

❖ Advance Python Programming

(1) Printing on Screen

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

- The `print()` function is a built-in Python function used to display output on the console (standard output device). It is one of the most fundamental and frequently used tools in Python programming for tasks such as showing messages to users, displaying the values of variables, and debugging code.

➤ Basic Usage

- The basic syntax involves calling the function with the item you want to display inside the parentheses:

```
print("Hello, World!")
```

```
# Output: Hello, World!
```

```
# Printing a variable's value
```

```
name = "Alice"
```

```
age = 25
```

```
print("Name:", name, "Age:", age)
```

```
# Output: Name: Alice Age: 25
```

- You can pass multiple items (strings, numbers, lists, etc.) to the `print()` function by separating them with commas.

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

- In Python, string formatting is primarily handled using f-strings (formatted string literals) and the older, but still valid, `format()` method. F-strings are the recommended approach for modern Python (3.6+) due to their conciseness, readability, and speed.

➤ Using f-strings

- F-strings are prefixed with an `f` or `F`, allowing you to embed Python expressions and variables directly within curly braces `{ }`.

- Basic Usage

- To use f-strings, simply place the variable or expression inside the braces:

```
name = "Alice"
age = 30
print(f"Hello, my name is {name} and I am {age} years old.")
# Output: Hello, my name is Alice and I am 30 years old.
```

```
# You can also use expressions directly
print(f"Next year, I will be {age + 1} years old.")
# Output: Next year, I will be 31 years old.
```

➤ Using the `format()` method

- Before f-strings, the `format()` method was the standard for formatting. It is still fully functional and useful in scenarios where dynamic formatting is required or for compatibility with older Python versions.

- **Basic Usage**

- Placeholders are defined by curly braces {}, and the values are passed as arguments to the format() method.

```
name = "Bob"
```

```
age = 25
```

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

Output: Hello, my name is Bob and I am 25 years old.

You can also use named arguments for clarity

```
print("Hello, my name is {n} and I am {a} years old.".format(n=name, a=age))
```

Output: Hello, my name is Bob and I am 25 years old.

(2) Reading Data from Keyboard

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

- In Python, the built-in function `input()` is used to read a line of text entered by the user from the keyboard. The function pauses your program's execution until the user presses the Enter key.

- **Syntax**

```
input(prompt)
```

- **Example:**

```
name = input("Enter your name: ")  
  
print("Hello", name)
```

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

- The `input()` function always returns what the user types as a string, even if they enter numbers.
- In Python, you can convert user input into various data types using built-in functions such as `int()`, `float()`, and others.
- To perform numeric operations (like adding, subtracting, etc.), you must convert that string to the correct data type (like `int` or `float`).

- **Primary Functions For Type Conversion:**

- **int(x)**: Converts x to an integer.
- **float(x)**: Converts x to a floating-point number (a number with a decimal).
- **str(x)**: Converts x to a string (usually used to display numerical data alongside text).
- **bool(x)**: Converts x to a boolean value (True or False).

- **Example:**

(1) Converting to int (Integer)

- Use int() to convert a string to an integer.

```
age = int(input("Enter your age: "))  
print("Your Age is:", age)
```

(2) Converting to float (Floating-Point Number)

- Use float() to convert a string to a floating-point number (with decimals).

```
price = float(input("Enter price: "))  
total = price * 1.1  
print("Price with tax:", total)
```

(3) Converting Back to str (String)

- Sometimes you convert numbers back to strings for printing or formatting.

```
num = 10  
  
text = "The number is " + str(num)  
  
print(text)
```

(4) Converting to Boolean

- While less common for direct conversion of typical user input, you can use `bool()`:

```
user_input = input("Type something (e.g., 'yes' or leave blank): ")  
is_present = bool(user_input)  
print(f"Is input present? {is_present}")
```

(3) Opening and Closing Files

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

- In Python, the `open()` function is used to interact with files. The function accepts a file path and a mode string as arguments, determining how the file is accessed. The primary modes are 'r' (read), 'w' (write), and 'a' (append), which can be combined with other characters to modify their behavior, such as '+' for reading and writing.

