Theory Exercise

Printing on Screen

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

A. Introduction to the print() Function in Python

The **print() function** in Python is one of the most commonly used built-in functions.
It is used to **display information or output** to the console (screen).

- **Basic Syntax**

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

- **Simplest Example**

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

Output:

```
Hello, World!
```

- **Printing Multiple Values**

You can print multiple items by separating them with commas:

```
print("My name is", "Zaid", "Pathan")
```

Output:

```
My name is Zaid Pathan
```

Here, each value is separated by a **space** (default behavior).

- **Using sep (Separator) Parameter**

The sep parameter defines how multiple values are separated.

```
print("2025", "11", "03", sep="-")
```

Output:

```
2025-11-03
```

- **Using end Parameter**

The end parameter defines what to print **at the end** of the statement (default is newline \n).

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

Output:

```
Hello World!
```

- **Printing Variables**

```
name = "Alice"
age = 25
print("Name:", name, "Age:", age)
```

Output:

```
Name: Alice Age: 25
```

- **Using f-Strings (Formatted Print)**

A modern and clean way to print variables:

```
name = "Bob"
age = 30
print(f"My name is {name} and I am {age} years old.")
```

Output:

```
My name is Bob and I am 30 years old.
```

- **Printing Special Characters**

```
print("Hello\nWorld")   # newline
print("Hello\tWorld")   # tab space
```

Output:

```
Hello
World
Hello    World
```

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

A. Formatting Outputs in Python using f-strings and format()

Formatting output is important when you want to display data **neatly and clearly**, especially in reports, tables, or messages.
Python provides two main ways to format strings:

🔹 1. Using f-Strings (Formatted String Literals)

Introduced in **Python 3.6**, **f-strings** are the **simplest and most powerful** way to embed variables and expressions inside strings.

✅ **Basic Example**

```
name = "Alice"
age = 25
print(f"My name is {name} and I am {age} years old.")
```

Output:

```
My name is Alice and I am 25 years old.
```

✅ **Using Expressions Inside f-Strings**

You can perform **calculations** directly:

```
a = 10
b = 5
print(f"The sum of {a} and {b} is {a + b}.")
```

Output:

```
The sum of 10 and 5 is 15.
```

✅ **Formatting Numbers**

You can format **decimals**, **percentages**, or **commas** for large numbers.

```
pi = 3.14159265
print(f"Pi rounded to 2 decimal places: {pi:.2f}")


salary = 2500000
print(f"Formatted salary: ₹{salary:,}")
```

Output:

```
Pi rounded to 2 decimal places: 3.14
Formatted salary: ₹2,500,000
```

✅ Aligning Text and Numbers

You can align text or numbers using special formatting codes:

Symbol	Meaning
<	Left-align
>	Right-align
^	Center-align

```
name = "Python"
print(f"|{name:<10}|")  # Left align
print(f"|{name:>10}|")  # Right align
print(f"|{name:^10}|")  # Center align
```

Output:

```
|Python    |
|    Python|
|  Python  |
```

🔹 2. Using the format() Method

Before f-strings, Python used the .format() method to achieve the same result.

✅ Basic Example

```
name = "Bob"
age = 30
print("My name is {} and I am {} years old.".format(name, age))
```

Output:

```
My name is Bob and I am 30 years old.
```

✅ Using Positional and Keyword Arguments

```
print("I love {0} and {1}.".format("Python", "AI"))
print("I love {lang} and {tech}.".format(lang="Python", tech="AI"))
```

Output:

```
I love Python and AI.
I love Python and AI.
```

✅ **Formatting Numbers**

```
pi = 3.14159265
print("Pi rounded to 2 decimal places: {:.2f}".format(pi))

salary = 2500000
print("Formatted salary: ₹{:,.0f}".format(salary))
```

Output:

```
Pi rounded to 2 decimal places: 3.14
Formatted salary: ₹2,500,000
```

✅ **Aligning Text**

```
print("|{:<10}|{:>10}|{:^10}|".format("Left", "Right", "Center"))
```

Output:

```
|Left      |     Right|  Center  |
```

Reading Data from Keyboard

Q3. Common list operations: concatenation, repetition, membership.

A. Using the input() Function in Python

The **input() function** is used to **take input from the user** through the **keyboard**.
It allows your program to **interact** with the user by reading data typed in during execution.

🔹 **Basic Syntax**

```
variable = input(prompt)
```

- **prompt** – (optional) A message shown to the user before taking input.
- **variable** – The data entered by the user is stored in this variable.
- **By default**, the input is taken as a **string (text)**.

✅ **Example 1: Basic Input**

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

Output:

```
Enter your name: Alice
Hello, Alice
```

✅ **Example 2: Reading Numeric Input**

All input values are **strings**, so you must convert them to numbers using:

- int() → for integers
- float() → for decimal numbers

```
age = int(input("Enter your age: "))
print("You will be", age + 1, "next year.")
```

Output:

```
Enter your age: 20
You will be 21 next year.
```

✅ **Example 3: Taking Multiple Inputs**

You can read multiple values in a single line using split():

```
a, b = input("Enter two numbers separated by space: ").split()
print("You entered:", a, "and", b)
```

Output:

```
Enter two numbers separated by space: 5 10
You entered: 5 and 10
```

➞ To convert them into numbers:

```
a, b = map(int, input("Enter two numbers: ").split())
print("Sum =", a + b)
```

Output:

```
Enter two numbers: 5 10
Sum = 15
```

✅ **Example 4: Input and f-Strings**

You can combine input() with **formatted output**:

```
city = input("Enter your city: ")
print(f"You live in {city}.")
```

Output:

```
Enter your city: Mumbai
You live in Mumbai.
```

✅ **Example 5: Using input() for Calculations**

```
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))
print(f"The average is {(num1 + num2) / 2}")
```

Output:

```
Enter first number: 10
Enter second number: 20
The average is 15.0
```

Q4. Understanding list methods like append(), insert(), remove(), pop().

A. Converting User Input into Different Data Types in Python

When you use the **input()** function in Python,
➡️ **everything entered by the user is read as a string (str)** — even numbers.

So, if you want to perform **mathematical operations**, you must **convert** the input to another data type like int or float.

🔹 **1. Input Always Returns a String**

Example:

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

```
print(x)
print(type(x))
```

Output:

```
Enter a number: 10
10
<class 'str'>
```

✅ The data type is **string**, not integer.

