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

The print() function is one of the most important and commonly used built-in functions in Python. It is used to display data, messages, or results on the screen. Whenever we want to show any output to the user, we use the print() function.

Definition of print()

print() is a built-in Python function that outputs text, numbers, variables, and expressions to the console.

Example:
print("Hello, Python")

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

InPython,outputformattingisusedtodisplayvaluesinaclean, structured,andreadablemanner. Two modern and widely used methods for formatting outputs are **f-strings** and the **format() method**. Both allow us to insert variables and expressions inside strings easily.

FormattingUsingf-Strings(FormattedStringLiterals)

Definition

f-stringswereintroducedinPython3.6.
Theyallowwritingstringswithvariablesinside{}usingtheprefixf.

Syntax

f"string{variable_name}"

Example 1:

```
name="Hardik"
age = 25
print(f"Mynameis{name}andIam{age}yearsold.")
```

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

In Python, programs often need data from the user. To take input from the keyboard during program execution, Python provides the built-in input() function. It allows the user to enter text, numbers, or other information, which can then be processed by the program.

Definition of input() Function:

The input() function is used to read data entered by the user from the keyboard. It always returns the input as a string (text), even if the user enters a number.

Syntax:

input(prompt_message)
Where:
• prompt_message → (optional) a message shown to the user before taking input.

Example:

input("Enter your name:")

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

InPython,theinput()functionalwaysreturnsuserinputasa**string**(text).However,inmany programs we need numerical values such as integers or floating-point numbers. To perform calculations or comparisons, we must convert the string input into the required data type.

WhyConversionisNeeded?

- input()givesoutputinstringformat
- Arithmeticoperationscannotbedoneonstrings
- Forcalculations,weneednumberslike **int**or**float**
- Datatypeconversionhelpstheprogramunderstandthecorrecttypeof input

CommonDataTypesforConversion:

DataType	Description	ConversionFunction
int	Whole numbers	int()
float	Decimalnumbers	float()
str	Text(defaultinputtype)	Noconversion needed

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

File Handling in Python

File handling is an important part of Python programming. It allows us to create, read, write, and modify files. To work with files, Python provides the built-in function `open()`.
The behavior of file operations depends on the mode used while opening the file.

Syntax of open():
open(filename, mode)

Where:
• filename → name of the file (e.g., "data.txt")
• mode → operation type: read, write, append, etc.

A) Read Mode – r

Definition:
• Opens the file only for reading.

• File must already exist.
• File pointer is at the beginning.

Example:
```
f = open("data.txt", "r")
print(f.read())
```

B) Write Mode – w

Definition:
• Opens the file for writing.
• If the file exists → content is erased (overwritten).
• If the file does not exist → new file is created.

Example:
```
f = open("notes.txt", "w")
f.write("Hello Python")
```

C) Append Mode – a

Definition:
• Opens the file for adding new data at the end.
• Does not erase the existing content.
• Creates the file if it does not exist.

Example:
```
f = open("log.txt", "a")
f.write("New entry added\n")
```

D) Read and Write Mode – r+

Definition:
• Opens the file for both reading and writing.
• File must already exist.
• Pointer starts at the beginning.

Example:
```
f = open("data.txt", "r+")
content = f.read()
f.write("\nNew data added")
```

E) Write and Read Mode – w+

Definition:
• Opens the file for reading and writing.
• Erases existing content.
• Creates a new file if it does not exist.
• Pointer starts at the beginning.

Example:
```
f = open("sample.txt", "w+")
f.write("Python file handling")
f.seek(0)
print(f.read())
```

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

The open() function is used to open a file and return a file object.
This file object allows the program to read from or write to the file depending on the mode provided.

Syntax of open():
```
open(filename, mode)
```

• filename – name of the file (e.g., "data.txt")
• mode – operation mode such as r, w, a, r+, etc.

Creating a File Using open():

A new file can be created using the following modes:

• w → write mode (creates file if it doesn’t exist)
• a → append mode (creates file if it doesn’t exist)
• x → exclusive creation mode (creates file but gives error if it exists)

Example: Creating a File

```
f = open("myfile.txt", "w")
f.write("This is a new file.")
f.close()
```

This will create myfile.txt and write content to it.

Accessing (Opening) an Existing File:

To open and read an already existing file, we use read mode (r).

Example: Reading a File

```
f = open("myfile.txt", "r")
content = f.read()
print(content)
f.close()
```

7.Closing files using close().

The close() Method in Python

The close() method is used to close an open file.
Closing a file releases system resources and ensures that all data is properly written to the file.