- Common Modes

Mode	Name	Description	If file doesn't exist	If file exists
'r'	Read	Opens the file for reading only. This is the default mode.	Raises a FileNotFoundError.	Opens the file at the beginning.
'w'	Write	Opens the file for writing.	Creates a new file.	Truncates (clears) the existing file content.
'a'	Append	Opens the file for writing, but adds data to the end of the file.	Creates a new file.	Appends new data to the end of the existing content.

- **Read/Write Combination Modes**

Mode	Description
'r+'	Read and Write. Opens the file for both reading and writing. The file pointer is at the beginning. The file must already exist; otherwise, it raises an error.
'w+'	Write and Read. Opens the file for both writing and reading. This mode creates a new file if it doesn't exist, and truncates (clears) the file if it does.

- **Example:**

'r' mode: Reading from a file

try:

 with open('example.txt', 'r') as file:

 content = file.read()

 print("Read content:", content)

except FileNotFoundError:

 print("File not found when trying to read.")

'w' mode: Writing to a file (overwrites existing content)

with open('example.txt', 'w') as file:

 file.write("Hello, world!")

'a' mode: Appending to a file

with open('example.txt', 'a') as file:

 file.write("\nThis line is appended.")

'r+' mode: Reading and writing (file must exist)

with open('example.txt', 'r+') as file:

 # Read the current content

 content = file.read()

 print("Current content:", content)

 # Write new content at the current position (end of file after read())

```
file.write("\nAdded after reading.")  
# To write at the beginning, you would use file.seek(0)
```

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

- The built-in Python function `open()` is used to create and access files. It requires at least one argument, the file path, and returns a file object which can be used to read from or write to the file. The basic syntax is `open(file, mode)`, where `mode` specifies how the file will be used.

- **The `open()` function syntax**

```
file_object = open(file, mode, buffering, encoding, errors, newline,  
closefd, opener)
```

file: The path to the file you want to open (required).

mode: Specifies the purpose of opening the file (e.g., read, write, append). This is optional and defaults to 'r' (read mode).

buffering, encoding, etc.: Other optional arguments that control file processing.

- **Using `with open()`**

- It is best practice to use the `with` statement with `open()` as it automatically closes the file, even if errors occur, preventing resource leaks.

- **Example: Creating and Writing to a file (using 'w')**

- This code creates a file named `my_file.txt` and writes text to it.

'w' mode will create the file if it doesn't exist, or overwrite it if it does

try:

```
with open('my_file.txt', 'w') as file:
    file.write("Hello, world!\n")
    file.write("This is a new line of text.")
    print("Successfully wrote to my_file.txt")
except IOError as e:
    print(f"Error writing to file: {e}")
```

- **Example: Reading from a file (using 'r')**

- This code reads the content from the created file and prints it.

'r' mode opens the file for reading

try:

```
with open('my_file.txt', 'r') as file:
    content = file.read()
    print("\nContent of my_file.txt:")
    print(content)
except FileNotFoundError:
    print("Error: The file my_file.txt was not found.")
except IOError as e:
    print(f"Error reading file: {e}")
```

- **Example: Appending to a file (using 'a')**

- This code adds a new line to the existing file without erasing the previous content.

'a' mode appends to the end of the file

try:

```
with open('my_file.txt', 'a') as file:
    file.write("\nThis line was appended later.")
    print("\nSuccessfully appended to my_file.txt")
```

```
except IOError as e:  
    print(f"Error appending to file: {e}")
```

Q-3) Closing files using close().

- In Python, the `close()` method is used to manually close an opened file, freeing up system resources and ensuring data is saved.
- However, the recommended and safest method is to use the `with open()` statement, which automatically handles closing the file, even if errors occur.

- **Using the `close()` Method Manually**

- When a file is opened using the `open()` function, a file object is returned. You can call the `close()` method on this object to close the file explicitly.

```
# Open a file in write mode ('w')  
file_object = open("example.txt", "w")
```

```
try:
```

```
    file_object.write("Hello, World!") # Write data to the file
```

```
finally:
```

```
    file_object.close() # Ensure the file is closed, even if an error  
    occurred in the try block
```

(4) Reading and Writing Files

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

- Reading from a file in Python can be accomplished using the built in `read()`, `readline()`, and `readlines()` methods. All these operations require first opening the file using the `open()` function, typically managed within a `with` statement to ensure proper file closure.

- Using `read()`

- The `read()` method is used to read the entire contents of a file as a single string. You can optionally pass an integer argument to limit the number of characters read.