🔹 2. Converting String Input to Integer (int)

If you need to perform arithmetic, convert the input using int():

```
num = int(input("Enter an integer: "))
print("You entered:", num)
print("After adding 5:", num + 5)
```

Output:

```
Enter an integer: 10
You entered: 10
After adding 5: 15
```

🔹 3. Converting String Input to Float (float)

For decimal numbers, use float():

```
price = float(input("Enter the price: "))
print("Price with tax:", price * 1.18)
```

Output:

```
Enter the price: 100.5
Price with tax: 118.59
```

🔹 4. Converting to Other Data Types

Data Type	Conversion Function	Example	Output Type
Integer	int()	int("10")	<class 'int'>
Float	float()	float("3.14")	<class 'float'>

Data Type	Conversion Function	Example	Output Type
String	str()	str(25)	<class 'str'>
Boolean	bool()	bool("True")	<class 'bool'>

🔹 **5. Taking and Converting Multiple Inputs**

You can read multiple values and convert them at once using map():

```
a, b = map(int, input("Enter two integers: ").split())
print("Sum:", a + b)
```

Output:

```
Enter two integers: 4 6
Sum: 10
```

➡️ Here, map(int, ...) applies the int() function to **each** value entered.

🔹 **6. Example: Combining Input and Conversion**

```
name = input("Enter your name: ")
age = int(input("Enter your age: "))
height = float(input("Enter your height in meters: "))


print(f"Hello {name}, you are {age} years old and {height} meters tall.")
```

Output:

```
Enter your name: Zaid
Enter your age: 21
Enter your height in meters: 1.75
Hello Zaid, you are 21 years old and 1.75 meters tall.
```

Opening and Closing Files

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

A. Opening Files in Different Modes in Python

Python provides a simple way to **work with files** — reading from and writing to them — using the built-in **open()** function.

The **file mode** you choose determines **how** the file is opened (read, write, append, etc.).

🔹 1. The open() Function — Basic Syntax

file = open(filename, mode)

Parameter	Description
filename	Name (and path) of the file to open
mode	Defines the purpose (read, write, etc.)

After working with a file, always **close it** using:

file.close()

🔹 2. Common File Modes

Mode	Meaning	File must exist?	Description
'r'	Read	✅ Yes	Opens file for reading (default mode).
'w'	Write	❌ No	Creates a new file or **overwrites** an existing one.
'a'	Append	❌ No	Opens file for writing, appends new data at the end.
'r+'	Read & Write	✅ Yes	Reads and writes (does not overwrite).
'w+'	Read & Write	❌ No	Creates a new file (overwrites if exists).

🔹 3. Mode 'r' – Read Mode

Used to **read data** from an existing file.

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

➡️ **Notes:**

- The file **must exist** or you'll get an error (FileNotFoundError).
- Use read(), readline(), or readlines() to get content.

🔹 4. Mode 'w' – Write Mode

Used to **write new data**.
If the file exists, its content will be **erased**.

```
file = open("example.txt", "w")
file.write("Hello, Python!\n")
file.write("This is a new file.")
file.close()
```

➡️ **Notes:**

- If the file doesn’t exist, Python **creates it**.
- Existing content is **deleted** before writing.

🔹 **5. Mode 'a' – Append Mode**

Used to **add new data** to an existing file **without deleting old content**.

```
file = open("example.txt", "a")
file.write("\nAppending new line at the end.")
file.close()
```

➡️ **Notes:**

- Adds text at the **end** of the file.
- Creates the file if it doesn’t exist.

🔹 **6. Mode 'r+' – Read and Write Mode**

Used to **read and modify** an existing file.

```
file = open("example.txt", "r+")
content = file.read()
print("Before:", content)
file.seek(0)  # Move to start
file.write("Modified text\n")
file.close()
```

➡️ **Notes:**

- The file **must exist**.
- Can both **read** and **write**, but writing starts from the beginning unless you use seek().

🔹 **7. Mode 'w+' – Write and Read Mode**

Used to **create a file**, write to it, and **read back** content.

```
file = open("sample.txt", "w+")
file.write("Hello World!")
file.seek(0)  # Move back to start to read
print(file.read())
file.close()
```

➡️ **Notes:**

- Existing file is **overwritten**.
- If file doesn't exist, Python **creates** it.

🔹 **8. Using with Statement (Recommended)**

Best practice: Use with to handle files — it automatically **closes** the file.

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

✅ **Advantage:**
You don't need to call file.close() — it's done automatically, even if an error occurs.

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

A. Using the open() Function to Create and Access Files in Python

The **open()** function is the main way to **create**, **read**, and **write** files in Python.
It allows your program to interact with files stored on your computer — to save data permanently or retrieve it later.

🔹 **Basic Syntax**

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

Parameter	Description
filename	The name (and path) of the file you want to open.
mode	Defines what you want to do — read, write, append, etc.

🔹 **Common File Modes**

Mode	Meaning	Description
'r'	Read	Opens an existing file for reading only.
'w'	Write	Creates a new file or overwrites an existing file.
'a'	Append	Opens a file to add data at the end.
'r+'	Read + Write	Opens an existing file to read and modify.
'w+'	Write + Read	Creates a new file and allows reading/writing.

🧩 **1. Creating a New File using open()**

You can **create** a new file by opening it in **write ('w')** or **append ('a')** mode.
If the file does not exist, Python automatically creates it.

```
file = open("myfile.txt", "w")
file.write("Hello, this is my first file in Python!")
file.close()
```

✅ **Result:**

- If myfile.txt doesn't exist → it will be **created**.
- If it exists → its old content will be **erased** and replaced.

🧩 **2. Writing Multiple Lines to a File**

```
file = open("notes.txt", "w")
file.write("Python File Handling\n")
file.write("We are writing multiple lines.\n")
file.close()
```

✅ After this, a file named notes.txt will be created in the same directory.

🧩 **3. Reading a File**

You can **read** a file's content using the **read()**, **readline()**, or **readlines()** methods.

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

```
file.close()
```

Output:

```
Python File Handling
We are writing multiple lines.
```

🧩 4. Appending Data to an Existing File

To **add** content without deleting the existing text, use 'a' mode.

```
file = open("notes.txt", "a")
file.write("This line is added later.\n")
file.close()
```

✅ Opens the file and **adds new text at the end**.

🧩 5. Reading and Writing Together

Use 'r+' (read + write) or 'w+' (write + read).

```
file = open("data.txt", "w+")
file.write("Python makes file handling easy!")
file.seek(0)  # Move the cursor to the beginning
print(file.read())
file.close()
```

Output:

Python makes file handling easy!

🧩 6. Best Practice — Using with Statement

The **with** statement automatically closes the file for you.

```
with open("example.txt", "w") as file:
    file.write("Using 'with' is safer and cleaner.")

with open("example.txt", "r") as file:
    print(file.read())
```

✅ You don’t need to call file.close() manually.

🧩 7. Checking if a File Exists (Optional)

You can check before accessing a file to avoid errors:

```
import os


if os.path.exists("example.txt"):
   print("File exists!")
else:
   print("File not found, creating one...")
   open("example.txt", "w").close()
```

Q7. Closing files using close().

A. Closing Files Using close() in Python

When you work with files in Python (for reading or writing), it's very important to **close the file** after you're done.
This ensures that all data is properly saved and that system resources are released.

🔹 1. Why Closing Files is Important

When you open a file using the open() function, Python reserves **system resources** (like memory and file handles) for that file.

