★ Module 15) Advance Python Programming

➢ 1. Printing on Screen

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

Ans :

The `print()` function in Python is used to display messages or outputs on the screen or another standard output device—it is one of the most commonly used functions for debugging and program output in Python.

➢ Purpose and Basic Usage

The primary purpose of `print()` is to output information, such as strings, numbers, collections (lists, tuples), or objects. It automatically converts objects to strings before displaying them, making it simple to print almost any data type.

Examples:

`python`

```
print("Hello, World!")   # prints a string
a = [1, 2, 3]
print(a)                 # prints a list
```

➢ Syntax and Parameters

The syntax for the `print()` function is:

```
print(object(s), sep=' ', end='\n', file=sys.stdout, flush=False)
```

- `object(s)`: The data or objects to be printed (can pass multiple, separated by commas)
- `sep`: Separator between objects, default is a single space (`' '`)
- `end`: Character printed at the end, default is newline (`'\n'`)
- `file`: Output stream, default is the console (`sys.stdout`)
- `flush`: Boolean, flush output or not, default is `False`

Example with parameters:

```
python

print("Hello", "World", sep=", ", end="!")  # Output: Hello, World!
```

➢ **Features and Use Cases**

- Useful for outputting results, debugging, and displaying variable values.
- Handles multiple data types, converting each to a string before printing.
- Can customize output using `sep` (separator), `end` (ending character), and direct output to files.
- Is always available as a built-in function and returns `None` after execution.

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

Ans: Python provides two popular methods for formatting output: f-strings and the format() method. Both enable clear, flexible, and readable ways to embed variables and expressions directly within strings.

➢ **F-Strings**

F-strings, introduced in Python 3.6, are now the preferred way to format strings. Prefix the string with `f` or `F` and include variables or expressions inside curly braces **`{}`.**

Basic Example:

```
python

name = "Anu"

age = 21

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

Output:

My name is Anu and I am 21 years old.

➢ **Formatting Numbers:**

```
python

    price = 56.7896

    print(f"Price: ₹{price:.2f}")
```

Output:

Price: ₹56.79

➢ **Advanced Use:**

Expressions and methods can be used inside the braces:

```
python

    score = 88

    print(f"Your grade is {'A' if score > 85 else 'B'}")
```

➢ **format() Method**

The `format()` method is available on string objects and lets you insert values using curly braces `{}` as placeholders.

Basic Example:

```
python

  print("My name is {} and I am {} years old.".format("Anu", 21))
```

Output:

My name is Anu and I am 21 years old.

- **Argument Order and Named Arguments:**

```
python

    print("My name is {name} and I am {age} years
    old.".format(name="Anu", age=21))
```

- **Number Formatting:**

  ```
  python

      price = 56.7896

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

  Output:

  Price: ₹56.79

➢ **Key Differences**

- F-strings are concise and allow embedding of direct expressions, making code more readable and efficient.
- format() allows for flexible placeholder replacement but is more verbose and less performant than f-strings for most purposes.

➢ 2. Reading Data from Keyboard

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

Ans:

The input() function in Python is used to read user input from the keyboard and is essential for interactive programs.

- **Basic Usage**

  The syntax for the input() function is:

  ```
  python

      input(prompt)
  ```

- The optional `prompt` argument displays a message to the user before waiting for input.

  Example:

```
python

    name = input("Enter your name: ")

    print("Hello,", name)
```

- This program displays the prompt, waits for the user to type their name, and then prints a message saying hello to the entered name.

➢ **Reading Different Data Types**

- The input() function always returns a string. If you want numeric input, you must convert the result:

```
python

    number = int(input("Enter a number: "))

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

- This way, the typed value is converted from string to integer or float, allowing for calculations.

➢ **Taking Multiple Values**

- You can read multiple values by splitting the input string:

```
python

    x, y = input("Enter two numbers separated by space: ").split()

    print("First number:", x)

    print("Second number:", y)
```

This splits the typed input into separate variables based on spaces.

➢ **Lists and Other Data Structures**

If you want to read a list of elements:

python