- Example

```
# Read the entire file
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)
```

```
# Read only the first 10 characters
with open('example.txt', 'r') as file:
    first_10_chars = file.read(10)
    print(first_10_chars)
```

- **Using readline()**

- The `readline()` method reads a single line from the file each time it is called. It includes the newline character (`\n`) at the end of the line. This is useful for processing files line by line efficiently.

- **Example**

with `open('example.txt', 'r')` as file:

```
line1 = file.readline()
line2 = file.readline()
print(f"Line 1: {line1}", end="")
print(f"Line 2: {line2}", end="")
```

- **Using readlines()**

- The `readlines()` method reads all the lines of the file into a list. Each element in the list is a single line from the file, including the newline character.

- **Example**

with `open('example.txt', 'r')` as file:

```
all_lines_list = file.readlines()
for line in all_lines_list:
    print(line, end="")
```

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

- Writing to a file in Python can be done using either the write() or the writelines() method. The primary difference lies in how each handles data types:

write() : takes a single string argument and writes it to the file.

writelines() : takes an iterable (like a list) of strings and writes all items sequentially to the file.

- **Using write()**

- The write() method is used for writing a single string at a time. It is useful for writing line by line or outputting the result of some processing.
- Crucially, write() does not automatically add a newline character (\n) at the end of the string; you must include it manually if you want each string on a new line.

- **Example**

```
with open(file_path, 'w') as file:  
    file.write("Hello, world!\n")  
    file.write("This is a second line.\n")  
    file.write("Writing a final line.")
```

```
# Result in example.txt:  
# Hello, world!  
# This is a second line.  
# Writing a final line.
```

- **Using writelines()**

- The writelines() method accepts an iterable of strings (e.g., a list or a tuple). It efficiently writes all elements in the sequence to the file without needing a loop.
- Similar to write(), writelines() also does not automatically add newline characters between the items. You must ensure that the newline character is part of the strings within your iterable if you desire line breaks.

- **Example**

```
lines_to_write = [  
    "First line of text\n",  
    "Second line here\n",  
    "And the third line"  
]
```

```
with open(file_path, 'w') as file:  
    file.writelines(lines_to_write)
```

```
# Result in example.txt:  
# First line of text  
# Second line here  
# And the third line
```

(5) Exception Handling

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

- In Python, an exception is an error that occurs during the execution of a program, interrupting its normal flow. Python provides a structured mechanism using try, except, and finally blocks to handle these errors gracefully and prevent the program from crashing.
- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it.
- Errors detected during execution are called exceptions.

- Components

- **try**: This block contains the code that might raise an exception.
- **except**: This block catches and handles the exception if one occurs within the try block.
- **else**: The code in this block executes if no exception was raised in the try block.
- **finally**: This block's code always runs, regardless of whether an exception occurred or not. It is typically used for cleanup actions, such as closing files or network connections.

- The try and except Blocks

- The core of exception handling involves wrapping the potentially problematic code in a try block. If an exception occurs within that block, Python stops executing the try block's code and jumps to the associated except block.

- **The finally Block**

- The finally block is optional but highly useful. The code inside this block will always execute, regardless of whether an exception occurred in the try block or was caught by an except block. This is ideal for crucial cleanup operations like closing files or network connections.

- **Example**

```
try:
```

```
    # Code that might raise an exception (e.g., FileNotFoundError,
    ZeroDivisionError)
```

```
    file_path = "data.txt"
```

```
    with open(file_path, 'r') as file:
```

```
        data = file.read()
```

```
        print("File content successfully read!")
```

```
    # Example of a potential error within the try block
```

```
    result = 10 / 0 # This will raise a ZeroDivisionError
```

```
    print(f"Result is: {result}")
```

```
except FileNotFoundError:
```

```
    # Handle a specific exception (file not found)
```

```
    print(f"Error: The file '{file_path}' was not found.")
```

```
except ZeroDivisionError:

    # Handle another specific exception (division by zero)

    print("Error: You cannot divide by zero!")


except Exception as e:

    # Handle any other unexpected exceptions

    print(f"An unexpected error occurred: {e}")


finally:

    # This code will always run, useful for cleanup

    print("Program execution completed and resources handled.")
```

Q-2) Understanding multiple exceptions and custom exceptions.