If you forget to close it:

- 🧠 Memory may remain in use.
- 💾 Data might not be written completely to disk (especially in write or append mode).
- ⚠️ You could face errors if too many files are open.

So, **always close the file** after finishing your operations.

🔹 2. Basic Syntax

```
file_object.close()
```

🔹 3. Example: Opening and Closing a File

```
file = open("example.txt", "w")  # Open file in write mode
file.write("Hello, World!")      # Write some data
```

file.close() # Close the file

✅ This ensures that:

- The data "Hello, World!" is saved to the file.
- The file is properly released by the system.

🔹 **4. Reading and Then Closing**

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

✅ Always use file.close() after reading to release the resource.

🔹 **5. Checking If a File is Closed**

You can check the file's status using the .closed attribute:

```
file = open("example.txt", "r")
print(file.closed)  # False
file.close()
print(file.closed)  # True
```

Output:

```
False
True
```

🔹 **6. Using with Statement (Best Practice)**

Instead of manually closing the file, use a **with block** — Python will automatically close the file once the block ends, even if an error occurs.

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

# File is automatically closed here
print(file.closed)  # True
```

✅ **Advantages of using with:**

- Automatically closes the file.
- Prevents data loss.
- Makes code cleaner and safer.

Reading and Writing Files

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

A. Reading from a File in Python using read(), readline(), and readlines()

When working with files in Python, you often need to **read data** from them.
Python provides three main methods to do this:
read() readline() readlines()

Let's understand each in detail

🔹 **1. Opening a File for Reading**

To read from a file, first open it in **read mode ('r')**:

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

If the file does not exist, Python will raise a **FileNotFoundError**.

🔹 **2. The read() Method — Read Entire File**

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

✅ **Example:**

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

If example.txt contains:

```
Python is fun!
File handling is easy.
```

Output:

```
Python is fun!
File handling is easy.
```

🔸 **Read a specific number of characters**

```
file = open("example.txt", "r")
print(file.read(10))   # Reads first 10 characters
file.close()
```

Output:

```
Python is
```

🔹 **3. The readline() Method — Read One Line at a Time**

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

✅ **Example:**

```
file = open("example.txt", "r")
line1 = file.readline()
line2 = file.readline()
print("Line 1:", line1)
print("Line 2:", line2)
file.close()
```

Output:

```
Line 1: Python is fun!
Line 2: File handling is easy.
```

🔸 **Using a loop with readline()**

```
file = open("example.txt", "r")
line = file.readline()
while line:
    print(line.strip())   # remove newline character
    line = file.readline()
file.close()
```

🔹 **4. The readlines() Method — Read All Lines into a List**

The **readlines()** method reads **all lines at once** and returns a **list** where each line is a separate string.

✅ **Example:**

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

Output:

```
['Python is fun!\n', 'File handling is easy.\n']
```

You can loop through this list:

```
for line in lines:
  print(line.strip())
```

Output:

```
Python is fun!
File handling is easy.
```

🔹 **5. Using with Statement (Best Practice)**

Instead of manually closing the file, use the **with** statement.
It automatically closes the file after the block ends.

✅ **Example:**

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

You can also use it with loops:

```
with open("example.txt", "r") as file:
  for line in file:
    print(line.strip())
```

Q9. Creating and accessing elements in a tuple.

A. Writing to a File in Python using write() and writelines()

In Python, you can **store data in files** by writing content using two main methods:
write() and writelines()

Let's understand each with examples.

🔹 1. Opening a File for Writing

To write data to a file, you must open it using the open() function with a **write mode**.

Mode	Description
'w'	Write mode – overwrites the file if it exists, or creates a new one
'a'	Append mode – adds new content at the end of the file
'w+'	Write and read mode

Example:

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

🔹 2. The write() Method

The write() method writes **a single string** to the file.

✅ **Example:**

```
file = open("example.txt", "w")
file.write("Python is fun!\n")
file.write("File handling makes programming easier.\n")
file.close()
```

This creates or overwrites a file named **example.txt** with:

```
Python is fun!
File handling makes programming easier.
```

🔸 Writing user input to a file

```
text = input("Enter some text: ")
file = open("user_data.txt", "w")
file.write(text)
file.close()
```

🔸 Appending content to an existing file

```
file = open("example.txt", "a")
file.write("This line is added later.\n")
```

```
file.close()
```

🔹 3. The writelines() Method

The writelines() method writes **a list of strings** to the file.
Each string is written exactly as given (so you must include \n for new lines).

✅ **Example:**

```
lines = [
    "Python is easy to learn.\n",
    "It is widely used in data science.\n",
    "File handling is important.\n"
]

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

Content of example.txt:

```
Python is easy to learn.
It is widely used in data science.
File handling is important.
```

⚠️ **Note:**

- write() → for single string
- writelines() → for list of strings
- Always include \n manually when you want new lines.

🔹 4. Using with Statement (Best Practice)

You can use the with statement to automatically close the file.

✅ **Example:**

```
with open("example.txt", "w") as file:
    file.write("Using 'with' ensures the file is closed automatically.\n")
    file.writelines(["Line 1\n", "Line 2\n", "Line 3\n"])
```

After this block, the file is automatically closed — no need for file.close().

Exception Handling

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

A. Introduction to Exceptions and Handling Them in Python (try, except, finally)

When writing Python programs, **errors can occur** during execution — such as dividing by zero, accessing a missing file, or converting invalid input.
These runtime errors are called **exceptions**.

To prevent your program from crashing when these occur, Python provides a way to **handle exceptions** using try, except, and finally blocks.

🔹 1. What is an Exception?

An **exception** is an event that occurs during program execution and disrupts the normal flow of the program.

Example:

```
x = 10
y = 0
print(x / y)   # ❌ ZeroDivisionError
```

Output:

```
ZeroDivisionError: division by zero
```

The program stops execution because of this unhandled exception.

🔹 2. Handling Exceptions with try and except

The try block contains code that may cause an exception.
The except block contains code that handles the exception.

✅ **Example:**

```
try:
    x = int(input("Enter a number: "))
    print(10 / x)
except ZeroDivisionError:
    print("Error: You cannot divide by zero.")
```

Output 1 (if user enters 2):

5.0

Output 2 (if user enters 0):

Error: You cannot divide by zero.

🔹 3. Handling Multiple Exceptions

You can handle **different types of errors** separately by using multiple except blocks.

Example:

```
try:
    a = int(input("Enter a number: "))
    b = int(input("Enter another number: "))
    print(a / b)
except ZeroDivisionError:
    print("❌ Cannot divide by zero!")
except ValueError:
    print("❌ Please enter valid numbers only!")
```

🔹 4. Catching All Exceptions

If you don't know what error might occur, you can use a general except: block.

Example:

```
try:
    num = int(input("Enter a number: "))
    print(100 / num)
except:
    print("An error occurred.")
```

⚠️ *However, this is not recommended in real programs* because it hides the specific error type.