```
list1 = input("Enter elements separated by space: ").split()
print(list1)
```

- This will store the entered values as a list of strings.

➢ **Key Points**

- input() is used for interactive programs and simple demos.
- All input values are returned as strings and must be converted when needed.
- Can read and process more complex types with additional string manipulation and conversion functions.

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

Ans: In Python, user input from the input() function is always received as a string. To use the input value in calculations or other operations, it must be explicitly converted to the desired data type using built-in conversion functions such as int(), float(), or bool().

➢ **Converting to Integer**

- To convert a string input to an integer:

python

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

- This allows the value to be used for arithmetic or other integer-specific operations.

➢ **Converting to Float**

- To convert input to a floating-point number:

```
python

price = float(input("Enter a price: "))
```

- This is useful when working with decimal numbers, such as currency.

➢ **Converting to Boolean**

- Booleans can be converted like this:

```
python

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

- This requires inputting 0 (False) or 1 (True).

➢ **Converting to String**

- While input() returns a string by default, explicit conversion can be done with str():

```
python

text = str(input("Type anything: "))
```

- This type of step is usually unnecessary because input() always provides a string.

➢ **Handling Multiple Inputs**

- For multiple values, use split() and map():

```
python

x, y = map(int, input("Enter two numbers separated by space:
").split())
```

- This converts both inputs to integers in one step.

❖ Summary Table

Input	Conversion Examples
Integer	`int(input("Enter int: "))`
Float	`float(input("Enter float: "))`
Boolean	`bool(int(input("Enter 1 (True) or 0 (False): ")))`
String	`str(input("Enter string: "))`
Multiple	`a, b = map(int, input().split())`

➢ 3. Opening and Closing Files

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

Ans:

In Python, the open() function is used to open a file with a specified mode that determines how the file will be accessed and manipulated. Here are the common file modes and their descriptions:

Mode	Description
'r'	Read-only mode. Opens the file for reading. Raises an error if the file does not exist.
'w'	Write-only mode. Opens the file for writing. Overwrites the file if it exists or creates a new file if it doesn't.
'a'	Append-only mode. Opens the file for writing, but adds new data to the end of the file. Creates the file if missing.
'r+'	Read and write mode. Opens the file for reading and writing. File must exist.
'w+'	Read and write mode. Opens the file for reading and writing. Overwrites the file or creates a new one.

➢ **Explanation of Each Mode**

- 'r' (read): Use when you just want to read data from an existing file. The file pointer is placed at the beginning of the file. If the file doesn't exist, Python raises an error.
- 'w' (write): Opens the file for writing. If the file already exists, its contents are erased. If not, a new file is created. The pointer is at the start, so writing begins from the beginning of the file.
- 'a' (append): Opens the file for writing but preserves existing data by placing the write pointer at the end of the file. If the file doesn't exist, it is created.
- 'r+' (read and write): Allows both reading and writing. The file pointer starts at the beginning. This mode requires the file to pre-exist. You can read from the file and write over existing data or append depending on pointer location. You can move the pointer using `seek()`.
- 'w+' (write and read): Like 'w', it overwrites or creates the file but also allows reading. The file pointer is at the beginning.

Example Usage:

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

```
# Write mode
    with open('file.txt', 'w') as f:
        f.write("Hello world!")
```

 # Append mode

```
with open('file.txt', 'a') as f:
    f.write("\nAppending this line.")
```

 # Read and write mode

```
with open('file.txt', 'r+') as f:
    data = f.read()
    f.write("\nAdding more data")
```

 # Write and read mode

```
with open('file.txt', 'w+') as f:
    f.write("Starting fresh")
    f.seek(0)
    print(f.read())
```

---

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

Ans : The open() function in Python is used to create or access files for reading, writing, or appending. It returns a file object that allows interaction with the file's contents.

- **Syntax**

```
open(file_name, mode)
```

- `file_name`: Name or path of the file.
- `mode`: Specifies how the file is opened (e.g., 'r', 'w', 'a', 'x', etc.).