- In Python, you handle multiple exceptions by using several `except` blocks or grouping exceptions in a tuple, and you create custom exceptions by defining a new class that inherits from the built-in `Exception` class.
- **Handling Multiple Exceptions**
 - When code within a try block can raise more than one type of exception, Python offers two main methods to handle them.

(1) Multiple except Blocks

- You can use a separate except block for each specific exception type that requires different handling logic. Python will execute the first except block that matches the exception raised.

- **Example**

try:

```
# Code that might raise ValueError or ZeroDivisionError
```

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

```
result = 10 / number
```

except ValueError:

```
    print("Error: Invalid input. Please enter a valid integer.")
```

except ZeroDivisionError:

```
    print("Error: Cannot divide by zero!")
```

except Exception as e:

```
# A general handler for any other unanticipated exceptions
```

```
    print(f"An unexpected error occurred: {e}")
```

else:

```
    print(f"Result is {result}")
```

finally:

```
    print("Execution finished.") # This runs regardless of an exception
```

(2) Grouping in a Single except Block

- If multiple exceptions require the same handling logic, you can list them as a tuple in a single except clause.

- **Example**

try:

```
# Code that might raise FileNotFoundError or ConnectionError
```

```
# Example function fetch_data might raise either
```

```
data = fetch_data(source)
```

```
except (FileNotFoundError, ConnectionError) as e:
```

```
    print(f"A recoverable error occurred: {e}")
```

- **Creating Custom Exceptions**

- Custom exceptions allow you to define specific, domain-related errors, making your code more readable, maintainable, and easier to debug. They are especially useful for enforcing business rules that built-in exceptions don't cover.

- **Steps to Create and Use a Custom Exception**

1. **Define the Class**: Create a new class that inherits from the built-in Exception class (or a more specific built-in exception like ValueError if appropriate).

2. **Raise the Exception**: Use the raise keyword to signal the error when a specific condition is met in your code.
3. **Handle the Exception**: Use standard try...except blocks to catch your custom exception.

- **Example**

```
class InvalidAgeError(Exception):

    """Exception raised for invalid age values."""

    def __init__(self, age, message="Age must be between 0 and 120"):

        self.age = age

        self.message = message

        super().__init__(self.message) # Initialize the base Exception class

    def __str__(self):

        return f'{self.age} -> {self.message}' # Customize the error message output


def set_age(age):

    if not isinstance(age, int):

        raise TypeError("Age must be an integer.")

    if age < 0 or age > 120:

        raise InvalidAgeError(age)

    print(f"Age set to {age}")
```

```
# Handling the custom exception

try:
    set_age(150)
except InvalidAgeError as e:
    print(f"Caught custom exception: {e}")
except TypeError as e:
    print(f"Caught built-in exception: {e}")
```

(6) Class and Object (OOP Concepts)

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

- Classes, objects, attributes, and methods are foundational concepts of Object-Oriented Programming (OOP) in Python. A class serves as a blueprint, defining the structure and behavior of objects, which are instances of that class.

- Class

- A class is a blueprint or template for creating objects. It defines a set of properties (attributes) and behaviors (methods) that all objects created from that class will share.

- Syntax:

```
class Dog:  
    # Attributes and methods go here  
    pass
```

- Object

- An object, also known as an instance, is a concrete realization of a class. When you create an object, you are building a specific item based on the template defined by the class.

- Syntax:

```
my_dog = Dog() # 'my_dog' is an object (instance) of the 'Dog'  
class  
your_dog = Dog()
```

- **Attribute**

- Attributes are variables that store data within a class or an object. They represent the state or properties of an object.

- **Syntax:**

```
class Dog:
    def __init__(self, name, breed):
        self.name = name    # Instance attribute
        self.breed = breed  # Instance attribute

my_dog = Dog("Buddy", "Golden Retriever")
print(my_dog.name) # Accessing the attribute: "Buddy"
```

- **Method**

- Methods are functions defined inside a class that describe the behaviors or actions an object can perform. They operate on the object's attributes.

