQL for database connectivity.

(I)SQLite3

- **SQLite** is a lightweight, serverless, self-contained SQL database engine.
- It stores the entire database in a single file on the disk.
- Ideal for **s**mall to medium applications, prototyping, or embedded systems.
- Python has a built-in module sqlite3 for connecting and interacting with SQLite databases.

Example:

```
import sqlite3
# Connect to a database (or create one if it doesn't exist)
conn = sqlite3.connect('example.db')
cursor = conn.cursor()
# Create a table
cursor.execute('CREATE TABLE IF NOT EXISTS users (id INTEGER, name TEXT)')
# Commit and close
conn.commit()
conn.close()
```

(II) PyMySQL

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

- Suitable for web applications, large datasets, and multi-user environments.
- Requires the MySQL server to be installed and running.

Example:

```
import pymysql
# Connect to MySQL
conn = pymysql.connect (   host='localhost',
        user='root',
        password='yourpassword',
        database='testdb'  )
        cursor = conn.cursor()
# Create a table
cursor.execute('CREATE TABLE IF NOT EXISTS users (id INT, name VARCHAR(100))')
# Commit and close
conn.commit()
conn.close()
```

Q19] Creating and executing SQL queries from Python using these connectors.

- **With SQLite3:**

```
import sqlite3
conn = sqlite3.connect('example.db')
cursor = conn.cursor()
# Insert data
cursor.execute("INSERT INTO users (id, name) VALUES (?, ?)", (1, 'Alice'))
# Select data
cursor.execute("SELECT * FROM users")
rows = cursor.fetchall()
for row in rows:
   print(row)
conn.commit()
conn.close()
```

- **With PyMySQL:**

```
import pymysql
```

```
conn = pymysql.connect ( host='localhost',
    user='root',
    password='yourpassword',
    database='testdb'    )
cursor = conn.cursor()
# Insert data
cursor.execute("INSERT INTO users (id, name) VALUES (%s, %s)", (1, 'Alice'))
# Select data
cursor.execute("SELECT * FROM users")
rows = cursor.fetchall()
for row in rows:
    print(row)
conn.commit()
conn.close()
```

10. Search and Match Functions

Q20] Using re.search() and re.match() functions in Python's re module for pattern matching.

➢ The re module provides support for **regular expressions** in Python.

re.search()

- Searches the **entire string** for a match.
- Returns a match object if found; otherwise, returns None.

re.match()

- Checks for a match **only at the beginning** of the string.

Example:

```
import re
text = "Hello, welcome to Python!"
# re.search
result1 = re.search("welcome", text)
print(result1.group())                  # Output: welcome
# re.match
result2 = re.match("Hello", text)
      print(result2.group())                  # Output: Hello
```

```
result3 = re.match("welcome", text)
print(result3)                # Output: None (because it doesn't start the string)
```

Q21] Difference between search and match.

Feature	**re.search()**	**re.match()**
Scope	Scans the **entire string**	Matches **only at the beginning**
Use Case	Useful when pattern can appear anywhere	Useful when pattern must be at start
Return	Match object or None	Match object or None