Why is Closing a File Important?

Closing a file is necessary because:

• It saves all data written to the file
• It frees system resources
• It prevents file corruption
• It reduces memory usage
• It avoids errors when reopening or modifying the file later

If a file is not closed properly, data might not be written to the disk correctly.

Syntax:
```
file_object.close()
```

Example:
```
f = open("info.txt", "r")
f.close()
```

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

File Handling in Python – Reading Data

File handling in Python allows us to work with text files easily. Reading data from files is an essential operation. Python provides three main methods to read data from a file:

1. read()
2. readline()
3. readlines()

Each method behaves differently and is used based on the requirement.

Opening a File for Reading

To read data, the file must be opened in read mode (r).

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

---

1. Reading Using read()

The read() method reads the entire file content as a single string.

Syntax:
```
file.read(size)
```

• size (optional): number of characters to read
• If size is not given → reads the whole file

Example:
```
f = open("data.txt", "r")
content = f.read()
print(content)
f.close()
```

---

2. Reading Using readline()

The readline() method reads only one line at a time from the file.

Syntax:
```
file.readline()
```

Example:
```
f = open("data.txt", "r")
line1 = f.readline()
line2 = f.readline()
print(line1)
print(line2)
f.close()
```

---

3. Reading Using readlines()

The readlines() method reads all lines and returns them as a list, where each list item is a line.

Syntax:
```
file.readlines()
```

Example:
```
f = open("data.txt", "r")
lines = f.readlines()
print(lines)
f.close()
```

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

write() Method

• Used to write a single string into a file.
• If the file doesn't exist, Python creates it (in w or a mode).
• If opened in write mode (w), it overwrites the file.
• If opened in append mode (a), it adds at the end of the file.

Example:
```
f = open("demo.txt", "w")
f.write("Hello, this is the first line.\n")
f.close()
```

---

writelines() Method

• Used to write a list of strings to a file.
• Does NOT add newline automatically — you must add \n yourself.

Example:
```
lines = [
"Apple\n",
"Banana\n",
"Cherry\n"
]

f = open("fruits.txt", "w")
f.writelines(lines)
f.close()
```

10.Introduction to exceptions and how handle the musingtry,except,and finally.

Exception Handling in Python

An exception is an error that occurs during program execution and stops the program abruptly.

Examples of Common Exceptions

• ZeroDivisionError → dividing by zero
• ValueError → wrong value
• TypeError → wrong data type
• FileNotFoundError → file does not exist
• IndexError → accessing invalid index

Why Handle Exceptions?

Exception handling:

• Prevents the program from crashing
• Allows smooth flow of the program
• Helps detect and fix errors

Exception Handling Using try, except, and finally

try Block:
Write code that may cause an error.

except Block:
Write code that runs if the error occurs.

finally Block:
Always runs — whether an error occurs or not.
Used for cleanup activities (closing files, releasing resources).

Basic Syntax

```
try:
# code that may cause an exception
except:
# code to handle the exception
finally:
# code that always runs
```

11.Understanding multiple exceptions and custom exceptions.

Exceptions in Python

In Python, exceptions occur when an error happens during the execution of a program.
To prevent the program from crashing, we handle exceptions using **try**, **except**, and **finally**.

Sometimes a program can generate different types of errors, and we need to handle each one differently. This is done using **multiple exceptions**.

Python also allows us to create our own exceptions, known as **custom exceptions**, for special situations.

Handling Multiple Exceptions

A single try block can raise different errors.

Example:
• ValueError → invalid input
• ZeroDivisionError → dividing by zero

To handle each one separately, we use different **except** blocks.

Syntax for Multiple Exceptions:

```
try:
    # Code that may raise multiple types of exceptions
except ExceptionType1:
    # Handle first exception
except ExceptionType2:
    # Handle second exception
```

What is a Custom Exception?

A custom exception is a user-defined error class.
We create it when built-in exceptions are not enough.

Example Use-Cases:
• Age must not be negative
• Password must be strong
• Marks cannot exceed 100

Creating a Custom Exception

A custom exception must inherit from the base class **Exception**.

Syntax:

```
class MyException(Exception):
    pass
```

12. Understanding the concepts classes,objects,attributes,andmethods in Python.

Object-Oriented Programming (OOP) in Python

Python is an **object-oriented programming (OOP)** language.
OOP helps in building programs using real-world concepts like **objects, attributes, and behaviors**.

