Module 8: Advance Python Programming

Printing on Screen

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

Ans: The print() function in Python is used to display output on the screen. It is commonly used for debugging, displaying messages, and printing variable values. The basic syntax is:

print("Hello, World!")

Key Features of print():

1. **Printing Strings and Numbers**:

   python
   CopyEdit
   print("Python is fun!")
   print(100)

2. **Printing Multiple Values**:

   print("The sum of", 5, "and", 10, "is", 5 + 10)

   - By default, print() separates values with a space.

3. **Using the sep Parameter**:

   print("Hello", "World", sep="-")

   - This changes the separator from a space to -.

4. **Using the end Parameter**:

   print("Hello", end=" ")
   print("World")

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

Ans:

1. Using f-strings (Python 3.6+)

F-strings (formatted string literals) allow inserting variables inside a string using {}.

name = "Alice"
age = 25
print(f"My name is {name} and I am {age} years old.")

- Expressions can also be evaluated inside {}:

  print(f"5 + 10 = {5 + 10}")

- Formatting with decimal places:

  pi = 3.14159
  print(f"Value of π: {pi:.2f}") # Outputs: 3.14

2. Using format() Method

Before f-strings, .format() was commonly used.

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

- Using positional arguments:

  print("The numbers are {1}, {0}, and {2}".format(10, 20, 30))

- Using named placeholders:

  print("My name is {name} and I am {age} years old.".format(name="Charlie", age=28))

Reading Data from Keyboard

(Q1) Using the input() function to read user input from the keyboard.

Ans: The input() function in Python is used to take input from the user through the keyboard. The input is always returned as a string.

Basic Syntax

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

- The text inside input() acts as a prompt.
- The entered value is stored in the variable user_input.

Example: Getting User Input

name = input("Enter your name: ")
age = input("Enter your age: ")
print(f"Hello {name}, you are {age} years old.")

(Q2) Converting user input into different data types (e.g., int, float, etc.).

Ans: Since input() always returns a string, we often need to convert it to other data types, such as integers (int), floating-point numbers (float), etc.

1. Converting to an Integer (int)

```
age = int(input("Enter your age: "))  # Convert input to an integer
print(f"In 5 years, you will be {age + 5} years old.")
```

- If the user enters 25, the value is stored as an integer.
- If a non-numeric value is entered, it will cause an error.

2. Converting to a Float (float)

```
price = float(input("Enter the product price: "))  # Convert input to a float
print(f"The price after tax is: {price * 1.1:.2f}")
```

- If the user enters 50.5, the value is stored as 50.5 (float).

3. Converting to a Boolean (bool)

```
response = bool(int(input("Enter 1 for Yes, 0 for No: ")))
print("You selected:", response)
```

- If the user enters 1, it converts to True.
- If the user enters 0, it converts to False.

Handling Conversion Errors

Since int(input()) or float(input()) may cause errors if the user enters non-numeric data, it's good practice to handle exceptions.

```
try:
    num = int(input("Enter a number: "))
    print(f"You entered: {num}")
except ValueError:
    print("Invalid input! Please enter a valid number.")
```

Opening and Closing Files

(Q1) Opening files in different modes ('r', 'w', 'a', 'r+', 'w+').

Ans: Python provides several modes for opening files using the open() function. Each mode determines how the file is accessed.

Mode	Meaning	Behavior
'r'	Read	Default mode. Opens file for reading; errors if the file does not exist.
'w'	Write	Creates a new file or overwrites an existing file.
'a'	Append	Opens file for appending; creates file if it does not exist.
'r+'	Read & Write	Reads and writes to an existing file (file must exist).
'w+'	Write & Read	Creates a new file or overwrites an existing file, allowing both reading and writing.