➢ **Creating and Accessing Files with Modes**

- **Read mod**e ('r'): Opens an existing file for reading. Raises an error if the file does not exist.

```
python

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

content = f.read()

f.close()
```

- **Write mode** ('w'): Creates a new file or truncates an existing file and opens it for writing.

```
python

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

f.write("This is a new file.")

f.close()
```

- **Append mode** ('a'): Opens a file for writing but appends new data to the end. Creates the file if it doesn't exist.

```
python

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

f.write("\nAppending some text.")

f.close()
```

- Exclusive Creation mode ('x'): Creates a new file but raises an error if the file already exists.

```
python

f = open("newfile.txt", "x")
```

- Using context manager with open() is recommended because it automatically closes the file after the block executes:

```
python
```

```
with open("example.txt", "r") as f:
    content = f.read()
```

➢ **Key Points**

- Without specifying mode, 'r' (read-text) mode is the default.
- For binary files like images, add 'b' to the mode, e.g., 'rb', 'wb'.
- Always close files using close() or preferably use with to avoid resource leaks.
- **Note** :This function provides flexibility to create, read, write, and append files in Python programs efficiently and safely.

3.Closing files using close().

Ans: The close() method in Python is used to close an open file, releasing any system resources associated with it. It is important to close files after completing operations to ensure data is properly saved and to avoid reaching the limit of open files allowed by the operating system.

- ## Key Points About close()

- Once a file is closed, you cannot perform read or write operations on it. Trying to do so raises a ValueError.
- Closing a file flushes any unwritten information from the buffer to the file system, ensuring data integrity.
- The close() method does not take any parameters and does not return a value.
- The close() method can be called multiple times without raising an error.
- Using **`with open()`** context manager is recommended as it automatically closes the file even if an exception occurs.

➢ Basic Usage Example

python

```
f = open("example.txt", "w")
f.write("Hello, world!")
f.close()  # Close the file to save changes and free resources
```

➢ Error Example After Closing

python

```
f = open("example.txt", "w")
f.write("Hello")
f.close()
f.write("More text")  # This will raise ValueError as file is closed
```

➢ Recommended Pattern With Context Manager

python

```
with open("example.txt", "w") as f:
    f.write("Hello, world!")
 # File is automatically closed here
```

➢4. Reading and Writing Files

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

Ans :

In Python, you can read data from files using three important methods of file objects: read(), readline(), and readlines(). Each serves a specific purpose for reading file contents.

➢ read() Method

- Reads the entire file content as a single string.
- Useful when you want the whole content at once.
- Example:

```
python

with open("example.txt", "r") as f:
    content = f.read()
    print(content)
```

➢ readline() Method

- Reads one line from the file at a time.
- Returns the line including the trailing newline character (`\n`).
- You can call it repeatedly to read subsequent lines.
- Optionally, specify a number of bytes to read from the line.
- Example:

```
python

with open("example.txt", "r") as f:
    line1 = f.readline()
    line2 = f.readline()
```

```
print(line1)

print(line2)
```

- Useful for processing large files line-by-line without loading the entire file into memory.

➢ readlines() Method

- Reads all lines at once and returns them as a list of strings.
- Each list item is a line from the file, including newline characters.
- You can also specify a hint parameter to limit the number of bytes read.
- Example:

```
python

    with open("example.txt", "r") as f:

        lines = f.readlines()

        print(lines)
```

- Handy when you want to iterate over lines easily with Python list operations.

❖ Summary

Method	**Description**	**Returns**	**Use Case**
read()	Reads full file content	Single string	Smaller files, entire content
readline()	Reads next single line	Single string (line)	Large files, line by line reading
readlines()	Reads all lines as list of strin	List of strings (lines)	File as list to iterate conveniently

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

Ans:

In Python, writing data to a file is commonly done using the write() and writelines() methods of a file object, usually after opening the file in write ('w'), append ('a'), or read-write ('r+') mode.

➢ write() Method :

- Writes a single string to the file.
- The position where the text is inserted depends on the file mode and current file pointer position.
- In 'w' or 'w+' modes, existing content is erased and writing starts at the beginning.
- In 'a' or 'a+' modes, writing is appended at the end.
- Does not add a newline character automatically; you must include '\n' for new lines.
- Returns the number of characters written.

Example:

```
python

    with open("myfile.txt", "w") as f:
        f.write("Hello, world!\n")
        f.write("This is on a new line.\n")
```

➢ writelines() Method :

- Writes a list (or any iterable) of strings to the file.
- Does not automatically add newlines; each string should end with '\n' if needed.
- Does not return any value.

Example:

```
python
```

```
lines = ["First line\n", "Second line\n", "Third line\n"]
with open("myfile.txt", "w") as f:
    f.writelines(lines)
```

➢ Key Points

- Both methods require a file opened in a mode that supports writing.
- For appending, open the file with mode 'a' or 'a+'.
- Use write() for individual strings and writelines() for multiple lines.
- Always close the file after writing, or use a `with` statement for automatic closing.