The four basic concepts of OOP are:
• Class
• Object
• Attributes
• Methods

Class

A **class** is a blueprint or template for creating objects. It defines:
• the data (attributes)
• the behavior (methods)

Example:
```
class Student:
pass
```

Object

An **object** is an instance of a class.
When a class is created, no memory is allocated. Memory is allocated only when an object is created.

Creating objects:
```
s1 = Student()
s2 = Student()
```

Attributes

Attributes are variables inside a class. They are used to store data related to the object.
Python has two types of attributes:

1. Instance Attributes

• Unique for each object
• Defined inside the **init()** method

Example:

```
class Student:
    def __init__(self, name, age):
        self.name = name  # instance attribute
        self.age = age    # instance attribute
```

```
# Creating objects
s1 = Student("Dharmesh", 25)
s2 = Student("Raman", 22)
```

2. Class Attributes

• Shared by all objects
• Defined outside the **init()** method

Example:

```
class Car:
    wheels = 4  # class attribute

    def __init__(self, brand):
        self.brand = brand
```

Methods

Methods are functions inside a class. They define the behaviors or actions of the object.

Types of Methods:

1. Instance Methods
2. Class Methods
3. Static Methods

1. Instance Method

Uses **self** and works on object attributes.

Example:

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

    def show(self):  # instance method
        print("Student Name:", self.name)
```

2. Class Method

Uses **cls** and works on class-level data.

Example:

```
class School:
    school_name = "ABC School"

    @classmethod
    def get_school(cls):
        print("School Name:", cls.school_name)
```

3. Static Method

General-use method. Does not use **self** or **cls**.

Example:

```
class Math:
    @staticmethod
    def add(a, b):
        return a + b
```

13.Difference between local and global variables.

Variables in Python – Scope

Variables in Python can be classified based on their **scope**, i.e., where they can be accessed in a program.
The two main types are:
• Local Variables
• Global Variables

Local Variables

A **local variable** is a variable declared inside a function and can be used only within that function.

Key Features:
• Created inside a function
• Accessible only inside that function
• Memory is allocated when the function is called
• Destroyed when the function ends

Global Variables

A **global variable** is declared outside all functions and can be accessed throughout the program.

Key Features:
• Declared outside functions
• Accessible inside and outside functions
• Remains in memory until the program ends

14.Single,Multilevel,Multiple,Hierarchical,andHybridinheritancein Python.

Inheritance in Python

Inheritance allows a class (child) to acquire the properties and methods of another class (parent). Python supports different types of inheritance:

---

1. Single Inheritance

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

Example:

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

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

obj = B()
obj.showA()
obj.showB()
```

---

2. Multilevel Inheritance

A class inherits from another derived class, forming a chain.

Example:

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

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

class C(B):  # Multilevel inheritance
    def showC(self):
        print("Class C")

obj = C()
obj.showA()
obj.showB()
obj.showC()
```

---

3. Multiple Inheritance

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

Example:

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

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

class C(A, B):  # Multiple inheritance
    def showC(self):
        print("Class C")

obj = C()
obj.showA()
obj.showB()
obj.showC()
```

---

4. Hierarchical Inheritance

One parent class is inherited by **multiple child classes**.

Example:

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

class B(A):  # Child 1
    def showB(self):
        print("Class B")

class C(A):  # Child 2
    def showC(self):
        print("Class C")

obj1 = B()
obj2 = C()

obj1.showA()
obj1.showB()
obj2.showA()
obj2.showC()
```

---

5. Hybrid Inheritance

Combination of two or more types of inheritance (e.g., multiple + hierarchical).

Example:

```
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 (multiple + hierarchical)
    def showD(self):
        print("Class D")

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

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

The super() Function in Python

super() is a built-in function used to call **parent class methods or constructors** from a child class.

Purpose

• Avoid rewriting code
• Reuse parent class functionality
• Maintain clean and structured code

Syntax

• super().method_name()
• super().__init__()

Using super() to Call Parent Constructor

Example:

```
class Parent:
    def __init__(self):
        print("Parent constructor called")

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

obj = Child()
```

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

Method Overloading in Python

Method overloading is an **OOP concept** where multiple methods have the **same name** but **different parameters** (number or type of parameters) in the same class.

Purpose

• Allows a class to perform different tasks using methods with the same name
• Improves code readability and reusability

Note: Unlike some languages (like Java or C++), Python **does not support traditional method overloading directly**.
The latest method defined **overrides the previous one**.

We can achieve method overloading in Python using:

• Default parameters
• Variable-length arguments (`*args`)

17.Method overriding:redefining a parentclass method in the child class.