Examples of File Modes:

```
# Read mode ('r')
with open("example.txt", "r") as file:
    content = file.read()
    print(content)

# Write mode ('w') - Overwrites existing file or creates a new one
with open("example.txt", "w") as file:
    file.write("This is a new file.")

# Append mode ('a') - Adds new content without deleting existing data
with open("example.txt", "a") as file:
    file.write("\nAppending a new line.")

# Read and Write mode ('r+')
with open("example.txt", "r+") as file:
    data = file.read()
    file.write("\nAdding new content to the file.")

# Write and Read mode ('w+') - Overwrites the file but allows reading
with open("example.txt", "w+") as file:
    file.write("New content.")
    file.seek(0)  # Move to the beginning of the file
    print(file.read())  # Read the newly written content
```

(Q2) Using the open() function to create and access files.

Ans: The open() function is used to create and access files.

Example: Creating and Writing to a File

```
file = open("sample.txt", "w")  # Opens or creates the file in write mode
file.write("Hello, this is a test file.")
file.close()  # Always close the file after writing
```

 If the file does not exist, w mode will create it.

Example: Reading a File

```
file = open("sample.txt", "r")
content = file.read()
print(content)
file.close()
```

 If the file does not exist, opening in 'r' mode will raise an error.

Using with Statement (Recommended)

```
with open("sample.txt", "r") as file:
    print(file.read())  # No need to manually close the file
```

(Q3) Closing files using close().

Ans: After working with a file, it is important to close it using close() to free system resources.

Example: Manually Closing a File

```
file = open("data.txt", "w")
file.write("This is some data.")
file.close()
```

 Forgetting to close a file can lead to data loss or memory leaks.

Why Use with open() Instead of close()?

Using with open() automatically closes the file, even if an error occurs. This is the best practice for file handling.

```
with open("data.txt", "r") as file:
    content = file.read()
    print(content)  # File closes automatically when exiting the block
```

Reading and Writing Files

(Q1) Reading from a file using read(), readline(), readlines().

Ans: Python provides multiple ways to read content from a file:

1. Using read()

Reads the entire file content as a single string.

```
with open("sample.txt", "r") as file:
    content = file.read()
```

```
print(content)  # Prints the entire file content
```

 You can also specify the number of characters to read: file.read(10) reads the first 10 characters.

2. Using readline()

Reads one line at a time.

```
with open("sample.txt", "r") as file:
    line1 = file.readline()
    print(line1)  # Prints the first line
    line2 = file.readline()
    print(line2)  # Prints the second line
```

 Useful when processing large files line by line.

3. Using readlines()

Reads all lines and returns them as a list.

```
with open("sample.txt", "r") as file:
    lines = file.readlines()
    print(lines)  # Prints a list of all lines in the file
```

 Each line in the list includes the newline character (\n). You can iterate over the list to process each line separately.

```
with open("sample.txt", "r") as file:
    for line in file.readlines():
        print(line.strip())  # Removes extra newline characters
```

(Q2) Writing to a file using write() and writelines().

Ans:

1. Using write()

Writes a string to a file.

```
with open("output.txt", "w") as file:
    file.write("Hello, World!\n")
    file.write("This is a second line.\n")
```

 If the file exists, it will be overwritten. Use 'a' mode to append instead.

2. Using writelines()

Writes a list of strings to a file.

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

with open("output.txt", "w") as file:
    file.writelines(lines)
```

 Make sure each string in the list includes \n if you want line breaks.

Appending Data ('a' Mode)

To add content to an existing file instead of overwriting it, use 'a' (append mode).

```
with open("output.txt", "a") as file:
    file.write("Appending a new line.\n")
```

Exception Handling

(Q1) Introduction to exceptions and how to handle them using try, except, and finally.

Ans:

What Are Exceptions?

An exception is an error that occurs during the execution of a program, which interrupts the normal flow. Python provides exception handling to prevent the program from crashing.

Handling Exceptions with try and except

The try block contains the code that may cause an exception, and the except block handles the error.

Example: Handling a Division by Zero Error

```
try:
    result = 10 / 0  # This will cause a ZeroDivisionError
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
```

 Without exception handling, the program would crash with an error message.

Using finally Block

The finally block runs **regardless** of whether an exception occurred or not.