- **Syntax:**

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

    def bark(self): # This is a method
        return f"{self.name} is barking!"

    def fetch(self, item): # Another method
        return f"{self.name} is fetching the {item}!"

my_dog = Dog("Buddy", "Golden Retriever")
```

```
print(my_dog.bark()) # Calling the method: "Buddy is barking!"  
print(my_dog.fetch("ball")) # "Buddy is fetching the ball!"
```

Q-2) Difference between local and global variables.

- In Python, the primary difference between local and global variables lies in their scope (accessibility) and lifetime.

➤ Key Differences: Local vs. Global Variables

Comparison Basis	Local Variable	Global Variable
Definition	Declared inside a function or a specific code block.	Declared outside of all functions, at the module level.
Scope/Accessibility	Accessible only within the function where it is defined.	Accessible from anywhere in the program, including inside functions.
Lifetime	Created when the function is called and destroyed when the function's execution ends.	Created when the program starts and persists until the program terminates.
Data Sharing	Cannot be used for data sharing between different functions.	Can be used to share data among multiple functions.
Modification inside function	By default, an assignment in a function creates a new local variable, even if a global variable has the same name.	To modify a global variable inside a function, you must explicitly use the global keyword.

- **Example**

```
# Global variable

global_var = 10


def my_function():

    # Local variable with the same name shadows the global one

    local_var = 20

    print(f"Inside function, local_var: {local_var}")

    # Accessing the global variable without using the 'global'
    keyword to modify it

    print(f"Inside function, accessing global_var: {global_var}")


my_function()

print(f"Outside function, global_var: {global_var}")

# print(local_var) would cause a NameError because local_var is
not defined in the global scope
```

Output:

```
Inside function, local_var: 20
Inside function, accessing global_var: 10
Outside function, global_var: 10
```

(7) Inheritance

Q-1) Single, Multilevel, Multiple, Hierarchical, and Hybrid inheritance in Python.

- Python supports five types of inheritance, each defining how classes can share and reuse code: Single, Multilevel, Multiple, Hierarchical, and Hybrid inheritance.

- **Types of Inheritance in Python**

- **Single Inheritance**

- In single inheritance, a child class inherits from only one parent class. This is the simplest and most common form of inheritance, creating a clear, linear hierarchy.
 - For an example of single inheritance in Python, please refer to the code provided in reference.

- **Multilevel Inheritance**

- Multilevel inheritance involves a chain where a class inherits from a child class, which itself inherits from a grandparent class. An example demonstrating this can be found in the code in reference.

- **Multiple Inheritance**

- Multiple inheritance allows a child class to inherit from two or more parent classes, combining their functionalities.

Python uses Method Resolution Order (MRO) to manage potential conflicts from similarly named methods in different parent classes. An example of multiple inheritance is available in reference.

- **Hierarchical Inheritance**

- Hierarchical inheritance occurs when multiple child classes inherit from a single parent class. This structure is useful for classes that share a common foundation but have distinct behaviors. For a code example of hierarchical inheritance, see reference.

- **Hybrid Inheritance**

- Hybrid inheritance combines two or more inheritance types within the same structure, such as multilevel and multiple inheritance. This allows for complex class designs, and Python utilizes MRO to handle their complexities. An example illustrating hybrid inheritance is provided in reference.

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

- To access properties (including methods and attributes) of a parent class in Python, the `super()` function is used primarily within the child class's methods. This function provides a proxy object that delegates method calls to the parent or sibling class.

- **Key Usage Scenarios**

(1) Calling the Parent Class's __init__ Method

- The most common use of super() is within the child class's __init__ method to ensure the parent class's initialization logic (setting up attributes) is executed. This prevents overwriting essential parent attributes.

- **Example**

```
class ParentClass:
    def __init__(self, name):
        self.name = name
        print(f"Parent Class __init__ called for {self.name}")

class ChildClass(ParentClass):
    def __init__(self, name, age):
        # Call the parent class's __init__ method using super()
        super().__init__(name)
        self.age = age
        print(f"Child Class __init__ called for {self.name}, Age: {self.age}")

# Create an instance of the child class
child_instance = ChildClass("Alice", 30)

# Accessing properties defined in the parent class
print(f"Name (from parent): {child_instance.name}")
print(f"Age (from child): {child_instance.age}")
```

(2) Overriding and Extending Parent Methods

- `super()` allows a child class to override a parent method but still call the original parent method within the new, extended logic.

- Example

```
class Animal:
    def speak(self):
        return "Generic animal sound"

class Dog(Animal):
    def speak(self):
        # Call the parent class's speak method
        parent_sound = super().speak()
        return f"{parent_sound} and Bark!"

