Module 8) Advance Python Programming

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

Ans:

1. Purpose

The print() function sends the data you pass to it as arguments to the standard output (usually the terminal or console window).
It's often used for:

- Displaying messages to users
- Showing variable values during debugging
- Printing results of calculations

---

2. Basic Syntax

print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

Parameters:

1. ***objects** – One or more values to print. Multiple values can be separated by commas.
2. **sep** – Separator between multiple objects (default: a space ' ').
3. **end** – String appended after the output (default: newline '\n').
4. **file** – Output destination (default: sys.stdout for console).
5. **flush** – If True, forces the output to be written immediately (default: False).

---

3. Examples

Printing a simple message

print("Hello, World!")

Output:

Hello, World!

Printing multiple values

```
name = "Dhruvi"
age = 19
print("Name:", name, "Age:", age)
```

Output:

Name: Dhruvi Age: 19

Changing the separator

```
print("Python", "is", "fun", sep="-")
```

Output:

Python-is-fun

Changing the end character

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

Output:

Hello World

Using expressions

```
print("Sum of 5 and 3 is", 5 + 3)
```

Output:

Sum of 5 and 3 is 8

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

Ans:

1. Why Formatting is Needed

When printing output, we often want it to look neat and readable — for example:

- Inserting variables inside strings
- Controlling decimal places
- Adding alignment and padding

Python provides two modern ways for this:

1. **f-strings** (Python 3.6+) – fast and concise
2. **str.format()** method – versatile and works in older versions

---

2. Using f-Strings

Introduced in Python 3.6, **f-strings** allow variables and expressions to be directly embedded in strings using {}.
Prefix the string with f or F.

Example – Simple Variable Insertion

```
name = "Dhruvi"

age = 19

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

Output:

My name is Dhruvi and I am 21 years old.

Formatting Numbers

```
price = 49.567
print(f"Price: {price:.2f}")
```

Output:

```
Price: 49.57
```

(.2f means 2 decimal places)

Expressions Inside f-Strings

```
x = 5
y = 3
print(f"Sum: {x + y}")
```

Output:

```
Sum: 8
```

Alignment and Width

```
text = "Python"
print(f"{text:>10}")   # Right aligned in width 10
print(f"{text:<10}")   # Left aligned in width 10
```

Output:

```
    Python
Python
```

---

3. Using the format() Method

The format() method works by placing {} placeholders in the string, which are replaced by arguments passed to .format().

Basic Example

```
name = "Dhruvi"
age = 19
print("My name is {} and I am {} years old.".format(name, age))
```

Output:

```
My name is Dhruvi and I am 19 years old.
```

Positional and Named Placeholders

```
print("Hello {0}, you are {1} years old.".format("Dhruvi",19))
print("Hello {name}, you are {age} years old.".format(name="Dhruvi", age=19))
```

Number Formatting

```
price = 49.567
print("Price: {:.2f}".format(price))
```

Output:

```
Price: 49.57
```

Alignment and Padding

```
print("{:>10}".format("Python"))  # Right align
print("{:<10}".format("Python"))  # Left align
print("{:^10}".format("Python"))  # Center align
```

---

4. Quick Comparison

Feature	f-Strings	format()
Python version	3.6+	2.7+
Readability	High	Medium
Speed	Faster	Slower
Expression support	Yes	Limited

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

Ans:

In Python, the input() function is used to read data entered by the user from the keyboard.
When input() is called, the program pauses and waits for the user to type something, then it returns that input as a **string**.

Syntax

variable_name = input("Prompt message: ")

- **"Prompt message"** → Optional text displayed to the user.
- **variable_name** → Stores the user's input.

Example

```
# Reading user input
name = input("Enter your name: ")


# Printing the entered value
print("Hello,", name)
```

Output:

Enter your name: Dhruvi

Hello, Dhruvi

Important Points

1. **Always returns a string**
   Even if the user types a number, it will be stored as a string.
   Example:

```
age = input("Enter your age: ")
print(type(age))  # Output: <class 'str'>
```

2. **Convert to other data types** when needed:

