1. Printing on Screen

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

Ans:

Introduction to print() Function in Python

- Definition: A built-in function used to display output on the console.
- Syntax:
    - print(*objects, sep=' ', end='\n')
- **Parameters**:
    - `*objects`: Values to print.
    - `sep`: Separator (default space).
    - `end`: String after output (default newline).
- **Example**:
    - print("Hello", "World", sep="-", end="!")

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

Ans:

1. Using f-strings (Python 3.6+):

- Syntax:

```
f"string {variable}"
```

- Example:

```
name = "Mitesh"

age = 20

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

2. Using format():

- Syntax:

"string {}".format(value)

- Example:

  name = "Mitesh"

  age = 20

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

2 Reading Data from Keyboard

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

Ans:

input() Function in Python

- Definition: Built-in function to take input from the user as a string.
- Syntax:
  - variable = input("Prompt message")

Examples:

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

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

Ans:

- input() always returns **string** → must be converted for calculations.

  Examples:

  ```
  # Integer conversion
  ```

```
age = int(input("Enter your age: "))

# Float conversion
price = float(input("Enter product price: "))

# Boolean conversion
flag = bool(int(input("Enter 1 for True, 0 for False: ")))
```

3. Opening and Closing Files

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

Ans:

- **r** → Read (default). Error if file not found.
- **w** → Write. Creates new file or overwrites existing.
- **a** → Append. Adds data at end, keeps existing content.
- **r+** → Read + Write. File must exist.
- **w+** → Write + Read. Creates new or overwrites existing.

Example:

```
f = open("data.txt", "r")   # read

f = open("data.txt", "w")   # write

f = open("data.txt", "a")   # append

f = open("data.txt", "r+")  # read+write

f = open("data.txt", "w+")  # write+read
```

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

Ans:

open() Function in Python

- Definition: Used to create, open, and access files.
- Syntax:
  - f = open("filename", "mode")
- Common Modes:
  - "r" → Read
  - "w" → Write (creates/overwrites)
  - "a" → Append
  - "x" → Create new file (error if exists)

Example:

```
# Create/Write

f = open("test.txt", "w")

f.write("Hello, File!")

f.close()



# Read

f = open("test.txt", "r")

print(f.read())

f.close()
```

Q) Closing files using close()

Ans:

Closing Files using close()

- Purpose: Frees system resources and ensures all data is saved properly.
- Syntax:

```
f.close()
```

Example:

```
f = open("test.txt", "w")

f.write("Hello World")

f.close()   # file is now closed
```

4 . Reading and Writing Files

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

Ans:

- **read()** → Reads entire file (or specific number of characters).

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

- **readline()** → Reads one line at a time.

```
f = open("test.txt", "r")

line1 = f.readline()

print(line1)

f.close()
```

- **readlines()** → Reads all lines into a list.

```
f = open("test.txt", "r")
lines = f.readlines()
print(lines)
f.close()
```

Q) Writing to a file using write() and writelines()

Ans:

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

```
f = open("test.txt", "w")

f.write("Hello World\n")

f.close()
```

- **writelines()** → Writes a list of strings (no newline added automatically).

```
f = open("test.txt", "w")
f.writelines(["Line1\n", "Line2\n", "Line3\n"])
f.close()
```

5 . Exception Handling

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

Ans:

Introduction to Exceptions in Python

- Exception: An error that occurs during program execution, which stops the normal flow (e.g., division by zero, file not found).
- Handling: Done using try, except, and finally.

Syntax:

```
try:
    # code that may cause error
except SomeError:
    # handle error
finally:
    # always executes (cleanup code)
```

Example:

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
finally:
    print("Execution complete")
```

Q) Understanding multiple exceptions and custom exceptions.

Ans:

- Multiple Exceptions: Handle more than one error type using multiple except blocks or a tuple.

```
try:
    x = int("abc")
except ValueError:
    print("Invalid conversion")
except ZeroDivisionError:
    print("Division by zero")
```

- Custom Exceptions: User-defined errors created by subclassing Exception.

```
class MyError(Exception):
    pass
try:
    raise MyError("Custom error occurred")
except MyError as e:
    print(e)
```

6. Class and Object (OOP Concepts)

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

Ans:

- **Class**: A blueprint for creating objects (defines structure and behavior).

```
class Car:
    pass
```

- **Object**: An instance of a class.

```
my_car = Car()
```

- **Attributes**: Variables that hold data inside a class/object.

```
class Car:
    def __init__(self, brand, color):
        self.brand = brand      # attribute
        self.color = color
```

- **Methods**: Functions defined inside a class that define behavior

```
class Car:
    def __init__(self, brand):
        self.brand = brand
    def drive(self):          # method
        print(f"{self.brand} is driving")

c = Car("Toyota")
c.drive()
```

Q) Difference between local and global variables.