🔹 5. Using finally Block

The finally block is **always executed**, whether an exception occurs or not.
It's often used for **cleanup actions** like closing files or releasing resources.

✅ **Example:**

```
try:
```

```
    file = open("data.txt", "r")
    content = file.read()
    print(content)
except FileNotFoundError:
    print("File not found.")
finally:
    print("Execution completed.")
```

Output (if file doesn't exist):

```
File not found.
Execution completed.
```

✅ finally runs **even if an exception occurs**.

🔹 **6. Using else with try-except**

If no exception occurs, the else block will run.

Example:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("You cannot divide by zero.")
else:
    print("Division successful! Result =", result)
finally:
    print("Program finished.")
```

Output (if user enters 2):

```
Division successful! Result = 5.0
Program finished.
```

Q11. Understanding multiple exceptions and custom exceptions.

A. Understanding Multiple Exceptions and Custom Exceptions in Python

When writing programs, you may encounter **different kinds of errors** — some caused by user input, others by missing files, etc.
Python allows you to handle **multiple exceptions** gracefully and even create your **own (custom)** exceptions.

Let's explore both in detail 👇

🔹 1. Handling Multiple Exceptions

When you expect more than one type of error, you can:

- Use **multiple except blocks** to handle each error separately, or
- Use a **single except** block to handle several exceptions together.

✅ Example 1: Multiple except Blocks

```
try:
    a = int(input("Enter a number: "))
    b = int(input("Enter another number: "))
    result = a / b
    print("Result:", result)
except ZeroDivisionError:
    print("❌ You cannot divide by zero.")
except ValueError:
    print("❌ Please enter valid numeric values.")
except Exception as e:
    print("⚠️ Some other error occurred:", e)
```

Explanation:

- ZeroDivisionError → if the user enters b = 0
- ValueError → if the user enters non-numeric input
- The general Exception block catches any other unexpected error.

✅ Example 2: Handling Multiple Exceptions Together

You can group multiple exceptions into a **tuple** inside one except block.

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

```
    b = int(input("Enter another number: "))
    print(a / b)
except (ZeroDivisionError, ValueError):
    print("⚠️ Invalid input or division by zero!")
```

✅ **Example 3: Catching All Exceptions (Not Recommended)**

```
try:
    x = int("abc")
except Exception as e:
    print("An error occurred:", e)
```

⚠️ **Note:** Catching all exceptions hides the actual cause.
Use this **only for debugging or logging**.

🔹 **2. Creating Custom Exceptions**

Sometimes, you may want to raise your **own exceptions** to handle specific cases in your program.
You can create a custom exception by **inheriting from the built-in Exception class**.

✅ **Example 1: Simple Custom Exception**

```
class NegativeNumberError(Exception):
    """Raised when a negative number is entered."""
    pass


try:
    num = int(input("Enter a positive number: "))
    if num < 0:
        raise NegativeNumberError("Negative numbers are not allowed.")
    print("You entered:", num)
except NegativeNumberError as e:
    print("Custom Exception:", e)
```

Output (if user enters -5):

```
Custom Exception: Negative numbers are not allowed.
```

✅ **Example 2: Custom Exception with Multiple Conditions**

```
class InvalidAgeError(Exception):
    """Raised when the entered age is invalid."""
    pass

try:
    age = int(input("Enter your age: "))
    if age < 0:
        raise InvalidAgeError("Age cannot be negative.")
    elif age < 18:
        raise InvalidAgeError("You must be 18 or older.")
    else:
        print("Age accepted!")
except InvalidAgeError as e:
    print("Error:", e)
```

Output (if user enters 15):

```
Error: You must be 18 or older.
```

✅ **Example 3: Using try-except-finally with Custom Exceptions**

```
class InsufficientBalanceError(Exception):
    pass

try:
    balance = 500
    withdraw = int(input("Enter amount to withdraw: "))

    if withdraw > balance:
        raise InsufficientBalanceError("Not enough balance!")

    balance -= withdraw
```

```
    print("Withdrawal successful. Remaining balance:", balance)

except InsufficientBalanceError as e:
    print("Transaction failed:", e)
finally:
    print("Transaction process completed.")
```

Class and Object (OOP Concepts)

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

A. Understanding Classes, Objects, Attributes, and Methods in Python (OOP Basics)

Python is an **object-oriented programming (OOP)** language — this means it allows you to organize code around **objects** rather than just functions and logic.
Let's break down the key concepts

🔹 1. What is a Class?

A **class** is a **blueprint** or **template** for creating objects.
It defines the **attributes (data)** and **methods (functions)** that the objects created from it will have.

Think of a **class** as a "plan" for building something.

✅ **Example:**

```
class Car:
    # attributes (variables)
    color = "Red"
    brand = "Toyota"
```

Here, Car is a **class** with two **attributes**: color and brand.

🔹 2. What is an Object?

An **object** is an **instance** of a class — created using the class blueprint.

Each object has its **own data** (attribute values) and can use the **methods** defined in the class.

✅ **Example:**

```
class Car:
    color = "Red"
    brand = "Toyota"

# Create objects
car1 = Car()
car2 = Car()

print(car1.color)  # Output: Red
print(car2.brand)  # Output: Toyota
```

✅ Both car1 and car2 are **objects** (instances) of the Car class.

🔹 **3. Attributes (Data Members)**

Attributes are **variables that store data** inside a class or an object.
There are two types:

- **Class attributes** → shared by all objects
- **Instance attributes** → unique to each object

✅ **Example: Class vs. Instance Attributes**

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

    # Constructor to create instance attributes
    def __init__(self, brand, color):
        self.brand = brand
        self.color = color

# Create objects
```

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

print(car1.brand, car1.color, car1.wheels)
print(car2.brand, car2.color, car2.wheels)
```

Output:

```
Toyota Red 4
BMW Black 4
```

✅ wheels → class attribute
✅ brand and color → instance attributes (different for each object)

🔹 4. Methods (Functions inside Classes)

Methods are **functions defined inside a class** that describe the behavior of the objects.

They always take self as the **first parameter**, which refers to the **current object**.

✅ Example: Method Definition

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

    def start(self):
        print(f"{self.brand} is starting!")

# Create object
my_car = Car("Tesla", "White")
my_car.start()
```

Output:

```
Tesla is starting!
```

✅ __init__() is a **special method** (called a constructor).
It automatically runs when you create an object and initializes its attributes.

🔹 5. The __init__() Constructor Method

The **constructor** initializes object attributes at creation time.