Method Overriding in Python

Method overriding is an **OOP concept** where a child class provides its **own implementation** of a method already defined in the parent class.

Purpose

• Allows a child class to change or extend the behavior of a parent class method
• Supports runtime polymorphism
• Improves flexibility in inheritance

Key Points

• The method in the child class must have the **same name** as in the parent class
• The **number and type of parameters** should be the same
• Python automatically calls the **child class method** when invoked using a child object

---

Syntax

```
class Parent:
    def method_name(self):
        # parent class method
        pass

class Child(Parent):
    def method_name(self):
        # overridden method in child
        pass
```

18.Introduction to SQ Lite3 and PyMySQL for database connectivity.

Database Connectivity in Python

Python provides several libraries to connect and interact with databases. Two commonly used libraries are:

1. **SQLite3** – Lightweight, file-based database
2. **PyMySQL** – Connects Python to MySQL server

These libraries allow Python programs to perform **CRUD operations**:

• **Create** → Insert records
• **Read** → Select records
• **Update** → Modify records
• **Delete** → Remove records

SQLite3

• SQLite3 is a lightweight, serverless, file-based relational database
• No need to install a separate server
• Comes built-in with Python

Connecting to SQLite3 Database:

```
import sqlite3

# Connect to a database (creates file if not exists)
conn = sqlite3.connect("mydatabase.db")

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

PyMySQL

• PyMySQL is a Python library to connect with MySQL Server
• MySQL is a server-based relational database, ideal for web applications
• PyMySQL allows Python programs to execute SQL queries remotely

Installing PyMySQL:

```
pip install pymysql
```

Connecting to MySQL Database:

```
import pymysql

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

cursor = conn.cursor()
```

19.Creating and executing SQLqueries from Python using the seconnectors.

Python MySQL CRUD Operations Using PyMySQL

1. Import and Connect

```
import pymysql

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

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

---

2. Create Table

```
cursor.execute('''
CREATE TABLE IF NOT EXISTS employees (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50),
    salary FLOAT
)
''')
```

---

3. Insert Data

```
cursor.execute("INSERT INTO employees (name, salary) VALUES (%s, %s)", ("Hardik", 50000))
cursor.execute("INSERT INTO employees (name, salary) VALUES (%s, %s)", ("Raman", 45000))
conn.commit()
```

---

4. Retrieve Data

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

for row in rows:
    print(row)
```

---

5. Update Data

```
cursor.execute("UPDATE employees SET salary=%s WHERE name=%s", (55000, "Hardik"))
conn.commit()
```

---

6. Delete Data

```
cursor.execute("DELETE FROM employees WHERE name=%s", ("Raman",))
conn.commit()
```

---

7. Close Connection

```
conn.close()
```

20.Usingre.search() and re.match() functions in Python's remodule for pattern matching.

Regular Expressions in Python

Python's `re` module provides support for **regular expressions**, which allow you to search, match, and manipulate strings based on patterns.

Two commonly used functions are:

1. **re.match()** – checks for a match **only at the beginning** of the string
2. **re.search()** – searches for a pattern **anywhere** in the string

---

Importing the `re` Module

```
import re
```

---

re.match() Function

Syntax:

```
re.match(pattern, string, flags=0)
```

Description:
• Checks if the pattern matches at the **beginning** of the string
• Returns a **match object** if found, otherwise **None**

Example:

```
import re

text = "Python is easy"
pattern = "Python"

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

`re.search()` Function

Syntax:

```
re.search(pattern, string, flags=0)
```

Description:
• Searches for the pattern **anywhere** in the string
• Returns a **match object** if found, otherwise **None**

Example:

```
import re

text = "I love Python programming"
pattern = "Python"

result = re.search(pattern, text)
if result:
    print("Pattern found:", result.group())
else:
    print("Pattern not found")
```

21. Difference between search and match

Feature	re.match()	re.search()
Purpose	Checksforamatchonlyatthebeginning of the string	Searchesforamatchanywhereinthestring
Returns	Amatchobjectifpatternisfoundatthe start; otherwise None	Amatchobjectifpatternisfoundanywhere; otherwise None
UseCase	Whenyouwanttocheckifthestringstarts with a pattern	Whenyouwanttocheckifthepatternexists anywhere in the string
Positionof Match	Must beatindex0 (start)	Canbeatanypositioninthestring
Example	re.match("Python","Pythoniseasy")→ Match re.match("Python","IlovePython")→ None	re.search("Python","IlovePython")→ Match re.search("Python","Pythoniseasy")→ Match