➢5. Exception Handling

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

Ans : An exception in Python is an unexpected event or runtime error that occurs during program execution, which can disrupt the normal flow of the program. For example, dividing a number by zero results in a ZeroDivisionError exception.

Python provides a way to handle exceptions gracefully using the **`try`**, **`except`**, **`else`**, and **`finally`** blocks, allowing your program to catch errors and either respond to them or clean up resources.

➢ The `try` and `except` Blocks

- Code that might cause an exception is placed inside the **`try`** block.
- If an exception occurs in the **`try`** block, the rest of the code there is skipped.
- The matching **`except`** block then executes to handle the exception.
- If no exception occurs, the **`except`** block is skipped.
- Multiple **`except`** blocks can catch different types of exceptions.

Example:

```
python

def divide(x, y):
    try:
        result = x // y  # Integer division
        print("Result:", result)
    except ZeroDivisionError:
        print("Error: Cannot divide by zero.")


# Usage
divide(10, 2)  # Prints: Result: 5
divide(10, 0)  # Prints: Error: Cannot divide by zero.
```

➢ The `else` Block

- The optional **`else`** block runs if no exceptions occur in the **`try`** block.
- It is useful to run code that should only execute when everything in **`try`** succeeds.

➢ The `finally` Block

- The **`finally`** block always executes, regardless of whether an exception occurred or not.
- It is typically used for cleanup tasks, like closing files or releasing resources.
- Even if the **`try`** or **`except`** blocks contain **`return`** statements, the **`finally`** block still runs.

Example with finally:

```
python
```

```
def divide(x, y):
    try:
        result = x // y
    except ZeroDivisionError:
        print("Cannot divide by zero.")
    else:
        print("Result:", result)
    finally:
        print("This runs no matter what.")

# Usage
divide(10, 2)
divide(10, 0)
```

Output:

```
Result: 5
This runs no matter what.
Cannot divide by zero.
This runs no matter what.
```

2. Understanding multiple exceptions and custom exceptions.

Ans: In Python, you can handle multiple exceptions and also create your own custom exceptions for more precise error management.

➢ Handling Multiple Exceptions

You can catch multiple exceptions in a single `except` block by specifying them as a tuple. This is useful when the handling code is the same for all the exceptions.

Syntax:

```
python

    try:
        # code that may raise exceptions
    except (ExceptionType1, ExceptionType2) as e:
        print(f"An error occurred: {e}")
```

Example:

```
python

    try:
        x = int(input("Enter a number: "))
        y = int(input("Enter another number: "))
        result = x / y
    except (ValueError, ZeroDivisionError) as e:
        print("Error:", e)
    else:
        print("Result is", result)
```

- ➢ Here, both `ValueError` (if input is not a number) and `ZeroDivisionError` (if dividing by zero) are handled by the same `except` block.
- ➢ You can also use multiple `except` blocks if you want different handling for different exceptions.

➢ Creating Custom Exceptions

- Python allows defining custom exception classes by inheriting from the built-in `Exception` class. This is helpful for application-specific error handling.

Example:

```
python

    class CustomError(Exception):
        pass
    def check_value(x):
        if x < 0:
            raise CustomError("Negative values are not allowed")
    try:
        check_value(-1)
    except CustomError as e:
        print("Caught custom exception:", e)
```

➢ 6. Class & Object (Oops Concept)

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

Ans:

➢ Classes :

A class in Python is like a blueprint or template for creating objects. It defines a set of attributes (data) and methods (functions) that the created objects will have.

➢ Objects :

An object is an instance of a class. When you create (instantiate) a class, you get an object with its own unique data.

- For example, if `Dog` is a class, then `dog1` and `dog2` are objects (instances) of this class.

➢ Attributes :

Attributes are variables that belong to a class or its objects. They hold the data describing the object's state or properties.

Attributes can be:

- Instance attributes: Unique to each object, usually defined inside the `__init__` method using `self`.

  Example: `self.name = name`
- Class attributes: Shared by all objects of the class, defined directly inside the class but outside any method.

