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

The print() function in Python is used to display output on the screen (standard output). It's one of the most commonly used functions and is built into Python, so you can use it without importing anything.

Example :

print("hello world!)

print("this is python programming")

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

1. f-strings (Formatted String Literals)

- **Introduced in Python 3.6**, f-strings provide a concise and readable way to embed expressions inside string literals.
- To create an f-string, prefix the string with the letter **f** or **F** and place variables or expressions inside curly braces {}.
- **Advantages:**
    - More readable and concise.
    - Supports expressions directly inside the string.
    - Faster compared to older formatting methods.

Syntax:

f"Some text {variable} more text"

2. format() Method

- The str.format() method was introduced in **Python 2.7 and Python 3.0** as an improvement over the % formatting.
- It uses placeholders {} within the string, which are replaced by arguments passed to .format().
- **Advantages:**
    - Works in both Python 2 and Python 3.
    - Allows positional and named placeholders.
    - Supports formatting numbers, alignment, and padding.

Syntax:

```
"Some text {} more text".format(value)
```

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

input() Function in Python

The input() function in Python is used to take input from the user through the keyboard during program execution. It allows a program to be interactive by accepting data directly from the user.

Syntax:

```
variable_name = input(prompt)
```

 prompt (optional): A string displayed on the screen to tell the user what kind of input is expected.

 variable_name: A variable where the user’s input will be stored.

 The input() function always returns data as a string.

Q 4. Converting user input into different data types (e.g., int, float, etc.).

Converting User Input into Different Data Types in Python

By default, the input() function in Python always returns the entered value as a string (str), regardless of whether the user types numbers, characters, or symbols. If the program requires numeric values for calculations, the string input must be explicitly converted into the appropriate data type using type conversion functions.

Common Type Conversion Functions

1. **int()** – Converts a string into an integer.
   Example:

   ```
   num = int(input("Enter an integer: "))
   ```

This will only work if the user enters a whole number (e.g., 5).

2. **float()** – Converts a string into a floating-point number.
   Example:

price = float(input("Enter the price: "))

3. **str()** – Ensures the value is treated as a string (rarely needed since input() already returns a string).
   Example:

text = str(input("Enter text: "))

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

In Python, files are opened using the open() function, which takes two main arguments:

'r' (Read mode)

- Opens a file for reading only (default mode).
- The file must exist, otherwise an error (FileNotFoundError) will occur.
- The file pointer is placed at the beginning of the file.
  Example:

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

'w' (Write mode)

- Opens a file for writing.
- If the file exists, its contents are **erased**.
- If the file does not exist, a new file is created.
  Example:

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

'a' (Append mode)

- Opens a file for writing but preserves existing content.
- Data is written at the end of the file.

- Creates a new file if it does not exist.
  Example:

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

'r+' (Read and Write mode)

- Opens a file for both reading and writing.
- The file must already exist.
- The pointer starts at the beginning, allowing both reading and overwriting of existing content.
  Example:

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

'w+' (Write and Read mode)

- Opens a file for both writing and reading.
- If the file exists, its content is erased.
- If the file does not exist, a new one is created.
  Example:

f = open("data.txt", "w+")

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

The open() function in Python is used to create and access files.

Its syntax is:

file_object = open(filename, mode)

The filename is the name or path of the file, and the mode specifies the operation:

- 'r' → Read (file must exist)
- 'w' → Write (creates/overwrites file)
- 'a' → Append (creates if not exists)
- 'r+' → Read and write (file must exist)

- 'w+' → Write and read (creates/overwrites file)

Q7. Closing files using close().

Closing Files using close() in Python
After performing read or write operations on a file, it is important to close it to free system resources and ensure all data is saved properly. Python provides the close() method for this purpose.

Syntax:

file_object.close()

Example:

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

f.write("Hello, World!")

f.close()

Key Points:

- Closing a file ensures changes are written to disk.
- Avoids file corruption and memory leaks.
- Using with open(...) as f: automatically closes the file without calling close() explicitly.

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

1. read() Method

- Reads the **entire file** content (or a specified number of characters).
- Returns the data as a **string**.

Example:

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

content = f.read()

print(content)

f.close()
```

2. readline() Method

- Reads **one line** from the file at a time.
- The newline character (\n) is preserved.

Example:

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

line = f.readline()

print(line)

f.close()
```

3. readlines() Method

- Reads **all lines** from the file and returns them as a **list of strings**.
- Each list element represents one line.