# Create an instance of the dog class
my_dog = Dog()
print(my_dog.speak())
```

(8) Method Overloading and Overriding

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

- Method overloading, which involves defining multiple methods with the same name but different parameters, is not directly supported as a native feature in Python in the same way it is in statically typed languages like Java or C++. Python is dynamically typed and uses a different mechanism to handle this concept.

- Achieving Overloading-like Behavior

- Here are common Python ways to emulate method overloading:

(1) Using Default Arguments

- This is the most "Pythonic" way to provide flexibility in the number of arguments:

- Example

```
class Example:
    def greet(self, name="World", greeting="Hello"):
        """Greets the specified person with a greeting."""
        print(f'{greeting}, {name}!')

# Examples of calling the method
obj = Example()
obj.greet()                # Output: Hello, World!
obj.greet("Alice")         # Output: Hello, Alice!
obj.greet("Bob", "Hi")     # Output: Hi, Bob!
```

(2) Checking Argument Types and Count within the Function

- You can use checks like `isinstance()` or examine the number of arguments within the function to alter its behavior:

- **Example**

```
class Example:
    def display(self, data):
        if isinstance(data, int):
            print(f"Displaying an integer: {data}")
        elif isinstance(data, str):
            print(f"Displaying a string: {data}")
        elif isinstance(data, list):
            print(f"Displaying a list of size {len(data)}")

# Examples of calling the method
obj = Example()
obj.display(10)
obj.display("Hello")
obj.display([1, 2, 3])
```

(3) Using the `functools.singledispatch` Decorator

- For a more robust and cleaner way to handle different types of the first argument, Python provides the `functools.singledispatch` decorator. This is a form of generic functions, not true method overloading, but it achieves the desired result for different argument types:

- **Example**

```
from functools import singledispatch

class Example:

    @singledispatch
```

```
def process(self, arg):  
    print(f"Generic processing for type: {type(arg)}")
```

```
@process.register
```

```
def _(self, arg: int):  
    print(f"Processing integer: {arg * 2}")
```

```
@process.register
```

```
def _(self, arg: str):  
    print(f"Processing string: {arg.upper()}")
```

This approach is generally used for standalone functions,

but a class method variant also exists.

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

- Method overriding is a core concept in object-oriented programming, where a child (derived) class provides a specific implementation of a method that is already defined in its parent (base) class. This allows a child class to modify or extend the behavior inherited from its parent.

- **Example**

```
class Animal:
    def speak(self):
        return "Animal makes a generic sound"

class Dog(Animal):
    # Method overriding the speak() method from the parent class
    def speak(self):
        return "Woof woof!"

# Create objects
generic_animal = Animal()
dog = Dog()

# Calling the speak method
print(f"Generic Animal speaks: {generic_animal.speak()}")
print(f"Dog speaks: {dog.speak()}")
```

Output:

Generic Animal speaks: Animal makes a generic sound
Dog speaks: Woof woof!

- **Using super()**

- Often, when overriding a method, you might want to extend the parent method's functionality rather than completely replacing it. Python's built-in `super()` function allows you to call the parent class's method from within the child class's overridden method.

- **Example**

```
class Animal:

    def speak(self):

        return "Animal makes a generic sound"

class Cat(Animal):

    def speak(self):

        # Call the parent's method using super()

        parent_sound = super().speak()

        return f"{parent_sound}. A cat adds 'Meow!'"

# Create a Cat object

cat = Cat()

# Calling the speak method

print(f"Cat speaks: {cat.speak()}")
```

Output:

Cat speaks: Animal makes a generic sound. A cat adds 'Meow!'

(9) SQLite3 and PyMySQL (Database Connectors)

Q-1) Introduction to SQLite3 and PyMySQL for database connectivity.

- SQLite3 and MySQL are popular relational database management systems (RDBMS) that offer robust connectivity options within Python. SQLite3 is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured SQL database engine, while MySQL is an open-source enterprise-level database server.

- **SQLite3 in Python**

- SQLite is serverless, meaning it reads and writes directly to an ordinary disk file. This makes it a great choice for embedded systems, mobile applications, or local application data storage.
- **Driver:** Python's standard library includes the built-in sqlite3 module, so no external installation is required.
- **Connection:** You establish a connection to a database file using the connect() function. If the specified file does not exist, the sqlite3 module will create it automatically.

- **Example**

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Create table
cursor.execute("CREATE TABLE IF NOT EXISTS stocks (date
text, trans text, symbol text, qty real, price real)")

# Insert a row of data
cursor.execute("INSERT INTO stocks VALUES ('2026-01-
16','BUY','RHAT',100,35.14)")

# Save (commit) the changes
conn.commit()

# We can also close the connection if we are done with it.
# Just be sure any pending transactions have been committed or
they will be lost.
conn.close()
```