➢ Methods :

- Methods are functions defined inside a class that describe behavior or actions that objects can perform.
- They usually operate on the object's attributes, accessed via the `self` parameter.
- Methods are called on objects using dot notation.
- For example, `dog1.bark()` calls the `bark` method on the `dog1` object.

➢ Example :

```
python

    class Dog:

        species = "Canis familiaris"  # Class attribute


        def __init__(self, name, age):

            self.name = name          # Instance attribute

            self.age = age            # Instance attribute


        def bark(self):               # Method

            print(f"{self.name} is barking!")


    # Creating objects

    dog1 = Dog("Buddy", 3)

    dog2 = Dog("Charlie", 5)


    dog1.bark()  # Buddy is barking!
```

```
dog2.bark()  # Charlie is barking!

print(dog1.species)  # Canis familiaris

print(dog2.age)      # 5
```

➢ How it works:

- `Dog` is the class blueprint.
- `dog1` and `dog2` are two different objects with their own `name` and `age` attributes.
- `bark()` method performs an action associated with each dog, using their individual data.
- `species` is shared for all dogs (class attribute).

➢ Accessing Attributes and Methods

- Use dot notation to get or set attributes, and to call methods:

python

```
print(dog1.name)       # Access attribute

dog2.age = 6           # Modify attribute

dog1.bark()            # Call method
```

2.Difference between local and global variables.

Ans: Local and global variables in Python differ primarily in their scope and lifetime within a program:

➢ Local Variables

- Defined inside a function.

- Exist only during the function's execution.
- Cannot be accessed outside the function.
- Created when the function starts and destroyed when it ends.
- Local variables can shadow global variables if they share the same name within the function.

Example of local variable:

```python
def greet():
    msg = "Hello from inside the function!"
    print(msg)
greet()
# Accessing msg outside the function causes an error
# print(msg)  # NameError: name 'msg' is not defined
```

➢ Global Variables

- Defined outside any function, usually at the top of the program.
- Accessible anywhere in the program, including inside functions (unless shadowed).
- Persist for the lifetime of the program.
- Can be modified inside a function using the `global` keyword.

Example of global variable:

```python
a = 10  # Global variable

def print_a():
```

```
        print(a)  # Access global variable directly

    print_a()  # Prints: 10
```

➢ **Modifying a global variable inside a function:**

```
python

    a = 10

    def modify_a():

        global a

        a = 20

    modify_a()

    print(a)  # Prints 20
```

➢ Key Differences at a Glance:

Feature	Local Variable	Global Variable
Scope	Within the function only	Entire program
Lifetime	Only during function execution	Entire runtime of the program
Accessibility	Not accessible outside function	Accessible anywhere unless shadowed
Creation	Created when function is called	Created when program starts
Modification	Local to function, separate copy	Changes affect the variable globally
Usage of global	Not applicable	Use to modify global variable inside function

➢ 7.Inheritance

1.Single, Multilevel, Multiple, Hierarchical, and Hybrid inheritance in Python.
Ans:

➢ Types of Inheritance in Python

Inheritance lets a class (child) acquire properties and behaviors from another class (parent). Python supports several types depending on how many classes are involved and how they relate:

➢ 1. Single Inheritance

- One child inherits from a single parent class.
- Enables code reuse and extension.

Example:

```
python

class Parent:
    def func1(self):
        print("Parent function")

class Child(Parent):
    def func2(self):
        print("Child function")

obj = Child()
obj.func1()  # From Parent
obj.func2()  # Own method
```

➢ 2. Multiple Inheritance

- A child class inherits from two or more parent classes.
- Combines features from multiple sources.

Example:

```python
class Mother:
    def mother(self): print("Mother's method")

class Father:
    def father(self): print("Father's method")

class Child(Mother, Father):
    pass

obj = Child()
obj.mother()
obj.father()
```

➢ 3. Multilevel Inheritance

- A class inherits from a child class which itself inherits from a parent.
- Forms a chain of inheritance.

Example:

```python
class Grandparent:
    def greet(self): print("Hello from Grandparent")

class Parent(Grandparent):
    pass

class Child(Parent):
    pass

obj = Child()
obj.greet()  # Inherited from Grandparent
```

➢ 4. Hierarchical Inheritance

- Multiple child classes inherit from the same parent class.
- Useful to model different specialized versions sharing common traits.