Example:

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

lines = f.readlines()

print(lines)

f.close()
```

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

1. write() Method

- Writes a **single string** to the file.
- Overwrites existing content if the file is in 'w' mode.

- Does **not** add a newline automatically; it must be included manually (\n).

Example:

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

f.write("Hello, World!\n")

f.write("This is Python.")

f.close()
```

2. writelines() Method

- Writes a **list of strings** to the file.
- Does not add newlines automatically; each string in the list should contain \n if needed.

Example:

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

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

f.writelines(lines)

f.close()
```

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

An **exception** is an error that occurs during the execution of a program, which disrupts the normal flow of instructions. Examples include dividing by zero, accessing a file that doesn't exist, or using invalid data types.

If exceptions are not handled, the program will stop and display an error message. To prevent this, Python provides **exception handling** using try, except, and finally.

1. try Block

- Contains the code that may cause an exception.

- If an error occurs, control passes to the except block.

2. except Block

- Contains the code to handle the exception.
- Can specify the type of exception (e.g., ZeroDivisionError).

3. finally Block

- Contains code that will **always execute** whether an exception occurs or not.
- Commonly used for resource cleanup (e.g., closing files).

Q11. Understanding multiple exceptions and custom exceptions.

Multiple Exceptions
In Python, a try block can raise different types of errors. We can handle them using multiple except blocks, each for a specific exception type. Example:

Example:

```
try:
  a = int(input())
  b = int(input())
  print(a / b)
except ZeroDivisionError:
  print("Cannot divide by zero.")
except ValueError:
  print("Invalid input.")
```

Custom Exceptions
We can create our own exceptions by defining a class that inherits from Exception.
Example:

Example:

```
class MyError(Exception):
    pass
try:
    age = int(input())
    if age < 18:
        raise MyError("Age must be 18+.")
except MyError as e:
    print(e)
```

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

1. Class

A **class** is a blueprint for creating objects. It groups attributes (data) and methods (functions) into a single unit.

Syntax:

```
class ClassName:
    # attributes
    # methods
```

2. Object

An **object** is an instance of a class. It is created from the class blueprint and can access the class's attributes and methods.

Syntax:

```
object_name = ClassName()
```

3. Attributes
Attributes are variables inside a class that store data for each object.

- **Instance Attributes:** Unique to each object.
- **Class Attributes:** Shared by all objects of the class.

Syntax:

```
self.attribute_name = value      # inside class

object_name.attribute_name      # outside class
```

4. Methods
Methods are functions defined inside a class that describe an object's behavior. The first parameter is self, which refers to the current object.

Syntax:

```
def method_name(self, parameters):

    # method body
```

<u>Q13.</u> Difference between local and global variables.

Local Variable

- A variable declared **inside a function** and accessible **only within that function**.
- Created when the function is called and destroyed when the function ends.

Syntax:

```
def my_func():

    x = 10   # local variable

    print(x)
```

Global Variable

- A variable declared **outside all functions** and accessible from **any part of the program**.
- Can be read inside functions, but to modify it inside a function, the global keyword is required.

Syntax:

```
x = 10  # global variable

def my_func():

    global x

    x = 20
```

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

1. Single Inheritance:
Single inheritance is when a child class inherits from only one parent class. It helps in reusing the code of the base class in the derived class.

Syntax:

```
class Parent:

    # parent class members

    pass

class Child(Parent):

    # child class members

    pass
```

2. Multilevel Inheritance
In multilevel inheritance, a class is derived from another class, which is itself derived from another class. It forms a "chain" of inheritance.

Syntax:

```
class GrandParent:
```

pass

class Parent(GrandParent):

pass

class Child(Parent):

pass

3. Multiple Inheritance

Multiple inheritance allows a class to inherit from more than one base class, combining features from all parent classes.

Syntax:

python

CopyEditclass Parent1:

pass

class Parent2:

pass

class Child(Parent1, Parent2):

pass

---

4. Hierarchical Inheritance

Theory:
In hierarchical inheritance, multiple child classes inherit from a single parent class.

Syntax:

class Parent:

pass

class Child1(Parent):

pass