```
try:
    file = open("data.txt", "r")  # Attempt to open a file
    content = file.read()
except FileNotFoundError:
    print("Error: File not found!")
finally:
    print("This block always executes.")
```

(Q2) Understanding multiple exceptions and custom exceptions.

Ans:

Handling Multiple Exceptions

We can handle different exceptions separately or catch all exceptions using Exception.

Example: Catching Multiple Specific Exceptions

```
try:
    num = int(input("Enter a number: "))  # Could raise ValueError
    result = 10 / num  # Could raise ZeroDivisionError
except ValueError:
    print("Invalid input! Please enter a valid number.")
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
```

 Each except block handles a specific error type.

Example: Catching All Exceptions (Not Recommended)

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except Exception as e:
    print(f"An error occurred: {e}")  # Prints the exact error message
```

 Catching all exceptions is useful for debugging but should be avoided in production code.

Custom Exceptions

Python allows creating custom exceptions by defining a class that inherits from Exception.

Example: Creating a Custom Exception

```
class NegativeNumberError(Exception):
    """Custom exception for negative numbers"""
    pass

try:
    num = int(input("Enter a positive number: "))
    if num < 0:
        raise NegativeNumberError("Negative numbers are not allowed!")
except NegativeNumberError as e:
    print(f"Custom Error: {e}")
```

Class and Object (OOP Concepts)

(Q1) Understanding the concepts of classes, objects, attributes, and methods in Python.

Ans:

1. What is a Class?

A class is a blueprint for creating objects. It defines **attributes** (data) and **methods** (functions) that describe the behavior of the object.

```
class Car:
    # Class attribute (shared by all instances)
    wheels = 4

    # Constructor (__init__) to initialize object attributes
    def __init__(self, brand, model):
        self.brand = brand  # Instance attribute
        self.model = model  # Instance attribute

    # Method (Function inside a class)
    def show_details(self):
        print(f"Car: {self.brand} {self.model}, Wheels: {self.wheels}")
```

2. What is an Object?

An **object** is an instance of a class. It has **attributes** (variables) and can perform **methods** (functions).

```
# Creating objects from the class
car1 = Car("Toyota", "Corolla")
car2 = Car("Honda", "Civic")

# Accessing attributes
print(car1.brand)  # Output: Toyota

# Calling a method
car2.show_details()  # Output: Car: Honda Civic, Wheels: 4
```

 Each object has its own instance attributes (brand and model), but they share the class attribute wheels.

Key Concepts

Concept	Description
Class	A blueprint for creating objects.
Object	An instance of a class.
Attribute	Variables that store data for an object.
Method	A function inside a class that operates on the object.

(Q2) Difference between local and global variables.

Ans:

1. Local Variables

- Defined **inside a function**.
- Accessible **only within** that function.

```
def greet():
    message = "Hello, World!"  # Local variable
    print(message)

greet()
# print(message)  # ✕Error! 'message' is not accessible outside the
function
```

2. Global Variables

- Defined **outside** any function or class.
- Accessible **throughout** the program.

```
global_message = "Hello from global scope!"  # Global variable

def greet():
    print(global_message)  # Accessible inside the function

greet()
print(global_message)  # Accessible outside the function
```

3. Modifying Global Variables Inside Functions

By default, modifying a global variable inside a function **creates a new local variable** instead of modifying the original.
To modify the global variable, use the `global` keyword.

```
counter = 0  # Global variable

def increment():
    global counter  # Access and modify global variable
    counter += 1

increment()
print(counter)  # Output: 1
```

 Without `global`, Python would treat `counter` as a new local variable inside the function.

Key Differences Between Local and Global Variables

Feature	Local Variable	Global Variable
Defined in	Inside a function	Outside all functions
Scope	Available only within the function	Available throughout the script
Lifetime	Exists only during function execution	Exists as long as the program runs
Modification in function	Allowed	Requires `global` keyword

Inheritance

(Q1) Single, Multilevel, Multiple, Hierarchical, and Hybrid inheritance in Python.