Example:

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

s1 = Student("Alice", 20)
print(s1.name, s1.age)
```

Output:

```
Alice 20
```

🔹 6. Accessing and Modifying Attributes

You can access or modify attributes using the **dot (.) operator**.

Example:

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

car = Car("Honda", "Blue")

# Access attributes
print(car.color)      # Output: Blue
```

```
# Modify attributes
car.color = "Green"
print(car.color)      # Output: Green
```

🔹 **7. Example: Complete Class**

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

    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}")

# Create objects
s1 = Student("Alice", 20)
s2 = Student("Bob", 22)

s1.display_info()
s2.display_info()
```

Output:

```
Name: Alice, Age: 20
Name: Bob, Age: 22
```

Q13. Difference between local and global variables.

A. Difference Between Local and Global Variables in Python

In Python, **variables** have a *scope* — meaning, where in the program they can be **accessed** or **modified**.
Understanding the difference between **local** and **global** variables is very important when writing functions and programs.

Let’s break it down clearly

🔹 1. What is a Local Variable?

A **local variable** is a variable that is **declared inside a function** and is **accessible only within that function**.

It exists **temporarily** — once the function finishes, the variable is destroyed.

✅ **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
```

x is **local** to my_function(), and cannot be used outside it.

🔹 2. What is a Global Variable?

A **global variable** is declared **outside all functions** and is **accessible anywhere** in the program — both inside and outside functions.

✅ **Example:**

```
x = 100  # Global variable

def show():
    print("Inside function:", x)

show()
print("Outside function:", x)
```

Output:

```
Inside function: 100
Outside function: 100
```

✅ x is a **global variable**, so it can be accessed anywhere.

🔹 3. Local vs Global Variable with Same Name

If you define a **local variable with the same name** as a global one,
Python will **use the local variable inside the function**, leaving the global variable unchanged.

✅ **Example:**

```
x = 50  # Global variable


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


test()
print("Outside function:", x)
```

Output:

```
Inside function: 10
Outside function: 50
```

The local x *overrides* the global x inside the function.

🔹 4. Accessing and Modifying Global Variables inside a Function

If you want to **modify a global variable** inside a function, you must use the **global keyword**.

✅ **Example:**

```
x = 5  # Global variable


def update():
    global x
    x = x + 10  # Modify global variable
    print("Inside function:", x)


update()
print("Outside function:", x)
```

Output:

Inside function: 15

Outside function: 15

Without global, Python would assume x is **local** and show an error.

🔹 **5. Nested Functions (Local Scope Inside Local Scope)**

Python supports **nested functions** — a function inside another function.
A variable in the outer function is **not global**, but can be accessed using nonlocal.

✅ **Example (Bonus Concept):**

```
def outer():
    x = "outer value"

    def inner():
        nonlocal x
        x = "inner value"
        print("Inside inner():", x)

    inner()
    print("Inside outer():", x)

outer()
```

Output:

Inside inner(): inner value

Inside outer(): inner value

Here, nonlocal allows the inner function to modify x from the outer (non-global) scope.

Inheritance

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

A. Types of Inheritance in Python

Inheritance is one of the most powerful concepts in **Object-Oriented Programming (OOP)**.
It allows a class (**child class**) to **inherit properties and methods** from another class (**parent class**).

This promotes **code reusability**, **readability**, and **extensibility**.

Let’s explore all **five types of inheritance** in Python 👇

🔹 1. Single Inheritance

👉 **Definition:**
When a **child class inherits** from **only one parent class**.

✅ **Example:**

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


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


# Create object
obj = Child()
obj.display()  # Inherited from Parent
obj.show()     # From Child
```

Output:

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

🧠 **Explanation:**
Child inherits display() from Parent, so it can use both methods.

🔹 2. Multilevel Inheritance

👉 **Definition:**
When a class is derived from another derived class — forming a **chain of inheritance**.

✅ **Example:**

```
class GrandParent:
    def grandparent_method(self):
        print("This is the Grandparent class")
```

```
class Parent(GrandParent):
    def parent_method(self):
        print("This is the Parent class")

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

# Create object
obj = Child()
obj.grandparent_method()
obj.parent_method()
obj.child_method()
```

Output:

```
This is the Grandparent class
This is the Parent class
This is the Child class
```

🧠 **Explanation:**
Child inherits from Parent, and Parent inherits from GrandParent.

🔹 **3. Multiple Inheritance**

👉 **Definition:**
When a class **inherits from more than one parent class**.

✅ **Example:**

```
class Father:
    def skill1(self):
        print("Father: driving")

class Mother:
    def skill2(self):
```

```
        print("Mother: cooking")


class Child(Father, Mother):
    def skill3(self):
        print("Child: painting")


# Create object
obj = Child()
obj.skill1()
obj.skill2()
obj.skill3()
```

Output:

```
Father: driving
Mother: cooking
Child: painting
```

🧠 **Explanation:**
Child inherits from both Father and Mother.
So it can access methods from both classes.

🔹 **4. Hierarchical Inheritance**

👉 **Definition:**
When **multiple child classes inherit from the same parent class**.

✅ **Example:**

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


class Child1(Parent):
    def show1(self):
        print("This is Child 1")


class Child2(Parent):
```

```
    def show2(self):
        print("This is Child 2")

# Create objects
obj1 = Child1()
obj2 = Child2()

obj1.display()
obj1.show1()

obj2.display()
obj2.show2()
```

Output:

```
This is the Parent class
This is Child 1
This is the Parent class
This is Child 2
```

🧠 **Explanation:**
Both Child1 and Child2 inherit display() from Parent.

🔹 **5. Hybrid Inheritance**

👉 **Definition:**
A **combination** of two or more types of inheritance (e.g., Multiple + Multilevel).
It forms a **complex hierarchy**.

✅ **Example:**

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

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

```
class C(A):
    def showC(self):
        print("Class C")


class D(B, C):   # Hybrid Inheritance
    def showD(self):
        print("Class D")


# Create object
obj = D()
obj.showA()
obj.showB()
obj.showC()
obj.showD()
```

Output:

```
Class A
Class B
Class C
Class D
```

🧠 **Explanation:**
Here, D inherits from both B and C (Multiple inheritance),
and B and C both inherit from A (Hierarchical inheritance).
Together → this is **Hybrid Inheritance**.

🔹 6. Method Resolution Order (MRO)

In **Multiple or Hybrid Inheritance**, if two parent classes have methods with the **same name**, Python uses **MRO (Method Resolution Order)** to decide which one to call first.

Example:

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

```
class B(A):
    def display(self):
        print("Class B")

class C(A):
    def display(self):
        print("Class C")

class D(B, C):
    pass

obj = D()
obj.display()

print(D.mro())  # Shows the search order
```

Output:

```
Class B
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

🧠 **Explanation:**
MRO says: Python will look for the method in this order → D → B → C → A.

Q15. Using the super() function to access properties of the parent class.

A. Using the super() Function in Python

In **Object-Oriented Programming (OOP)**, the super() function is used to **access methods and properties of the parent (base) class** from the **child (derived) class**.
It's a powerful and clean way to reuse parent class functionality **without explicitly naming the parent class**.

🔹 **1. What is super()?**

The **super()** function returns a **temporary object** of the **parent class**,
which allows you to **call its methods or access its attributes**.

✅ **Syntax:**

```
super().method_name()
```

🔹 **2. Basic Example**

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


class Child(Parent):
    def display(self):
        super().display()  # Call Parent's display()
        print("This is the Child class")


# Create object
obj = Child()
obj.display()
```

Output:

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

🧠 **Explanation:**

- The Child class **overrides** the display() method.
- Inside it, we use super().display() to call the **Parent class's version** first.