```
age = int(input("Enter your age: "))  # Converts to integer
height = float(input("Enter your height: "))  # Converts to float
```

Example with Numbers

```
# Adding two numbers entered by the user
num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))
total = num1 + num2
print("The sum is:", total)
```

Output:

Enter first number: 5

Enter second number: 3

The sum is: 8

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

Ans:

When you use input() in Python, **the data you get is always a string**, even if the user types a number.
To work with it as a number or another type, you must **convert** it using type casting functions like int(), float(), bool(), etc.

---

1. Converting to Integer (int)

Used when you want whole numbers.

age = int(input("Enter your age: ")) # Convert string → int

print("Age after 5 years:", age + 5)

Example Output

Enter your age: 19

Age after 5 years: 24

---

2. Converting to Float (float)

Used when you want decimal numbers.

price = float(input("Enter the price: ")) # Convert string → float

print("Price with tax:", price + (price * 0.18))

Example Output

Enter the price: 100.5

Price with tax: 118.59

---

3. Converting to Boolean (bool)

Any non-empty string becomes True, and an empty string becomes False.

```
value = bool(input("Enter something: "))
print("Boolean value:", value)
```

Example Output

```
Enter something: Hello
Boolean value: True
```

---

4. Converting to List (list)

Used when you want to split input into characters.

```
chars = list(input("Enter a word: "))
print("List of characters:", chars)
```

Example Output

```
Enter a word: Dhruvi
List of characters: ['D', 'h', 'r', 'u', 'v', 'i']
```

---

5. Converting to Other Types

- tuple() → Convert to tuple
- set() → Convert to set (unique values)
- complex() → Convert to complex number

✅ **Tip:** Always validate and handle errors when converting, because if the user types something invalid, Python will raise a ValueError.

```
try:
    num = int(input("Enter a number: "))
    print("Square is:", num ** 2)
```

except ValueError:

print("Please enter a valid number!")

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

Ans:

In Python, you open files using the **open()** function, and the *mode* you choose determines how you can read/write to that file.

Syntax

file = open("filename.txt", "mode")

- "filename.txt" → The file's name (and path if not in the same folder).
- "mode" → Tells Python how to open the file.

File Modes Table

Mode	Meaning	File Must Exist?	What It Does
'r'	**Read** (default)	✅ Yes	Opens file for reading only. Error if file doesn't exist.
'w'	**Write**	❌ No	Creates a new file or overwrites existing file.
'a'	**Append**	❌ No	Creates file if not exists; adds content at the end without erasing old data.
'r+'	**Read + Write**	✅ Yes	Can read and write, but doesn't create new file if missing.

Mode	Meaning	File Must Exist?	What It Does
'w+'	**Write + Read**	❌ No	Creates new file or overwrites, and allows reading.

Examples

1. 'r' → Read mode

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

---

2. 'w' → Write mode

```
file = open("data.txt", "w")
file.write("Hello, World!")
file.close()
```

(Overwrites the file if it exists)

---

3. 'a' → Append mode

```
file = open("data.txt", "a")
file.write("\nThis line is appended.")
file.close()
```

---

4. 'r+' → Read and Write

```
file = open("data.txt", "r+")
data = file.read()
file.write("\nNew content added.")
file.close()
```

---

5. 'w+' → Write and Read

```
file = open("data.txt", "w+")
file.write("New file content.")
file.seek(0)  # Move cursor to start
print(file.read())
file.close()
```

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

Ans:

In Python, the **open()** function is used to **create**, **read**, **write**, or **append** files.
The way the file behaves depends on the **mode** you choose ('r', 'w', 'a', 'x', 'r+', 'w+', etc.).

Syntax

```
file_object = open("filename", "mode")
```

- **"filename"** → Name (and path) of the file.
- **"mode"** → How the file is opened (read, write, append, etc.).

---

1. Creating a File

You can create files using:

- 'w' → Creates new or overwrites existing.
- 'x' → Creates new file, gives an error if file already exists.
- 'a' → Creates new if it doesn’t exist, otherwise appends.