- **MySQL in Python**

- MySQL is a client-server system that requires a separate server process to be running. It is a powerful solution commonly used in large-scale web applications and enterprise environments.

Driver: You must install an external driver, typically mysql-connector-python, using pip install mysql-connector-python.

Connection: You connect using your server credentials (username, password, host, and database name).

- **Example**

```
import mysql.connector

try:
    conn = mysql.connector.connect(
        host="localhost",
        user="yourusername",
        password="yourpassword",
        database="yourdatabase"
    )
    cursor = conn.cursor()

    # Create table (example)
    cursor.execute("CREATE TABLE IF NOT EXISTS employees
(id INT AUTO_INCREMENT PRIMARY KEY, name
VARCHAR(255), salary INT)")

    # Insert a row of data (example)
    sql = "INSERT INTO employees (name, salary) VALUES (%s,
%s)"
    val = ("John Doe", 50000)
    cursor.execute(sql, val)

    # Commit changes
    conn.commit()

    print(cursor.rowcount, "record inserted.")

except mysql.connector.Error as err:
    print(f"Error: {err}")
finally:
    if conn.is_connected():
        cursor.close()
        conn.close()
        print("MySQL connection is closed.")
```

- **Key Differences**

Feature	SQLite3	MySQL
Type	Serverless (Embedded)	Client-Server
Setup	No external server needed; uses a file	Requires a running server instance
Driver	Built-in Python module	Requires external library (mysql-connector-python)
Scalability	Good for local/small scale	Excellent for enterprise/large scale web apps
Concurrency	Generally only one writer at a time	Multiple concurrent users and writers

Q-2) Creating and executing SQL queries from Python using these connectors.

- SQLite3 is a built-in Python library used to interact with SQLite databases, while PyMySQL is a pure-Python library for connecting to MySQL or MariaDB databases. Both follow similar patterns for connecting, creating a cursor, executing queries, and committing changes.

- **Essential Steps for Both Connectors**

- The overall process involves four main stages:

- (1) **Connect** to the database (or create one if it doesn't exist).
- (2) **Create a cursor object** to interact with the database.
- (3) **Execute SQL queries** using the cursor.
- (4) **Commit changes** (for data modification) and **close** the connection.

(1) Using the sqlite3 Connector

- The sqlite3 module is part of Python's standard library and is ideal for working with local, disk-based or in-memory databases.

- **Example: Connect, Create Table, and Insert Data**

```
import sqlite3
```

```
# 1. Connect to the database (or create it)
conn = sqlite3.connect('example.db')
```

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

```
# 3. Execute SQL queries
# Create table
cursor.execute("""
CREATE TABLE IF NOT EXISTS stocks (
    date text,
    trans text,
    symbol text,
    qty real,
    price real
)
```

```
"")
```

```
# Insert a row of data
```

```
cursor.execute("INSERT INTO stocks VALUES ('2026-01-01','BUY','IBM',100,35.36)")
```

```
# 4. Commit changes and close connection
```

```
conn.commit()
```

```
conn.close()
```

Selecting and Fetching Data

```
import sqlite3
```

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

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

```
# Execute a SELECT query
```

```
cursor.execute("SELECT * FROM stocks ORDER BY price")
```

```
# Fetch the results
```

```
print("All results:")
```

```
results = cursor.fetchall()
```

```
for row in results:
```

```
    print(row)
```

```
# Close connection
```

```
conn.close()
```

(2) Using the PyMySQL Connector

- PyMySQL is used to connect to a remote MySQL server. You must first install it using pip: `pip install PyMySQL`.