🔹 **3. Using super() with __init__() (Constructors)**

You can use super() to **initialize the parent class constructor** inside the child class constructor.

✅ **Example:**

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

```
class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)   # Call Parent constructor
        self.age = age
        print("Child constructor called")

# Create object
obj = Child("Zaid", 21)
print("Name:", obj.name)
print("Age:", obj.age)
```

Output:

```
Parent constructor called
Child constructor called
Name: Zaid
Age: 21
```

🧠 **Explanation:**

- The parent constructor initializes name.
- The child constructor calls it using super(), then adds its own age attribute.

🔹 **4. Use super() Instead of Parent Class Name**

You *could* write:

```
Parent.display(self)
```

But super() is **better** because:

✅ It automatically handles **multiple inheritance** (follows **MRO — Method Resolution Order**).
✅ It's **easier to maintain** — if the parent class name changes, no need to edit everywhere.
✅ It ensures **consistent initialization** of all parent classes.

🔹 **5. Example: Multiple Inheritance with super()**

```
class A:
    def __init__(self):
        print("A initialized")
```

```
        super().__init__()


class B:
    def __init__(self):
        print("B initialized")
        super().__init__()


class C(A, B):  # Multiple inheritance
    def __init__(self):
        print("C initialized")
        super().__init__()


# Create object
obj = C()
```

Output:

```
C initialized
A initialized
B initialized
```

🧠 **Explanation:**

- super() follows the **Method Resolution Order (MRO)**: C → A → B → object.
- This ensures that each parent class is initialized **only once** in the correct order.

🔹 **6. Accessing Parent Attributes Using super()**

```
class Parent:
    def __init__(self):
        self.value = "Parent Value"


class Child(Parent):
    def __init__(self):
        super().__init__()  # Initialize parent
        print("Accessing:", super().value)  # Access parent property (Not direct)
```

⚠️ **Note:**
super() is typically used to access **methods**, not attributes directly.
If you need the parent's attribute, initialize it using super().__init__() and then access it via self.value.

Method Overloading and Overriding

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

A. Method Overloading in Python

In Object-Oriented Programming, **method overloading** means having **multiple methods with the same name but different numbers or types of parameters** — so the method that gets executed depends on how it's called.

However, in **Python**, things work a little differently from languages like **Java** or **C++**.
Let's understand this clearly 👇

🔹 **1. What is Method Overloading?**

👉 **Definition:**
Method overloading allows a class to have **more than one method with the same name**, but **different arguments** (number or type).

✅ **Example (Conceptually):**

```
# Example like in Java/C++

def add(a, b)

def add(a, b, c)
```

In other languages, the compiler decides which add() to call based on parameters.
But in **Python**, this is **not directly supported**.

🔹 **2. Why Not Supported Directly in Python?**

Python does **not support true method overloading** because:

- Python functions can take **any type** of arguments.
- The **latest defined method** with the same name **overwrites the previous one**.

❌ **Example:**

```
class Example:

  def show(self, a):

    print("One argument:", a)
```

```
    def show(self, a, b):   # This overwrites the previous 'show'
        print("Two arguments:", a, b)


obj = Example()
obj.show(10, 20)  # Works fine
obj.show(10)      # ❌ Error: missing 1 required positional argument
```

Output:

Two arguments: 10 20

TypeError: show() missing 1 required positional argument: 'b'

🧠 **Explanation:**
The **second show()** replaced the first one.
So Python only remembers the **latest definition**.

🔹 **3. Simulating Method Overloading (Workaround)**

Although Python doesn't support it directly,
we can **simulate method overloading** using:

✅ **a) Default Arguments**

```
class Example:
    def show(self, a=None, b=None):
        if a is not None and b is not None:
            print("Two arguments:", a, b)
        elif a is not None:
            print("One argument:", a)
        else:
            print("No arguments")


obj = Example()
obj.show()
obj.show(10)
obj.show(10, 20)
```

Output:

No arguments

One argument: 10

Two arguments: 10 20

🧠 **Explanation:**
By using **default values (None)**, we can make one method behave differently based on how many arguments are passed.

✅ **b) Variable-Length Arguments (*args)**

You can accept **any number of arguments** and handle them accordingly.

```
class Example:
    def add(self, *args):
        total = 0
        for num in args:
            total += num
        print("Sum =", total)

obj = Example()
obj.add(10)
obj.add(10, 20)
obj.add(10, 20, 30)
```

Output:

Sum = 10

Sum = 30

Sum = 60

🧠 **Explanation:**
The *args allows the method to take **a variable number of arguments** — making it behave like an **overloaded** method.

✅ **c) Using Type Checking (isinstance())**

You can also check **argument types** to change behavior dynamically.

```
class Calculator:
    def multiply(self, a, b=None):
        if b is not None:
```

```
            print("Product =", a * b)
        elif isinstance(a, list):
            product = 1
            for num in a:
                product *= num
            print("Product of list =", product)
        else:
            print("Invalid input")

obj = Calculator()
obj.multiply(5, 3)
obj.multiply([2, 3, 4])
```

Output:

Product = 15

Product of list = 24

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

A. Method Overriding in Python

Method Overriding is a core concept in **Object-Oriented Programming (OOP)** that allows a **child class** to **redefine (override)** a **method** that already exists in its **parent class**.

It helps in **customizing or extending** the behavior of inherited methods.

🔹 **1. What is Method Overriding?**

👉 **Definition:**
When a **child class** defines a method **with the same name, parameters, and behavior** as one in the **parent class**,
the **child's method overrides** the **parent's method**.

This means:
➡️ When the method is called on a child object, the **child class version** is executed instead of the parent one.

✅ **Example: Basic Method Overriding**

```
class Parent:
```

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

class Child(Parent):
    def show(self):   # Overriding Parent method
        print("This is the Child class")

# Create object
obj = Child()
obj.show()
```

Output:

```
This is the Child class
```

🧠 **Explanation:**

- The Child class defines its own version of show().
- Even though Child inherits from Parent, its method **overrides** the parent one.

🔹 **2. Accessing the Parent Method using super()**

If you want to **call the parent class's version** of an overridden method,
use the **super()** function inside the child class.

✅ **Example:**

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

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

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

Output:

This is the Child class

This is the Parent class

🧠 **Explanation:**
super().show() allows you to access the parent's implementation of the overridden method.

🔹 **3. Overriding Constructor (__init__())**

Even **constructors** can be overridden in Python.

✅ **Example:**

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


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


obj = Child()
```

Output:

Child constructor

Parent constructor

🧠 **Explanation:**
Child overrides the parent's constructor but still calls the parent's one using super().