```
# Create or overwrite
file = open("example.txt", "w")
file.write("This is my first file.\n")
file.close()
```

---

2. Reading from a File

```
file = open("example.txt", "r")   # Open in read mode
content = file.read()             # Read entire file
print(content)
file.close()
```

---

3. Appending to a File

```
file = open("example.txt", "a")   # Open in append mode
file.write("This is an extra line.\n")
file.close()
```

---

4. Reading and Writing Together

```
file = open("example.txt", "r+")  # Read + Write
print(file.read())                # Read existing content
file.write("Adding more text.\n") # Write at current cursor
file.close()
```

---

5. Using with for Auto-Close (Best Practice)

```
with open("example.txt", "r") as file:
    print(file.read())  # File closes automatically after this block
```

7.Closing files using close().

Ans:

In Python, the **close()** method is used to **properly close a file** after you've finished working with it.
Closing a file is important because it:

1. **Frees system resources** → The file handle and memory are released.
2. **Saves changes** → Ensures all data you wrote is actually stored on disk.
3. **Prevents file corruption** → Especially important when writing.

Syntax

```
file_object.close()
```

Example: Without with

```
# Open file for writing
file = open("example.txt", "w")
file.write("Hello, world!")
file.close()  # Closing the file

# Open again for reading
```

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

Common Mistake

If you forget to close a file, the data **may not be saved immediately**, and the file remains locked until the program ends.

Better Practice — Use with Statement

The with statement automatically closes the file, even if an error occurs.

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
# File is automatically closed here
```

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

Ans:

****1. read() → Reads Entire File (or Specific Characters)**

```
# Example file content:
# Hello World
# Welcome to Python

file = open("example.txt", "r")
content = file.read()       # Reads the entire file
```

```
print(content)
file.close()
```

Output:

```
Hello World
Welcome to Python
```

- You can also specify **number of characters** to read:

```
file = open("example.txt", "r")
print(file.read(5))  # Reads first 5 characters
file.close()
```

Output:

```
Hello
```

****2. readline() → Reads One Line at a Time**

```
file = open("example.txt", "r")
line1 = file.readline()  # Reads first line
line2 = file.readline()  # Reads second line
print(line1)
print(line2)
file.close()
```

Output:

```
Hello World

Welcome to Python
```

- **Tip:** Each call moves the file cursor to the next line.

- You can also specify a number inside readline() to limit characters per line.

****3. readlines() → Reads All Lines into a List**

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

lines = file.readlines() # Each line becomes a list element

print(lines)

file.close()

Output:

['Hello World\n', 'Welcome to Python\n']

- Notice the \n — it's the newline character from the file.

✅ **When to Use Which**

Method	Use Case
read()	You want the **entire file** as one string.
readline()	You want to **process one line at a time** (e.g., in a loop).
readlines()	You want all lines in a **list** to iterate over later.

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

Ans:

****1. write() → Write a Single String**

- Writes exactly the string you provide.
- Does **not** add a newline automatically — you must include \n yourself if needed.

'w' mode creates a new file or overwrites existing content

```
file = open("example.txt", "w")
file.write("Hello, World!\n")
file.write("Welcome to Python.")
file.close()
```

Result in example.txt:

Hello, World!

Welcome to Python.

****2. writelines() → Write a List of Strings**

- Takes an **iterable** (e.g., list or tuple) of strings and writes them to the file.
- Does **not** automatically add newlines — you must include \n in each string if needed.

```
lines = ["First line\n", "Second line\n", "Third line\n"]

file = open("example.txt", "w")
file.writelines(lines)
file.close()
```

Result in example.txt:

First line

Second line

Third line

---

3. Key Differences

Feature	write()	writelines()
Input	Single string	Iterable (list/tuple) of strings
Newline	Must add manually	Must add manually
Use Case	Writing one block of text	Writing multiple lines at once

---

4. Best Practice with with

```
with open("example.txt", "w") as file:
    file.write("Line 1\n")
    file.writelines(["Line 2\n", "Line 3\n"])
```

(No need to call .close() — closes automatically)

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

Ans:

1. What is an Exception?

An exception is an **event** that disrupts the normal execution of a program.
Example:

```
num = int(input("Enter a number: "))
print(10 / num)  # Will cause error if num = 0
```

If you enter 0, you get:

ZeroDivisionError: division by zero

---

2. Handling Exceptions with try and except

The **try** block contains code that might cause an exception.
The **except** block handles the error so the program can continue.