Example:

```
python

class Parent:
    def greet(self): print("Greetings from Parent")

class Child1(Parent):
    pass

class Child2(Parent):
    pass

c1 = Child1()
c1.greet()
c2 = Child2()
c2.greet()
```

➢ 5. Hybrid Inheritance

- Combination of two or more types of inheritance such as single, multiple, and multilevel.
- Models complex class relationships.

Example:

```
python

class Person:
    def name(self): print("Person's name")

class Employee(Person):
    def role(self): print("Employee role")
```

```
class Project:
    def project_name(self): print("Project name")

class TeamLead(Employee, Project):
    pass

lead = TeamLead()
lead.name()
lead.role()
lead.project_name()
```

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

Ans :

The `super()` function in Python is used in a child class to access methods and properties of its parent class. It is most commonly used to call the parent class's constructor (`__init__` method) to initialize inherited attributes without explicitly naming the parent class. This makes code more maintainable, especially in complex inheritance scenarios.

➢ Key Benefits of `super()`

- Avoids hardcoding the parent class name.
- Works well with single, multiple, and multilevel inheritance.
- Helps reuse and extend parent class functionality in child classes.

➢ Basic Example of Using `super()`

python

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

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # Call parent constructor
        self.breed = breed
```

```
    def info(self):
        print(f"{self.name} is a {self.breed}")

dog = Dog("Buddy", "Golden Retriever")
dog.info()  # Output: Buddy is a Golden Retriever
```

- In this example, `super().__init__(name)` calls the `__init__` of `Animal` to initialize the `name` attribute for the `Dog` class, ensuring that the parent's initialization logic is properly reused.

➢ How `super()` Works with Multiple Inheritance

➢ In more complex cases with multiple inheritance, `super()` follows the Method Resolution Order (MRO) to call the next method in the inheritance chain appropriately, preventing duplicate calls.

➢ Method Resolution Order (MRO)

➢ Python determines the order in which base classes are searched when looking for a method. This order can be viewed using:

```
Python

print(ClassName.mro())
```

- `super()` thus provides a clean and flexible way to work with inheritance by enabling child classes to leverage the functionality of their parent classes safely and efficiently.

➢ Why Use `super()` in Multiple Inheritance?

- To avoid explicitly naming parent classes, making code easier to maintain.
- To ensure all parent classes' methods are called appropriately, avoiding duplicated calls or missed calls.
- To respect the MRO, which manages complex class hierarchies safely.

➢ 8. Method Overloading and Overriding

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

Ans :

Method overloading, in general, refers to defining multiple methods in the same class with the same name but different parameters (different number or types of arguments) to perform different tasks.

➢ Method Overloading in Python

- Unlike some other languages (Java, C++), Python does NOT support method overloading out of the box.
- Python allows only one method with a given name in a class - if you define multiple, only the last one is used.
- To achieve similar behavior, Python programmers use:
  - ➢ Default arguments
  - ➢ Variable-length arguments (*args, **kwargs)
  - ➢ Type checking inside a single method
  - ➢ External libraries like `multipledispatch` for true method overloading via decorators

➢ Example: Using Default Arguments to Simulate Overloading

python

```
class Calculator:
    def add(self, a, b, c=None):
        if c is not None:
            return a + b + c
        else:
            return a + b

calc = Calculator()
print(calc.add(2, 3))      # Output: 5
print(calc.add(2, 3, 4))   # Output: 9
```

➢ Example: Using `multipledispatch` Library for Overloading

python

```
from multipledispatch import dispatch

class Calculator:

    @dispatch(int, int)
    def add(self, a, b):
        return a + b

    @dispatch(int, int, int)
    def add(self, a, b, c):
        return a + b + c

calc = Calculator()
print(calc.add(2, 3))      # Output: 5
print(calc.add(2, 3, 4))   # Output: 9
```

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

Ans:

Method overriding in Python occurs when a child class defines a method with the same name and signature as a method in its parent class, effectively replacing or modifying the parent's implementation for instances of the child.

➢ Key Points about Method Overriding:

- Allows a child class to provide its own version of a method inherited from the parent class.
- The overridden method in the child class is called instead of the parent's method on child instances.
- Supports polymorphism, enabling objects of different classes to be treated through a common interface but behave differently.
- The child class can still call the parent's overridden method using `super()` if needed.

➢ **Basic Example of Method Overriding in Python:**

python