🔹 **4. Why Use Method Overriding?**

✅ To **change or extend** the behavior of a parent class method
✅ To **reuse parent functionality** but **add custom behavior**
✅ To **implement polymorphism** (same method, different behavior depending on the object)

🔹 **5. Real-Life Example:**

Imagine a program for **Vehicles**.

```
class Vehicle:
    def start(self):
        print("Vehicle is starting")

class Car(Vehicle):
    def start(self):
        print("Car engine starts with a key")

class ElectricCar(Vehicle):
    def start(self):
        print("Electric Car starts with a button")

# Create objects
v = Vehicle()
c = Car()
e = ElectricCar()

v.start()
c.start()
e.start()
```

Output:

```
Vehicle is starting
Car engine starts with a key
Electric Car starts with a button
```

SQLite3 and PyMySQL (Database Connectors)

Q18. Introduction to SQLite3 and PyMySQL for database connectivity.

A. Introduction to SQLite3 and PyMySQL for Database Connectivity in Python

Databases are used to **store and manage data** efficiently.
Python provides several built-in and external modules to connect and interact with databases — two of the most commonly used are **SQLite3** and **PyMySQL**.

Let’s explore both

🔹 **1. SQLite3 – Built-in Database in Python**

✅ **What is SQLite3?**

- **SQLite** is a **lightweight**, **serverless**, and **file-based** database system.
- It is included **by default** with Python — no installation required.
- It stores data in a **single .db or .sqlite file** on disk.

📘 **Best suited for:**

- Small to medium-sized applications
- Local storage
- Testing and prototyping

⚙️ **Basic Steps to Use SQLite3 in Python**

Step 1: Import the module

```
import sqlite3
```

Step 2: Connect to a database

```
conn = sqlite3.connect('students.db')  # Creates file if it doesn’t exist
```

Step 3: Create a cursor

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

Step 4: Execute SQL queries

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

Step 5: Insert data

```
cursor.execute("INSERT INTO students (name, age) VALUES (?, ?)", ("Zaid", 21))
conn.commit()
```

Step 6: Fetch data

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

```
    print(row)
```

Step 7: Close connection

```
conn.close()
```

✅ **Complete Example: SQLite3**

```
import sqlite3

# Connect to database (or create it)
conn = sqlite3.connect('students.db')
cursor = conn.cursor()

# Create 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 (?, ?)", ("Aisha", 20))
conn.commit()

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

# Close connection
conn.close()
```

Output:

```
(1, 'Aisha', 20)
```

🔹 2. PyMySQL – Connecting Python with MySQL

✅ What is PyMySQL?

- **PyMySQL** is a **third-party Python library** used to connect to a **MySQL database**.
- It is **not built-in**, so you must install it manually.

📦 Installation:

```
pip install PyMySQL
```

📘 Best suited for:

- Web applications
- Large-scale systems
- When you need remote, multi-user access

⚙️ Basic Steps to Use PyMySQL

Step 1: Import the module

```
import pymysql
```

Step 2: Connect to the MySQL server

```
conn = pymysql.connect(
    host="localhost",
    user="root",
    password="yourpassword",
    database="school"
)
```

Step 3: Create a cursor

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

Step 4: Execute SQL queries

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

Step 5: Insert data

```
cursor.execute("INSERT INTO students (name, age) VALUES (%s, %s)", ("Zaid", 21))
conn.commit()
```

Step 6: Fetch data

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

```
rows = cursor.fetchall()
for row in rows:
    print(row)
```

Step 7: Close connection

```
conn.close()
```

✅ **Complete Example: PyMySQL**

```
import pymysql

# Connect to MySQL
conn = pymysql.connect(
    host="localhost",
    user="root",
    password="1234",
    database="school"
)

cursor = conn.cursor()

# Create 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)", ("Ali", 22))
conn.commit()

# Retrieve data
cursor.execute("SELECT * FROM students")
for row in cursor.fetchall():
    print(row)
```

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

Output:

```
(1, 'Ali', 22)
```

🔹 3. Key Differences Between SQLite3 and PyMySQL

Feature	SQLite3	PyMySQL
Type	Built-in, file-based	External library, server-based
Installation	Not required	Requires pip install PyMySQL
Database File	Stored in .db or .sqlite file	Stored on MySQL server
Use Case	Local apps, testing	Web apps, enterprise systems
Speed	Fast for small datasets	Scales better for large datasets
Multi-user support	No	Yes

🔹 4. Best Practices

✅ Always use **parameterized queries** (? or %s) to prevent SQL Injection.
✅ Always **close the connection** (conn.close()) after use.
✅ Use **with statements** for automatic connection handling.
✅ Handle exceptions with try-except blocks.

🧩 Example with Exception Handling

```
import sqlite3

try:
    conn = sqlite3.connect('test.db')
    cursor = conn.cursor()
    cursor.execute("INSERT INTO students (name, age) VALUES (?, ?)", ("Sara", 19))
    conn.commit()
except sqlite3.Error as e:
    print("Error:", e)
```

```
finally:
    conn.close()
```

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

A. 1. Using SQLite3 (Built-in Database)

The **SQLite3** module lets you execute SQL commands on a lightweight, file-based database.

⚙️ **Step-by-Step Example:**

```
import sqlite3

# Step 1: Connect to database (or create one)
conn = sqlite3.connect('school.db')
cursor = conn.cursor()

# Step 2: Create a table
cursor.execute('''
CREATE TABLE IF NOT EXISTS students (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    age INTEGER,
    grade TEXT
)
''')

# Step 3: Insert data
cursor.execute("INSERT INTO students (name, age, grade) VALUES (?, ?, ?)", ("Zaid", 21, "A"))
cursor.execute("INSERT INTO students (name, age, grade) VALUES (?, ?, ?)", ("Aisha", 20, "B"))
conn.commit()  # Save changes

# Step 4: Read (SELECT) data
cursor.execute("SELECT * FROM students")
```

```
for row in cursor.fetchall():
    print(row)

# Step 5: Update data
cursor.execute("UPDATE students SET grade = ? WHERE name = ?", ("A+", "Zaid"))
conn.commit()

# Step 6: Delete data
cursor.execute("DELETE FROM students WHERE name = ?", ("Aisha",))
conn.commit()

# Step 7: Close connection
conn.close()
```

🧠 **What's Happening Here**

Operation	SQL Query	Python Method
Create table	CREATE TABLE ...	cursor.execute()
Insert record	INSERT INTO ... VALUES ...	cursor.execute()
Fetch records	SELECT * FROM ...	cursor.fetchall()
Update record	UPDATE ... SET ... WHERE ...	cursor.execute()
Delete record	DELETE FROM ... WHERE ...	cursor.execute()
Save changes	—	conn.commit()
Close connection	—	conn.close()

🐬 **2. Using PyMySQL (MySQL Connector)**

For connecting to **MySQL** databases (server-based), you use the **PyMySQL** library.