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

---

3. The finally Block

The **finally** block contains code that will **always execute**, no matter what — even if an error occurs or return is used.
It's often used for cleanup tasks (closing files, releasing resources, etc.).

```
try:
    file = open("example.txt", "r")
    print(file.read())
except FileNotFoundError:
    print("Error: File not found.")
finally:
    print("Closing file (if open).")
    try:
        file.close()
```

```
    except:
        pass
```

---

4. General Syntax

```
try:
    # Code that might raise an exception
except ExceptionType1:
    # Handle specific exception
except ExceptionType2:
    # Handle another specific exception
except:
    # Handle any other exception
finally:
    # Code that always runs
```

---

5. Example: Multiple Exceptions

```
try:
    x = int(input("Enter a number: "))
    y = 10 / x
    print("Result:", y)
except (ValueError, ZeroDivisionError) as e:
    print("Error:", e)
finally:
    print("Execution completed.")
```

11.Understanding multiple exceptions and custom exceptions.

Ans:

1. Handling Multiple Exceptions

In Python, you can handle more than one type of exception in different ways:

(a) Multiple except blocks

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

- Each except block handles a specific type of error.

(b) Multiple exceptions in one block

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
    print("Result:", result)
except (ValueError, ZeroDivisionError) as e:
    print("Error occurred:", e)
```

- (ValueError, ZeroDivisionError) → tuple of exceptions.

- as e → stores the actual error message in variable e.

(c) Catching all exceptions (not always recommended)

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except Exception as e:
    print("Something went wrong:", e)
```

- Useful for debugging, but in production, it's better to catch specific errors.

---

2. Creating Custom Exceptions

Sometimes built-in exceptions are not enough, so you can create **your own**.

Steps:

1. Create a class that inherits from Exception (or a subclass).
2. Raise it using raise.
3. Handle it with except.

Example: Custom Exception

```
# Step 1: Define custom exception
class AgeTooSmallError(Exception):
    pass
```

```
# Step 2: Use it in code
try:
    age = int(input("Enter your age: "))
    if age < 18:
        raise AgeTooSmallError("You must be at least 18 years old.")
    print("Access granted.")
except AgeTooSmallError as e:
    print("Custom Exception:", e)
except ValueError:
    print("Error: Please enter a valid number.")
```

Output Example:

Enter your age: 16

Custom Exception: You must be at least 18 years old.

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

Ans:

1. Class

A **class** is like a **blueprint** or **template** for creating objects.
It defines:

- **Attributes** → Variables that store data (properties of the object)
- **Methods** → Functions inside the class that define behavior

Example:

```
class Car:
    brand = "Toyota"   # Attribute
```

```
    def start(self):   # Method
        print("Car started")
```

---

2. Object

An **object** is a **real instance** of a class — created using the class blueprint.
When you create an object, Python allocates memory for it and gives it the class's properties and behaviors.

Example:

```
my_car = Car()   # Creating an object
my_car.start()   # Calling a method
print(my_car.brand)  # Accessing an attribute
```

Output:

```
Car started
Toyota
```

---

3. Attributes

Attributes are **variables** that belong to a class or object.

Two types:

- **Class attributes** → Shared by all objects of the class.
- **Instance attributes** → Unique to each object, usually defined in __init__.

Example:

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

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

car1 = Car("Toyota", "Red")
car2 = Car("BMW", "Black")

print(car1.brand, car1.color)  # Toyota Red
print(car2.brand, car2.color)  # BMW Black
print(Car.wheels)              # 4
```

---

4. Methods

Methods are **functions inside a class** that define what the object can do.
They must have self as the first parameter to access object data.

Example:

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

    def start(self):
        print(self.brand, "started!")
```

```
my_car = Car("Tesla")
my_car.start()
```

Output:

```
Tesla started!
```

13.Difference between local and global variables.