```
class Child2(Parent):

  pass
```

5. Hybrid Inheritance

Hybrid inheritance is a combination of two or more types of inheritance. It can involve complex hierarchies and may require careful handling of the **Method Resolution Order**

Syntax:

```
class Parent:

  pass

class Child1(Parent):

  pass

class Child2(Parent):

  pass

class GrandChild(Child1, Child2):

  pass
```

<u>Q15</u>. Using the super() function to access properties of the parent class.

- The super() function in Python is used to call a **method** or access **attributes** from a parent (super) class.
- It is commonly used in **inheritance** to:
    1. Invoke the **parent class constructor** (__init__()).
    2. Access **methods or properties** of the parent class without explicitly naming it.
- This helps avoid **repetition** and ensures **method resolution order (MRO)** is respected, which is important in **multiple inheritance**.

Syntax

```
class Parent:
  def __init__(self):
    # Parent constructor
    pass
  def parent_method(self):
    # Parent method
    pass
 class Child(Parent):
  def __init__(self):
    super().__init__()   # Calls Parent's constructor

  def child_method(self):
    super().parent_method()  # Calls Parent's method
```

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

- Method Overloading means defining multiple methods with the same name but different numbers or types of parameters.
- Python does not support traditional method overloading like Java or C++.
- If you define multiple methods with the same name in a class, the latest definition will overwrite the previous one.
- However, we can achieve method overloading-like behavior by:
    - Using default arguments.

- Using *args or **kwargs.

Syntax (Using Default Arguments)

```
class MyClass:
    def display(self, a=None, b=None):
        if a is not None and b is not None:
            print(f"a = {a}, b = {b}")
        elif a is not None:
            print(f"a = {a}")
        else:
            print("No arguments provided")


# Object creation
obj = MyClass()
obj.display()        # No arguments
obj.display(10)      # One argument
obj.display(10, 20)  # Two arguments
```

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

Definition:
Method overriding occurs when a child class defines a method with the same name and parameters as a method in its parent class.

- The child class method replaces (overrides) the parent class method when called on a child class object.
- It is mainly used to customize or extend the behavior of inherited methods.

Syntax:

```
class Parent:
    def method_name(self):
        # Parent class method code
        pass
class Child(Parent):
    def method_name(self):
        # Overriding the parent class method
        pass
```

Q18. Introduction to SQLite3 and PyMySQL for database connectivity.

SQLite3

- **Definition:** A lightweight, serverless database stored in a single file.
- **Use:** Good for small/local applications.
- **Syntax Example:**

```
import sqlite3
conn = sqlite3.connect('test.db')
cur = conn.cursor()
cur.execute("CREATE TABLE users(id INTEGER, name TEXT)")
conn.commit()
conn.close()
```

PyMySQL

- **Definition:** A Python library to connect to a **MySQL** database server.
- **Use:** Suitable for larger, multi-user applications.

- **Syntax Example:**

```
import pymysql

conn = pymysql.connect(host='localhost', user='root', password='1234', db='testdb')

cur = conn.cursor()

cur.execute("CREATE TABLE users(id INT, name VARCHAR(50))")

conn.commit()

conn.close()
```

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

1. Using sqlite3:

```
import sqlite3

# Connect (or create DB)

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

cursor = conn.cursor()


# Create table

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


# Insert data

cursor.execute("INSERT INTO users(name) VALUES (?)", ("Alice",))

conn.commit()


# Fetch data
```

```
cursor.execute("SELECT * FROM users")

print(cursor.fetchall())


conn.close()
```

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

- **re.search(pattern, string)** → Scans the entire string for the first match of the pattern.
- **re.match(pattern, string)** → Checks only at the **start** of the string for a match.

Example:

```
import re

text = "Hello Python"

# re.search() → looks anywhere

result1 = re.search("Python", text)

print(result1)  # Match found

# re.match() → looks only at the start

result2 = re.match("Python", text)

print(result2)  # None, because 'Python' is not at the start
```

Q21. Difference between search and match.

re.match() → Pattern must match **from the beginning** of the string. Stops checking after the first character mismatch.

re.search() → Scans the **entire string** and returns the first location where the pattern matches.

Return value → Both return a **match object** if found, else None.

Example →

```
import re

re.match("cat", "catfish")   # ✅ match

re.search("cat", "dogcat")   # ✅ search
```