```python
class Parent:
    def show(self):
        print("Inside Parent")
class Child(Parent):
    def show(self):
        print("Inside Child")
obj1 = Parent()
obj2 = Child()

obj1.show()  # Output: Inside Parent
obj2.show()  # Output: Inside Child
```

➢ **Example with `super()` to Extend Parent Method:**

python

```python
class Person:
    def introduce(self):
        print("Hello, I'm a person.")
class Programmer(Person):
    def introduce(self):
        super().introduce()
        print("And I write Python code.")

p = Programmer()
p.introduce()
#  Output:
# Hello, I'm a person.
# And I write Python code.
```

➢ Real-world Example: Specialized Employee Roles

python

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

    def get_salary(self):
        return self.salary

class Manager(Employee):
    def __init__(self, name, salary, incentive):
        super().__init__(name, salary)
        self.incentive = incentive

    def get_salary(self):
        # Override to add incentive
        return self.salary + self.incentive

e = Employee("Rahul", 5000)
m = Manager("Vipul", 7000, 2000)

print(e.get_salary())  # 5000
print(m.get_salary())  # 9000
```

➢ 9. SQLite3 and PyMySQL (Database Connectors)

1.Introduction to SQLite3 and PyMySQL for database connectivity.

Ans:

➢ SQLite3

- SQLite3 is a lightweight, serverless, file-based relational database engine.
- It is included by default in Python's standard library as the `sqlite3` module (no extra installation required).
- Ideal for small to medium-sized applications, prototyping, and development.
- Stores the entire database in a single file on disk.
- Supports standard SQL for queries and transactions.

➢ Key Features & Usage

- Connect to a SQLite database file or create one if it doesn't exist:

  python

```
import sqlite3
conn = sqlite3.connect('example.db')
```

- Create a cursor object to interact with the database:

  python

```
cursor = conn.cursor()
```

- Execute SQL commands to create tables, insert, update, delete, and query data.
- Commit changes and close the connection.
- Supports transactions with `commit()` and rollback with `rollback()`.

Example:

python

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

```
cursor = conn.cursor()
cursor.execute('CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY,
name TEXT, age INTEGER)')

cursor.execute('INSERT INTO users (name, age) VALUES (?, ?)',('Alice', 30))

conn.commit()

cursor.execute('SELECT * FROM users')
print(cursor.fetchall())

conn.close()
```

➢ PyMySQL

- PyMySQL is a pure-Python MySQL client library for connecting Python applications to MySQL or MariaDB databases.
- Requires installation via pip:

```
pip install PyMySQL
```

- Suitable for applications requiring a full-fledged client-server database.
- Allows interaction with MySQL databases over a network or locally.

➢ Key Features & Usage

- Connect to a MySQL database with hostname, username, password, and database name:

```python
import pymysql
connection = pymysql.connect(
    host='localhost',
```

```
    user='root',
    password='password',
    database='testdb'
)
```

- Cursor creation and SQL execution are similar to sqlite3.
- Supports transactions, stored procedures, and more advanced MySQL features.

Example:

```
python
```

```
import pymysql

connection = pymysql.connect(host='localhost', user='root',
password='password', database='testdb')
cursor = connection.cursor()

cursor.execute('CREATE TABLE IF NOT EXISTS users (id INT PRIMARY KEY
AUTO_INCREMENT, name VARCHAR(100), age INT)')
cursor.execute('INSERT INTO users (name, age) VALUES (%s, %s)', ('Bob',
25))

connection.commit()

cursor.execute('SELECT * FROM users')
print(cursor.fetchall())
connection.close()
```

- **Summary Comparison:**

Aspect	SQLite3	PyMySQL (MySQL)
Database Type	File-based, serverless	Client-server database
Installation	Part of Python standard library	Requires external package
Use Cases	Lightweight projects, prototypi	Large-scale, networked DB apps
Features	Basic SQL functionality	Full MySQL features support
Connection	File path	Host, port, credentials

2. Creating and executing SQL queries from Python using these connectors.
Ans:

Creating and executing SQL queries from Python using SQLite3 and PyMySQL involves connecting to the database, executing SQL commands through cursors, handling transactions, fetching results, and closing the connections properly.