Ans:

1. Local Variables

- Declared **inside a function** (or block).
- Exist **only while the function is running**.
- Can't be accessed outside the function.

Example:

```
def my_function():
    x = 10  # Local variable
    print("Inside function:", x)

my_function()
# print(x)  # ❌ Error: x is not defined outside the function
```

Output:

```
Inside function: 10
```

---

2. Global Variables

- Declared **outside any function**.
- Can be accessed from **anywhere** in the program.

- If you want to modify it inside a function, you must use the global keyword.

Example:

```
y = 20  # Global variable


def my_function():
    global y
    y = y + 5
    print("Inside function:", y)


my_function()
print("Outside function:", y)
```

Output:

Inside function: 25

Outside function: 25

---

3. Key Differences Table

Feature	Local Variable	Global Variable
Scope	Inside the function only	Entire program
Declared	Inside a function	Outside all functions
Lifetime	Created when function starts, destroyed when it ends	Exists until program ends

Feature	Local Variable	Global Variable
Access outside function	❌ No	✅ Yes
Change inside function	Directly	Need global keyword

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

Ans:

1. Single Inheritance

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

```
class Parent:
    def display_parent(self):
        print("This is the Parent class.")


class Child(Parent):
    def display_child(self):
        print("This is the Child class.")


obj = Child()
obj.display_parent()
obj.display_child()
```

Output:

```
This is the Parent class.
This is the Child class.
```

---

2. Multilevel Inheritance

A class inherits from another class, which itself inherits from another class (**grandparent → parent → child**).

```
class Grandparent:
    def display_grandparent(self):
        print("Grandparent class")


class Parent(Grandparent):
    def display_parent(self):
        print("Parent class")


class Child(Parent):
    def display_child(self):
        print("Child class")


obj = Child()
obj.display_grandparent()
obj.display_parent()
obj.display_child()
```

Output:

```
Grandparent class
Parent class
Child class
```

---

3. Multiple Inheritance

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

```
class Father:
    def skill_father(self):
        print("Father's skill: Driving")

class Mother:
    def skill_mother(self):
        print("Mother's skill: Cooking")

class Child(Father, Mother):
    def skill_child(self):
        print("Child's skill: Painting")

obj = Child()
obj.skill_father()
obj.skill_mother()
obj.skill_child()
```

Output:

```
Father's skill: Driving
Mother's skill: Cooking
Child's skill: Painting
```

---

4. Hierarchical Inheritance

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

```
class Parent:
    def display_parent(self):
        print("Parent class")

class Child1(Parent):
    def display_child1(self):
        print("Child 1 class")

class Child2(Parent):
    def display_child2(self):
        print("Child 2 class")

obj1 = Child1()
obj2 = Child2()
obj1.display_parent()
obj1.display_child1()
obj2.display_parent()
obj2.display_child2()
```

Output:

```
Parent class
Child 1 class
Parent class
```

Child 2 class

---

5. Hybrid Inheritance

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

```
class Grandparent:
    def display_grandparent(self):
        print("Grandparent class")

class Parent1(Grandparent):
    def display_parent1(self):
        print("Parent 1 class")

class Parent2:
    def display_parent2(self):
        print("Parent 2 class")

class Child(Parent1, Parent2):
    def display_child(self):
        print("Child class")

obj = Child()
obj.display_grandparent()
obj.display_parent1()
obj.display_parent2()
```

obj.display_child()

Output:

Grandparent class

Parent 1 class

Parent 2 class

Child class

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

Ans:

The **super()** function in Python is used inside a child class to **access methods and properties from its parent class** without explicitly naming the parent.

It's especially useful in **inheritance** because it allows you to call the parent class constructor (__init__) or other methods, avoiding code duplication.