Ans: Inheritance allows a class to inherit properties and behaviors (methods) from another class. Python supports different types of inheritance:

1. Single Inheritance

A child class inherits from a single parent class.

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):  # Dog inherits from Animal
    def bark(self):
        print("Dog barks")

# Creating an object of Dog class
dog = Dog()
dog.speak()  # Inherited method
dog.bark()   # Dog's own method
```

 The Dog class gets access to the speak() method from Animal.

2. Multilevel Inheritance

A child class inherits from a parent class, which itself inherits from another parent class.

```
class Animal:
    def speak(self):
        print("Animal speaks")
```

```
class Mammal(Animal):  # Inherits from Animal
    def walk(self):
        print("Mammal walks")

class Dog(Mammal):  # Inherits from Mammal
    def bark(self):
        print("Dog barks")

# Creating an object of Dog class
dog = Dog()
dog.speak()  # Inherited from Animal
dog.walk()   # Inherited from Mammal
dog.bark()   # Dog's own method
```

 The Dog class inherits from Mammal, which inherits from Animal, forming a chain.

3. Multiple Inheritance

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

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Wild:
    def habitat(self):
        print("Lives in the wild")

class Lion(Animal, Wild):  # Inherits from both Animal and Wild
    def roar(self):
        print("Lion roars")

# Creating an object of Lion class
lion = Lion()
lion.speak()   # Inherited from Animal
lion.habitat() # Inherited from Wild
lion.roar()    # Lion's own method
```

 Multiple inheritance allows a class to combine functionality from multiple parent classes.

4. Hierarchical Inheritance

Multiple child classes inherit from the same parent class.

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):  # Dog inherits from Animal
    def bark(self):
        print("Dog barks")
```

```
class Cat(Animal):  # Cat also inherits from Animal
    def meow(self):
        print("Cat meows")

# Creating objects
dog = Dog()
dog.speak()  # Inherited from Animal
dog.bark()   # Dog's own method

cat = Cat()
cat.speak()  # Inherited from Animal
cat.meow()   # Cat's own method
```

 Both Dog and Cat inherit from Animal, but they do not inherit from each other.

5. Hybrid Inheritance

A combination of two or more types of inheritance.

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Mammal(Animal):  # Single Inheritance
    def walk(self):
        print("Mammal walks")

class Bird(Animal):  # Another child class of Animal (Hierarchical)
    def fly(self):
        print("Bird flies")

class Bat(Mammal, Bird):  # Multiple Inheritance (Combining Mammal & Bird)
    def unique(self):
        print("Bat is a flying mammal")

# Creating an object of Bat class
bat = Bat()
bat.speak()  # From Animal
bat.walk()   # From Mammal
bat.fly()    # From Bird
bat.unique() # Bat's own method
```

(Q2) Using the super() function to access properties of the parent class.

Ans : The `super()` function allows us to call methods from the **parent class** inside the child class. It is commonly used in **method overriding**.

Example: Using super() in Method Overriding

```
class Animal:
    def speak(self):
```

```
        print("Animal makes a sound")

class Dog(Animal):
    def speak(self):
        super().speak()  # Call the parent class method
        print("Dog barks")

# Creating an object
dog = Dog()
dog.speak()
```

Output:

```
Animal makes a sound
Dog barks
```

 The Dog class overrides the speak() method but also calls the parent method using super().speak().

Example: Using super().__init__() to Call Parent Constructor

```
class Person:
    def __init__(self, name):
        self.name = name

class Employee(Person):
    def __init__(self, name, job):
        super().__init__(name)  # Call parent constructor
        self.job = job

    def show(self):
        print(f"Name: {self.name}, Job: {self.job}")

# Creating an object
emp = Employee("Alice", "Software Engineer")
emp.show()
```

Method Overloading and Overriding

(Q1) Method overloading: defining multiple methods with the same name but different parameters.