- **Example: Connect, Create Table, and Insert Data**

- You will need the appropriate host, user, password, and database name for your MySQL server.

```
import pymysql
```

```
# 1. Connect to the database
```

```
try:
```

```
    conn = pymysql.connect(  
        host='localhost',  
        user='your_user',  
        password='your_password',  
        database='your_database_name'  
    )
```

```
# 2. Create a cursor object
```

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

```
# 3. Execute SQL queries
```

```
# Create table (Syntax might differ slightly from SQLite)
```

```
cursor.execute("""
```

```
CREATE TABLE IF NOT EXISTS employees (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    name VARCHAR(255),  
    salary DECIMAL(10, 2)  
)  
""")
```

```
# Insert a row of data
```

```

insert_query = "INSERT INTO employees (name, salary) VALUES
(%s, %s)"
cursor.execute(insert_query, ('Alice', 60000.00))

# 4. Commit changes and close connection
conn.commit()
print("Data inserted successfully")

except pymysql.Error as e:
    print(f"Error connecting to MySQL: {e}")

finally:
    if conn and conn.open:
        cursor.close()
        conn.close()

```

Selecting and Fetching Data with PyMySQL

```

import pymysql

# Connection details...
conn = pymysql.connect(
    host='localhost',
    user='your_user',
    password='your_password',
    database='your_database_name'
)
cursor = conn.cursor()

# Execute a SELECT query
cursor.execute("SELECT name, salary FROM employees WHERE salary
> %s", (50000,))

# Fetch the results

```

```
print("Employees earning more than 50k:")
results = cursor.fetchall()
for row in results:
    print(f"Name: {row[0]}, Salary: {row[1]}")

# Close connection
cursor.close()
conn.close()
```

(10) Search and Match Functions

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

- The `re` module in Python provides powerful tools for pattern matching using regular expressions. The functions `re.search()` and `re.match()` are two fundamental components, but they differ in how they approach a string for a match.

- **re.match()**

- The `re.match()` function checks for a match only at the beginning of the string. If the pattern is not found at the very start (including after a newline if multiline mode isn't explicitly used), the function returns `None`.

- **Key Characteristics:**

Start-anchored: It implicitly behaves as if the pattern is preceded by a start-of-string anchor (^).

Returns a Match Object or None: If successful, it returns a match object; otherwise, it returns `None`.

- **Example:**

```
import re
```

```
line = "The quick brown fox"
```

```
# This will find a match because "The" is at the start
```

```
match1 = re.match(r"The", line)
if match1:
    print(f"Match found by re.match(): {match1.group()}") # Output:
Match found by re.match(): The

# This will not find a match because "quick" is not at the start
match2 = re.match(r"quick", line)
if match2:
    print(f"Match found by re.match(): {match2.group()}")
else:
    print("No match found by re.match() for 'quick'") # Output: No
match found by re.match() for 'quick'
```

- **re.search()**

- The re.search() function scans the **entire string** from left to right, looking for the first location where the regular expression pattern produces a match .

- **Key Characteristics:**

Full string scan: It checks all parts of the string for the pattern.

Returns a Match Object or None: Similar to re.match(), it returns a match object on success or None otherwise.

Stops after the first match: It returns the first match it finds and does not continue scanning for subsequent matches.

- **Example:**

```
import re
```

```
line = "The quick brown fox"
```

```
# This will find a match because "quick" is in the string
```

```
match1 = re.search(r"quick", line)
```

```
if match1:
```

```
    print(f"Match found by re.search(): {match1.group()}") # Output:  
Match found by re.search(): quick
```

```
# This will also find a match
```

```
match2 = re.search(r"fox", line)
```

```
if match2:
```

```
    print(f"Match found by re.search(): {match2.group()}") # Output:  
Match found by re.search(): fox
```

Q-2) Difference between search and match.

Feature	re.match()	re.search()
Search Location	Only checks for a match at the beginning (index 0) of the string.	Scans the entire string for the first match.
Use Case	Ideal for input validation, such as verifying if a string starts with a specific format (e.g., a phone number's country code).	Ideal for finding patterns or keywords that may appear anywhere within a larger body of text, such as log files or documents.
Performance	Generally faster as it stops immediately if the beginning doesn't match the pattern.	Slightly slower for very large strings as it may need to check every position.
Return Value	Returns a match object if the pattern is found at the start; otherwise, returns None.	Returns the first match object it finds anywhere in the string; otherwise, returns None.