---

1. Why use super()?

- Calls **parent class methods** without hardcoding the parent class name.
- Useful in **multiple inheritance** (ensures proper method resolution order — MRO).
- Makes code easier to maintain if the parent class name changes.

---

2. Basic Syntax

super().method_name(arguments)

---

3. Example — Calling Parent Class Method

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

class Child(Parent):
    def show(self):
        print("This is the Child class method.")
        super().show()  # Call parent method

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

Output:

```
This is the Child class method.
This is the Parent class method.
```

---

4. Example — Calling Parent Constructor (__init__)

```
class Person:
    def __init__(self, name):
        self.name = name
        print("Person constructor called")

class Student(Person):
```

```
    def __init__(self, name, student_id):
        super().__init__(name)  # Call parent constructor
        self.student_id = student_id
        print("Student constructor called")


obj = Student("Dhruvi", 101)
print(obj.name, obj.student_id)
```

Output:

```
Person constructor called
Student constructor called
Dhruvi 101
```

---

5. Example — Multiple Inheritance

super() follows **Method Resolution Order (MRO)** to decide which parent's method to call first.

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


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

```
class C(B):
    def show(self):
        print("Class C")
        super().show()


obj = C()
obj.show()
```

Output:

Class C

Class B

Class A

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

Ans:

method overloading (like in Java or C++) is **not directly supported** because Python does **not** allow defining multiple methods with the same name in the same class — the last definition overwrites the previous one.

However, Python achieves similar behavior by:

1. Using **default arguments**.
2. Using **variable-length arguments** (*args and **kwargs).
3. Checking **argument types and counts** inside the method.

---

1. What is Method Overloading?

In programming, **method overloading** means defining **multiple methods with the same name** but **different parameter lists** so they can perform similar tasks with different inputs.

---

2. Example — Default Arguments

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


obj = Math()
print(obj.add(5, 10))      # Adds 2 numbers
print(obj.add(5, 10, 15))  # Adds 3 numbers
print(obj.add(7))          # Adds 1 number
```

Output:

```
15
30
7
```

Here, one method works for multiple parameter counts.

---

3. Example — Using *args

```
class Math:
    def add(self, *args):
        return sum(args)
```

```
obj = Math()
print(obj.add(2, 3))         # Adds 2 numbers
print(obj.add(1, 2, 3, 4, 5)) # Adds 5 numbers
```

Output:

```
5
15
```

*args allows passing any number of arguments.

---

4. Example — Type Checking for Overloading Behavior

```
class Display:
    def show(self, a=None, b=None):
        if a is not None and b is not None:
            print(f"Two parameters: {a}, {b}")
        elif a is not None:
            print(f"One parameter: {a}")
        else:
            print("No parameters")

obj = Display()
obj.show(10, 20)
obj.show(5)
obj.show()
```

Output:

```
Two parameters: 10, 20
```

One parameter: 5

No parameters

<u>17.Method overriding: redefining a parent class method in the child class.</u>

<u>Ans:</u>

Method Overriding in Python happens when a **child class** defines a method **with the same name** as a method in its **parent class**, but provides a **new implementation**.
When the child class object calls that method, **the child's version is executed instead of the parent's**.

---

1. Why Use Method Overriding?

- To change or extend the behavior of an inherited method.
- To provide specific functionality for the child class.
- To follow **polymorphism** in object-oriented programming.

---

2. Basic Example

```
class Parent:
    def show(self):
        print("This is the parent class method.")


class Child(Parent):
    def show(self):
        print("This is the child class method.")
```

```
obj = Child()
obj.show()  # Calls the overridden method
```

Output:

This is the child class method.

Here, Child overrides the show() method from Parent.

---

3. Accessing the Parent Method using super()

Sometimes you want to override a method **but still use the parent's version** within the new implementation.

```
class Parent:
    def show(self):
        print("This is the parent class method.")

class Child(Parent):
    def show(self):
        super().show()  # Call parent class method
        print("This is the child class method.")

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

Output:

This is the parent class method.

This is the child class method.