Ans: Method overloading refers to defining multiple methods with the same name but different parameters. However, **Python does not support true method overloading** like other languages (e.g., Java or C++). Instead, we can achieve similar behavior using **default arguments** or *args and **kwargs.

Simulating Method Overloading Using Default Arguments

```
class MathOperations:
    def add(self, a, b=0, c=0):
        return a + b + c  # Handles 1, 2, or 3 arguments
```

```
# Creating object
math = MathOperations()
print(math.add(5))         # Output: 5  (Uses default values for b and c)
print(math.add(5, 10))     # Output: 15 (Uses default value for c)
print(math.add(5, 10, 15)) # Output: 30 (Uses all values)
```

 This mimics overloading by allowing different numbers of parameters.

Using *args for Method Overloading

```
class MathOperations:
    def add(self, *args):  # Accepts any number of arguments
        return sum(args)

math = MathOperations()
print(math.add(5))           # Output: 5
print(math.add(5, 10))       # Output: 15
print(math.add(5, 10, 15, 20)) # Output: 50
```

(Q2) Method overriding: redefining a parent class method in the child class.

Ans: Method overriding occurs when a child class **redefines** a method from the parent class to provide its own implementation.

Example: Overriding a Parent Class Method

```
class Animal:
    def speak(self):
        print("Animal makes a sound")

class Dog(Animal):
    def speak(self):  # Overriding parent method
        print("Dog barks")

# Creating objects
animal = Animal()
animal.speak()  # Output: Animal makes a sound

dog = Dog()
dog.speak()  # Output: Dog barks (overrides the parent method)
```

 The Dog class provides its own speak() method, replacing the speak() method from Animal.

Using super() to Call the Parent Method

```
class Animal:
    def speak(self):
        print("Animal makes a sound")

class Dog(Animal):
    def speak(self):
```

```
        super().speak()  # Calls parent class method
        print("Dog barks")

dog = Dog()
dog.speak()
```

Output:

```
Animal makes a sound
Dog barks
```

 The super().speak() calls the parent class method before executing the child's method.

Key Differences Between Overloading and Overriding

Feature	Method Overloading	Method Overriding
Definition	Same method name, different parameters	Same method name, same parameters
Python Support	Not directly supported	Fully supported
Implemented Using	Default arguments, *args, **kwargs	Redefining method in child class
Inheritance Required?	No	Yes
Purpose	Multiple ways to call the same method	Modify behavior of inherited method

SQLite3 and PyMySQL (Database Connectors)

(Q1) Introduction to SQLite3 and PyMySQL for database connectivity.

Ans: Python provides multiple ways to connect with databases. Two popular libraries are:

1. **SQLite3** – A lightweight, file-based database (built into Python).
2. **PyMySQL** – Used to connect to **MySQL databases**.

1. SQLite3 (Built-in, Lightweight Database)

- **No server required** – It stores data in a local .db file.
- **Best for small projects** or applications that don't require a separate database server.
- **Built into Python** – No need to install anything separately.

Example: Connecting to SQLite3

```
import sqlite3

# Connect to a database (or create one if it doesn't exist)
conn = sqlite3.connect("my_database.db")

# Create a cursor object to execute SQL queries
```

```
cursor = conn.cursor()

print("SQLite3 Connection Successful!")

# Close the connection
conn.close()
```

 This will create a local database file my_database.db.

2. PyMySQL (For Connecting to MySQL Databases)

- **Requires a running MySQL server**.
- **Supports remote database connections**.
- **Needs to be installed** using:

```
nginx
CopyEdit
pip install pymysql
```

Example: Connecting to MySQL with PyMySQL

```
import pymysql

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

# Create a cursor object
cursor = conn.cursor()

print("MySQL Connection Successful!")

# Close the connection
conn.close()
```

(Q2) Creating and executing SQL queries from Python using these connectors.