Ans:

- **Local Variable**: Declared inside a function, accessible only within that function.
- **Global Variable**: Declared outside all functions, accessible throughout the program.

Example:

```
x = 10   # global variable

def test():
    y = 5   # local variable
    print("Local:", y)
    print("Global inside function:", x)

test()
print("Global outside function:", x)
```

7. Inheritance

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

Ans:

- **Single Inheritance**

  One child class inherits from one parent class.

  ```
  class A: pass
  class B(A): pass
  ```

- **Multilevel Inheritance**

  A chain of inheritance (child → parent → grandparent).

  ```
  class A: pass
  class B(A): pass
  class C(B): pass
  ```

- **Multiple Inheritance**

  Child class inherits from multiple parent classes.

  ```
  class A: pass
  class B: pass
  class C(A, B): pass
  ```

- **Hierarchical Inheritance**

  Multiple child classes inherit from the same parent class.

  ```
  class A: pass
  class B(A): pass
  ```

```
class C(A): pass
```

- **Hybrid Inheritance**

  Combination of two or more types of inheritance.

```
class A: pass

class B(A): pass

class C(A): pass

class D(B, C): pass
```

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

Ans:

super() Function in Python

- Definition: super() is used to call methods or access properties of the parent class from the child class.
- Purpose: Helps in reusing parent class code without explicitly naming the parent.

Example:

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


class Child(Parent):
    def display(self):
        super().display()   # call parent method
        print("This is Child class")
```

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

8. Method Overloading and Overriding

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

Ans:

Method Overloading in Python

- Definition: Having multiple methods with the same name but different parameters.
- Note: Python does not support traditional method overloading (like Java/C++).
- Workaround: Achieved using *default arguments* or *args.

Example with default arguments:

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

calc = Calculator()
print(calc.add(2, 3))      # 5
print(calc.add(2, 3, 4))   # 90
```

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

Ans:

Method Overriding in Python

- Definition: Redefining a method in the child class that already exists in the parent class.
- Purpose: To change or extend the functionality of the parent class method.

Example:

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


class Child(Parent):
   def show(self):   # overriding parent method
      print("This is Child class")


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

Output:

```
This is Child class
```

9. SQLite3 and PyMySQL (Database Connectors)

Q) Introduction to SQLite3 and PyMySQL for database connectivity.

Ans:

1)SQLite3

- Definition: Lightweight, serverless database engine built into Python.
- Use: Good for small applications and local storage.

- Example:

```
import sqlite3


conn = sqlite3.connect("test.db")

cur = conn.cursor()

cur.execute("CREATE TABLE IF NOT EXISTS users(id INTEGER, name TEXT)")

conn.commit()

conn.close()
```

2) PyMySQL

- Definition: Python library used to connect to MySQL databases.
- Use: For larger applications needing a client-server database.
- Example:

```
import pymysql


conn = pymysql.connect(host="localhost", user="root", password="1234", database="testdb")

cur = conn.cursor()

cur.execute("CREATE TABLE IF NOT EXISTS users(id INT, name VARCHAR(50))")

conn.commit()

conn.close()
```

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

Ans:

1. Using SQLite3

```
import sqlite3

# Connect or create database
conn = sqlite3.connect("test.db")
cur = conn.cursor()

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

# Insert data
cur.execute("INSERT INTO users VALUES(1, 'Mitesh')")

# Fetch data
cur.execute("SELECT * FROM users")
rows = cur.fetchall()
print(rows)

conn.commit()
conn.close()
```

2. Using PyMySQL

```
import pymysql

# Connect to MySQL
conn = pymysql.connect(host="localhost", user="root", password="1234", database="testdb")
cur = conn.cursor()

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

# Insert data
cur.execute("INSERT INTO users VALUES(1, 'Mitesh')")

# Fetch data
cur.execute("SELECT * FROM users")
rows = cur.fetchall()
print(rows)

conn.commit()
conn.close()
```

10. Search and Match Functions

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

Ans:

re.search() vs re.match() in Python

Both are used for pattern matching with the re module.

1. re.match()

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

```
import re
text = "Hello Python"
if re.match("Hello", text):
    print("Match found")
```

2. re.search()

- Scans the entire string and returns the first occurrence of the pattern.

```
import re
text = "Hello Python"
if re.search("Python", text):
    print("Search found")
```

Q) Difference between search and match.

Ans:

Difference between re.search() and re.match()

- re.match():
  - Looks for a match only at the beginning of the string.
  - Returns None if the pattern is not at the start.
- re.search():
  - Scans the entire string to find the first occurrence of the pattern.
  - Returns None if the pattern is not found anywhere.

Example:

```
import re
text = "Hello Python"

print(re.match("Python", text))   # None (not at start)
print(re.search("Python", text))  # Match object (found inside)
```