---

4. Example with Polymorphism

```
class Animal:
    def sound(self):
        print("Some generic animal sound.")

class Dog(Animal):
    def sound(self):
        print("Woof! Woof!")

class Cat(Animal):
    def sound(self):
        print("Meow!")

animals = [Dog(), Cat()]

for animal in animals:
    animal.sound()  # Calls respective overridden method
```

Output:

Woof! Woof!

Meow!

18.Introduction to SQLite3 and PyMySQL for database connectivity.

Ans:

1. SQLite3 – Lightweight, File-Based Database

- **What it is:**
  SQLite3 is a built-in Python module for working with **SQLite**, a self-contained, serverless database that stores data in a **single file**.
  No separate server installation is required.
- **When to use:**
  For small to medium applications, prototypes, or desktop apps.

Example: Connecting & Performing Basic Operations

```
import sqlite3


# Connect to (or create) a database file
conn = sqlite3.connect("my_database.db")


# Create a cursor object to execute SQL commands
cursor = conn.cursor()


# Create a table
cursor.execute("""
CREATE TABLE IF NOT EXISTS students (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT,
    age INTEGER
)
""")
```

```
# Insert data
cursor.execute("INSERT INTO students (name, age) VALUES (?, ?)", ("Alice", 20))

# Fetch data
cursor.execute("SELECT * FROM students")
rows = cursor.fetchall()
for row in rows:
    print(row)

# Commit changes & close connection
conn.commit()
conn.close()
```

---

2. PyMySQL – MySQL Database Connectivity

- **What it is:**
  PyMySQL is a Python library for connecting to **MySQL/MariaDB** databases.
  It **requires MySQL Server** to be installed and running.
- **When to use:**
  For large-scale, multi-user applications needing a **full database server**.

Example: Connecting & Performing Basic Operations

```
import pymysql
```

```
# Connect to MySQL database
conn = pymysql.connect(
    host="localhost",
    user="root",
    password="your_password",
    database="test_db"
)

cursor = conn.cursor()

# Create a table
cursor.execute("""
CREATE TABLE IF NOT EXISTS students (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(50),
    age INT
)
""")

# Insert data
cursor.execute("INSERT INTO students (name, age) VALUES (%s, %s)",
("Bob", 22))

# Fetch data
```

```
cursor.execute("SELECT * FROM students")
rows = cursor.fetchall()
for row in rows:
    print(row)

# Commit changes & close connection
conn.commit()
conn.close()
```

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

Ans:

1. Creating & Executing SQL Queries in SQLite3

SQLite3 comes built into Python, so no installation is needed.

```
import sqlite3

# 1. Connect to the database (creates file if not exists)
conn = sqlite3.connect("school.db")
cursor = conn.cursor()

# 2. CREATE TABLE
create_table_query = """
CREATE TABLE IF NOT EXISTS students (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
```

```
    age INTEGER
)
"""
cursor.execute(create_table_query)

# 3. INSERT DATA
insert_query = "INSERT INTO students (name, age) VALUES (?, ?)"
cursor.execute(insert_query, ("Alice", 20))

# 4. SELECT DATA
select_query = "SELECT * FROM students"
cursor.execute(select_query)
rows = cursor.fetchall()
for row in rows:
    print(row)

# 5. UPDATE DATA
update_query = "UPDATE students SET age = ? WHERE name = ?"
cursor.execute(update_query, (21, "Alice"))

# 6. DELETE DATA
delete_query = "DELETE FROM students WHERE name = ?"
cursor.execute(delete_query, ("Alice",))
```

```
# 7. Commit and close
conn.commit()
conn.close()
```

Key Notes:

- ? is used as a placeholder to prevent **SQL injection**.
- .execute() is for single queries, .executemany() for multiple inserts.