Ans:

1. Creating a Table and Inserting Data in SQLite3

```
import sqlite3

# Connect to the database
conn = sqlite3.connect("my_database.db")
cursor = conn.cursor()

# Create a table
cursor.execute("""
```

```
    CREATE TABLE IF NOT EXISTS users (
        id INTEGER PRIMARY KEY,
        name TEXT,
        age INTEGER
    )
""")

# Insert data
cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)",
("Alice", 25))
conn.commit()  # Save changes

print("Data inserted successfully!")

# Close the connection
conn.close()
```

 Use `?` as a placeholder to prevent SQL injection.

2. Creating a Table and Inserting Data in MySQL

```
import pymysql

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

# Create a table
cursor.execute("""
    CREATE TABLE IF NOT EXISTS users (
        id INT AUTO_INCREMENT PRIMARY KEY,
        name VARCHAR(100),
        age INT
    )
""")

# Insert data
cursor.execute("INSERT INTO users (name, age) VALUES (%s, %s)",
("Bob", 30))
conn.commit()

print("Data inserted successfully!")

# Close the connection
conn.close()
```

 Use `%s` for parameterized queries to prevent SQL injection in MySQL.

3. Retrieving Data from the Database

SQLite3 Example: Fetching Data

```
import sqlite3

conn = sqlite3.connect("my_database.db")
cursor = conn.cursor()

# Fetch all users
cursor.execute("SELECT * FROM users")
rows = cursor.fetchall()

for row in rows:
    print(row)

conn.close()
```

PyMySQL Example: Fetching Data

```
import pymysql

conn = pymysql.connect(
    host="localhost",
    user="root",
    password="mypassword",
    database="my_database"
)
cursor = conn.cursor()

cursor.execute("SELECT * FROM users")
rows = cursor.fetchall()

for row in rows:
    print(row)

conn.close()
```

Key Differences Between SQLite3 and PyMySQL

Feature	SQLite3	PyMySQL
Storage	File-based (`.db`)	Server-based (MySQL)
Installation	Built into Python	Needs `pip install pymysql`
Best For	Small projects, local apps	Large projects, web applications
Performance	Fast for single-user apps	Better for multiple users

Search and Match Functions

(Q1) Using `re.search()` and `re.match()` functions in Python's `re` module for pattern matching.

Ans: Python's re module provides powerful tools for working with **regular expressions**. Two important functions are:

1. **re.match()** – Checks if the pattern matches **only at the beginning** of the string.
2. **re.search()** – Searches for the pattern **anywhere in the string** and returns the first match.

1. Using re.match()

The re.match() function checks if the pattern matches **from the start of the string**.

```
python
CopyEdit
import re

pattern = r"\d+"  # Looks for one or more digits
text = "123 is a number"

match = re.match(pattern, text)

if match:
    print("Match found:", match.group())  # Output: Match found: 123
else:
    print("No match")
```

 *Since 123 appears at the **beginning**, re.match() returns a match.*

2. Using re.search()

The re.search() function **scans the entire string** to find the first occurrence of the pattern.

```
python
CopyEdit
import re

pattern = r"\d+"  # Looks for one or more digits
text = "The number is 123"

search = re.search(pattern, text)

if search:
    print("Search found:", search.group())  # Output: Search found: 123
else:
    print("No match")
```

(Q2) Difference between search and match.

Ans:

Feature	`re.match()`	`re.search()`
Search Scope	Checks only at the **start** of the string	Scans the **entire string**
Returns	Match object if found at position 0	Match object if found anywhere
Example (Pattern: `\d+`)	`"123abc"` → ✅Match	`"abc 123"` → ✅Search (finds `123`)

Example: Showing the Difference

```
import re

pattern = r"\d+"
text = "Hello 123 world"

match_result = re.match(pattern, text)
search_result = re.search(pattern, text)

print("Match Result:", match_result)  # Output: None (no match at
start)
print("Search Result:", search_result.group())  # Output: 123 (found
in text)
```

 `re.match()` returns `None` because there's no number at the start, but `re.search()` finds `123` in the string.