⚙️ **Example:**

```
import pymysql
```

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

# Step 2: Create a table
cursor.execute('''
CREATE TABLE IF NOT EXISTS students (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(50),
    age INT,
    grade VARCHAR(5)
)
''')

# Step 3: Insert data
cursor.execute("INSERT INTO students (name, age, grade) VALUES (%s, %s, %s)", ("Ali", 22, "A"))
cursor.execute("INSERT INTO students (name, age, grade) VALUES (%s, %s, %s)", ("Sara", 19, "B"))
conn.commit()

# Step 4: Select data
cursor.execute("SELECT * FROM students")
for row in cursor.fetchall():
    print(row)

# Step 5: Update data
cursor.execute("UPDATE students SET grade=%s WHERE name=%s", ("A+", "Sara"))
```

```
conn.commit()

# Step 6: Delete data
cursor.execute("DELETE FROM students WHERE name=%s", ("Ali",))
conn.commit()

# Step 7: Close connection
conn.close()
```

⚙️ **Parameter Style**

Connector	Placeholder Syntax
SQLite3	?
PyMySQL	%s

Both are **parameterized queries** — this protects you from **SQL injection attacks** ✅

⚡ **3. Executing Multiple SQL Queries**

You can run multiple queries at once using **executemany()**:

```
data = [
    ("Rehan", 23, "B"),
    ("Fatima", 20, "A"),
    ("Omar", 21, "A-")
]

cursor.executemany("INSERT INTO students (name, age, grade) VALUES (?, ?, ?)", data)
conn.commit()
```

⚙️ **4. Using Exception Handling (Best Practice)**

```
import sqlite3

try:
```

```
    conn = sqlite3.connect('school.db')
    cursor = conn.cursor()
    cursor.execute("INSERT INTO students (name, age, grade) VALUES (?, ?, ?)", ("Mina", 22, "B+"))
    conn.commit()
except sqlite3.Error as e:
    print("Database Error:", e)
finally:
    conn.close()
```

Search and Match Functions

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

A. Introduction to the re Module

The **re module** in Python provides tools for working with **Regular Expressions (RegEx)** — patterns used to match and manipulate text.

Before using it:

```
import re
```

🔹 **1. re.match() — Matches Pattern Only at the Beginning of a String**

📘 **Syntax**

```
re.match(pattern, string)
```

✅ **Description**

- Tries to **match the pattern only at the start** of the string.
- Returns a **Match object** if found, otherwise None.

🧠 **Example 1: Simple match**

```
import re

text = "Python is fun!"
result = re.match("Python", text)
```

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

Output:

```
Match found: Python
```

✅ The pattern "Python" is at the **beginning** of the string, so it matches.

🧠 **Example 2: Pattern not at the start**

```
text = "I love Python"
result = re.match("Python", text)

print(result)
```

Output:

```
None
```

❌ re.match() fails because “Python” is **not at the start** of the string.

🔹 **2. re.search() — Searches the Entire String**

📘 **Syntax**

```
re.search(pattern, string)
```

✅ **Description**

- Scans the **entire string** for a match.
- Returns a **Match object** of the **first occurrence** if found, otherwise None.

🧠 **Example 1: Search in full string**

```
import re

text = "I love Python programming."
result = re.search("Python", text)
```

```
if result:
    print("Pattern found at position:", result.start())
else:
    print("Pattern not found.")
```

Output:

```
Pattern found at position: 7
```

✅ Found "Python" starting at index 7 in the string.

🧠 **Example 2: Using group() method**

```
text = "Learning Python is easy!"
match = re.search("Python", text)

if match:
    print("Matched text:", match.group())
```

Output:

```
Matched text: Python
```

🔹 **3. Key Difference Between re.match() and re.search()**

Feature	re.match()	re.search()
Searches	Only at the beginning	Entire string
Returns	Match object if pattern at start	Match object if pattern found anywhere
Typical use	When pattern must start at position 0	When pattern can occur anywhere
Example	re.match("abc", "abcdef") ✅	re.search("abc", "123abc456") ✅

🧩 **Example Comparing Both**

```
import re

text = "Welcome to Python world"

match_result = re.match("Python", text)
```

```
search_result = re.search("Python", text)


print("Match result:", match_result)
print("Search result:", search_result)
```

Output:

```
Match result: None
Search result: <re.Match object; span=(11, 17), match='Python'>
```

✅ re.match() failed since “Python” is not at the start.
✅ re.search() succeeded since “Python” appears later.

🔹 4. Using Regular Expression Patterns

You can use **special regex symbols** in both functions.

Example: Match digits at the start

```
text = "123abc"
if re.match(r"\d+", text):
    print("Starts with digits.")
```

Output:

```
Starts with digits.
```

Example: Search digits anywhere

```
text = "abc123xyz"
if re.search(r"\d+", text):
    print("Contains digits.")
```

Output:

```
Contains digits.
```

🔹 5. Commonly Used Methods with Match Objects

Method	Description	Example
.group()	Returns matched text	match.group()
.start()	Returns starting index	match.start()
.end()	Returns ending index	match.end()

Method	Description	Example
.span()	Returns (start, end) tuple	match.span()

🧠 **Example**

```
text = "The number is 12345"

match = re.search(r"\d+", text)


if match:
    print("Matched:", match.group())
    print("Start index:", match.start())
    print("End index:", match.end())
    print("Span:", match.span())
```

Output:

Matched: 12345

Start index: 14

End index: 19

Span: (14, 19)

Q21. Difference between search and match.

A. 🧩 Key Difference: re.match() vs re.search()

Feature	re.match()	re.search()
What it does	Checks **only the beginning** of the string for a match.	Searches **the entire string** for the first occurrence of the pattern.
When it matches	Only if the pattern is **at position 0** (start).	If the pattern appears **anywhere** in the string.
Return Value	Returns a **Match object** if found at the start; otherwise None.	Returns a **Match object** if found anywhere; otherwise None.
Common Use Case	Validate formats that must begin with something (e.g., start of line, prefix).	Find or extract information appearing anywhere in text.
Performance	Slightly faster for prefix checks.	Slightly slower since it scans the whole string.

🧠 **Example 1: When both give different results**

```
import re

text = "I love Python"

# Match checks only at the start
result_match = re.match("Python", text)

# Search checks the entire string
result_search = re.search("Python", text)

print("Match result:", result_match)
print("Search result:", result_search)
```

Output:

```
Match result: None
Search result: <re.Match object; span=(7, 13), match='Python'>
```

✅ **Explanation:**

- re.match() → returns None because "Python" is **not at the beginning**.
- re.search() → returns a match because "Python" appears **later in the string**.

🧠 **Example 2: When both give the same result**

```
text = "Python is fun"

result_match = re.match("Python", text)
result_search = re.search("Python", text)

print("Match:", result_match.group())
print("Search:", result_search.group())
```

Output:

```
Match: Python
```

Search: Python

✅ **Explanation:**
Both match because "Python" appears **at the beginning** of the string.