➢ SQLite3 Example: Creating and Executing SQL Queries

python

```
import sqlite3
# Connect to SQLite database file (or create it if it doesn't exist)
conn = sqlite3.connect('example.db')
cursor = conn.cursor()

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

```
# Insert a record using parameter substitution to prevent injection
cursor.execute('INSERT INTO users (name, age) VALUES (?, ?)', ('Alice',
30))

# Commit the transaction to save changes
conn.commit()

# Query data
cursor.execute('SELECT * FROM users')
rows = cursor.fetchall()

for row in rows:
    print(row)

# Close connection
conn.close()
```

➢ PyMySQL Example: Creating and Executing SQL Queries

python

```
import pymysql

# Connect to MySQL server
conn = pymysql.connect(
    host='localhost',
    user='root',
    password='your_password',
    database='testdb'
)
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 record with parameterized query
cursor.execute('INSERT INTO users (name, age) VALUES (%s, %s)', ('Bob',
25))

# Commit changes
conn.commit()

# Fetch data
cursor.execute('SELECT * FROM users')
rows = cursor.fetchall()

for row in rows:
    print(row)

# Close connection
conn.close()
```

➢Key Points:

- Use SQL commands as strings passed to `execute()` method of cursor.
- Delegate input values using placeholders (`?` for SQLite3, `%s` for PyMySQL).
- Commit transactions for modifying statements.
- Fetch results with `fetchall()` or `fetchone()`.
- Always close database connections to free resources.

➢10. Search and Match Functions

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

Ans :

`re.match()` and `re.search()` are both used for pattern matching with regular expressions, but they differ in where they look for matches within a string.

➢`re.match()`

- Checks for a match only at the beginning of the string.
- Returns a match object if the pattern matches starting at index 0.
- Returns `None` if the pattern is not found at the start.

Example:

```
python

   import re
   s = "Hello, world!"
   result = re.match(r"Hello", s)
   print(result)  # Match object because 'Hello' is at the start

   result_none = re.match(r"world", s)
   print(result_none)  # None, 'world' isn't at the start
```

➢`re.search()`

- Scans the entire string for the first location where the pattern matches.
- Returns a match object for the first occurrence anywhere in the string.
- Returns `None` if the pattern is not found at all.

Example:

```
python

   import re
   s = "Say Hello to the world"
```

```
result = re.search(r"Hello", s)
print(result)  # Match object; 'Hello' found at position 4

result_none = re.search(r"Python", s)
print(result_none)  # None, 'Python' not found anywhere
```

➢ Summary Table

Feature	re.match()	re.search()
Search Location	Beginning of the string only	Anywhere in the string
Return Value	Match object if matched at start	Match object for first found match
Use Case	Validate if string starts with patte	Find pattern anywhere in the string
Example	re.match(r"^Hello", s)	re.search(r"Hello", s)

➢ **When to use:**

- Use `re.match()` if you want to confirm the string starts with a pattern
- Use `re.search()` when the pattern may occur anywhere in the string

2. Difference between search and match.

Ans: The fundamental difference between **`re.search()`** and **`re.match()`** in Python's **`re`** module is about where they check for the pattern match in the string.

➢ re.match()

- Checks for a match only at the beginning of the string (position 0).
- Returns a match object if the pattern matches at the start.
- Returns **`None`** if the pattern is not found at the beginning.

➢ re.search()

- Scans the entire string for the first occurrence of the pattern.
- Returns a match object for the first found match anywhere in the string.
- Returns `None` if the pattern is not found at all.

➢ Example demonstrating the difference:

```python
import re

text = "Hello, welcome to Python."

# Using re.match
match_result = re.match(r"Hello", text)
print(match_result)  # Match object since 'Hello' is at the beginning

match_result_none = re.match(r"welcome", text)
print(match_result_none)  # None since 'welcome' is not at the start

# Using re.search
search_result = re.search(r"welcome", text)
print(search_result)  # Match object found for 'welcome'

search_result_none = re.search(r"Java", text)
print(search_result_none)  # None since 'Java' is not in the string
```

➢ Summary Table

Feature	re.match()	re.search()
Search Start	Only at the beginning of string	Anywhere in the string
Returns	Match object or None	Match object or None
Use Case	Check string starts with pattern	Find pattern anywhere
Performance	Slightly faster for start match	Flexible, scans entire string