---

2. Creating & Executing SQL Queries in PyMySQL (MySQL)

Requires:

```
pip install pymysql
import pymysql

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

# 2. CREATE TABLE
create_table_query = """
```

```
CREATE TABLE IF NOT EXISTS students (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    age INT
)
"""
cursor.execute(create_table_query)

# 3. INSERT DATA
insert_query = "INSERT INTO students (name, age) VALUES (%s, %s)"
cursor.execute(insert_query, ("Bob", 22))

# 4. SELECT DATA
select_query = "SELECT * FROM students"
cursor.execute(select_query)
rows = cursor.fetchall()
for row in rows:
    print(row)

# 5. UPDATE DATA
update_query = "UPDATE students SET age = %s WHERE name = %s"
cursor.execute(update_query, (23, "Bob"))

# 6. DELETE DATA
```

```
delete_query = "DELETE FROM students WHERE name = %s"
cursor.execute(delete_query, ("Bob",))

# 7. Commit and close
conn.commit()
conn.close()
```

Key Notes:

- In **PyMySQL**, placeholders are written as %s instead of ?.
- MySQL requires a **running MySQL server** and an existing database.

---

3. Common SQL Query Types You Can Execute

SQL Command	Purpose	Example
CREATE TABLE	Create a new table	CREATE TABLE students (...)
INSERT INTO	Add records	INSERT INTO students VALUES (...)
SELECT	Retrieve data	SELECT * FROM students
UPDATE	Modify existing data	UPDATE students SET age=21 WHERE name='Alice'
DELETE	Remove data	DELETE FROM students WHERE id=1
DROP TABLE	Remove a table completely	DROP TABLE students

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

Ans:

1. Introduction to the re Module

The **re** module in Python is used for **pattern matching** using **regular expressions (regex)**.
It allows you to search, match, and manipulate strings based on patterns.

To use it:

```
import re
```

---

2. re.match()

- **Purpose:** Checks **only at the beginning** of the string.
- If the pattern is found at the **start**, it returns a match object.
- If the start doesn't match, it returns None.

Example:

```
import re


text = "Hello Python World"


# Match from the start
result = re.match(r"Hello", text)
if result:
   print("Match found:", result.group())
else:
```

```
    print("No match")

# This will fail because 'Python' is not at the start
result = re.match(r"Python", text)
print("Result:", result)
```

Output:

```
Match found: Hello
Result: None
```

---

3. re.search()

- **Purpose:** Searches **anywhere in the string** for the first match.
- If found anywhere, returns a match object.
- If not found, returns None.

Example:

```
import re

text = "Hello Python World"

# Search anywhere
result = re.search(r"Python", text)
if result:
    print("Search found:", result.group())
else:
    print("No match")
```

Output:

Search found: Python

---

4. Difference Between re.match() and re.search()

Feature	re.match()	re.search()
Checks	Only at **start** of string	Anywhere in the string
Speed	Faster for start-only matches	Slightly slower (checks all)
Use case	Fixed format strings	Flexible search anywhere

---

5. Common Methods with Match Objects

When you get a **match object**, you can:

```
m.group()   # The matched string
m.start()   # Start index of match
m.end()     # End index of match
m.span()    # Tuple (start, end)
```

Example:

```
import re
text = "My phone number is 9876543210"
m = re.search(r"\d+", text)  # Find first sequence of digits
print("Matched:", m.group())
print("Position:", m.span())
```

Output:

Matched: 9876543210

Position: (19, 29)

21.Difference between search and match.

Ans:

Feature	re.match()	re.search()
Where it looks	Only at the **beginning** of the string	Anywhere in the string
When it returns a match	If the pattern matches **from the first character**	If the pattern is found **anywhere**
Use case	When you need to check if a string **starts** with a pattern	When you need to **find** a pattern anywhere in the text
Example	re.match(r"Hello", "Hello World") ✅ re.match(r"World", "Hello World") ❌	re.search(r"World", "Hello World") ✅

Quick Example:

```
import re

text = "Hello Python"

# match: checks only at start
print(re.match(r"Hello", text))  # ✅ Found
```

print(re.match(r"Python", text)) # ❌ None

search: checks anywhere

print(re.search(r"Python", text)) # ✅ Found

Output:

<re.Match object; span=(0, 5), match='Hello'>

None

<re.Match object; span=(6, 12), match='Python'>