Theory Exercise

Introduction Python

Q1. Introduction to Python and its Features (simple, high-level, interpreted language).

A.

Introduction to Python

Python is a **high-level, interpreted programming language** known for its **simplicity, readability, and versatility**.
It was created by **Guido van Rossum** and released in **1991**.
Python is widely used in various fields such as **web development, data science, machine learning, automation, and software development**.

Features of Python

1. **Simple and Easy to Learn**
    - Python has a clean and readable syntax similar to English.
    - It's beginner-friendly and reduces the learning curve for new programmers.
2. **High-Level Language**
    - You don't need to worry about low-level details like memory management.
    - Python handles most of the complex operations internally.
3. **Interpreted Language**
    - Python code is executed line by line by an interpreter.
    - This makes debugging easier since errors are detected one line at a time.
4. **Platform Independent**
    - Python programs can run on different operating systems (Windows, Mac, Linux) without modification.
5. **Open Source and Free**
    - Python is free to download and use.
    - It has a large community that contributes to libraries and support.
6. **Extensive Libraries**
    - Python provides a vast collection of built-in and external libraries for tasks like data analysis, web development, and AI.
7. **Object-Oriented**

- Python supports object-oriented programming concepts like classes and objects, promoting code reusability.

8. **Portable and Extensible**
    - Python can integrate with other languages like C, C++, or Java.
9. **Dynamic Typing**
    - You don't need to declare variable types; Python automatically detects them at runtime.
10. **Large Community Support**

- Python has a huge community, meaning help and documentation are easy to find.

Q2. History and evolution of Python.

A. 🐍 History and Evolution of Python

Python was created by **Guido van Rossum** in the **late 1980s** and officially released in **1991**. It was designed as a simple, easy-to-understand programming language that emphasizes **code readability** and **productivity**.

📜 Timeline of Python's Evolution

1. **1980s – The Beginning**
    - Guido van Rossum was working at **CWI (Centrum Wiskunde & Informatica)** in the Netherlands.
    - He wanted to create a language that could overcome the limitations of the **ABC language** (a simple teaching language at that time).
    - His goal was to design a language that was **easy to learn**, **powerful**, and **extensible**.
2. **1991 – Python 1.0 Released**
    - The first official version, **Python 1.0**, was released on **February 20, 1991**.
    - It included features like:
        - Functions
        - Exception handling
        - Core data types (str, list, dict, etc.)
        - Modules for code reuse
3. **2000 – Python 2.0 Released**

- Released on **October 16, 2000**.
- Introduced major improvements:
  - **List comprehensions**
  - **Garbage collection** (automatic memory management)
  - **Unicode support**
- However, Python 2 had design limitations that made it hard to maintain in the long term.

4. **2008 – Python 3.0 Released**
   - Released on **December 3, 2008**.
   - It was not backward compatible with Python 2, meaning old code needed updates.
   - Key improvements:
     - Better **Unicode support**
     - **Improved syntax** and **clearer semantics**
     - **print()** function instead of the print statement
     - More consistency across the language
5. **2020 – End of Python 2 Support**
   - Official support for **Python 2 ended on January 1, 2020**.
   - Since then, **Python 3** is the only maintained version.
6. **Present Day – Continuous Growth**
   - Python is one of the **most popular programming languages** worldwide.
   - It's widely used in **AI, Machine Learning, Data Science, Web Development, Automation**, and more.
   - The latest versions (Python 3.x) keep improving in performance, security, and simplicity.

Q3. Advantages of using Python over other programming languages.

A. ⚙️ Advantages of Using Python Over Other Programming Languages

1. **Easy to Learn and Read**
   - Python's syntax is simple and close to English, making it easy for beginners to learn and understand.

- Code readability is high, which reduces development and debugging time.

2. **Interpreted Language**
    - Python does not need compilation; it executes code line by line.
    - This makes testing and debugging easier compared to compiled languages like C or C++.
3. **Cross-Platform Compatibility**
    - Python programs can run on **Windows, macOS, and Linux** without any changes to the code.
4. **Extensive Standard Library**
    - Python comes with a large number of built-in modules and libraries for tasks like file handling, web development, data analysis, and networking.
    - This saves time and effort in writing code from scratch.
5. **Supports Multiple Programming Paradigms**
    - Python supports **procedural, object-oriented, and functional** programming styles, giving flexibility to developers.
6. **Open Source and Free**
    - Python is freely available for everyone to use, modify, and distribute.
    - It has a huge community that continuously improves the language and provides support.
7. **Large Community Support**
    - Python has one of the largest developer communities, meaning it's easy to find tutorials, documentation, and help online.
8. **Integration Capabilities**
    - Python can easily integrate with other languages like **C, C++, and Java**, and also supports **web APIs and databases**.
9. **Ideal for Rapid Development**
    - With fewer lines of code and a simple syntax, Python allows faster development compared to languages like Java or C++.
10. **Strong in Emerging Technologies**

- Python is widely used in **Artificial Intelligence, Machine Learning, Data Science, Automation, and Web Development**, making it future-ready.

Q4. Installing Python and setting up the development environment (Anaconda, PyCharm, or VS Code).

A. 🐍 Installing Python and Setting Up the Development Environment

1. Installing Python

🧩 Step 1: Download Python

- Go to the official website: https://www.python.org/downloads
- Click on **"Download Python [latest version]"** (e.g., Python 3.12).

⚙️ Step 2: Install Python

- Run the downloaded installer.
- **Important:** Check ✅ **"Add Python to PATH"** before clicking *Install Now*.
- After installation, open the **Command Prompt** or **Terminal** and type:
- python --version

If it shows the Python version, the installation is successful.

---

2. Setting Up Python Using Anaconda

🧱 What is Anaconda?

- **Anaconda** is a distribution of Python used mainly for **Data Science, Machine Learning, and AI**.
- It includes tools like **Jupyter Notebook**, **Spyder**, and pre-installed libraries (NumPy, Pandas, Matplotlib, etc.).

🔧 Installation Steps:

1. Go to **https://www.anaconda.com/download**
2. Download the version for your operating system (Windows/Mac/Linux).
3. Run the installer and follow the setup instructions.
4. After installation, open **Anaconda Navigator** to launch tools like:
    - **Jupyter Notebook** – for interactive coding.
    - **Spyder** – for script-based coding (like an IDE).

---

3. Setting Up Python in PyCharm

💡 What is PyCharm?

- **PyCharm** is a popular **IDE (Integrated Development Environment)** made by JetBrains for Python programming.
- It provides features like code completion, debugging, and project management.

⚙️ **Installation Steps:**

1. Go to https://www.jetbrains.com/pycharm/download
2. Download **PyCharm Community Edition** (free version).
3. Install and open PyCharm.
4. Click **"New Project"**, select **Python Interpreter**, and choose your installed Python version.
5. Start coding in the editor window.

---

4. Setting Up Python in VS Code

💻 **What is VS Code?**

- **Visual Studio Code (VS Code)** is a lightweight code editor developed by Microsoft.
- It supports many languages and has extensions for Python development.

⚙️ **Installation Steps:**

1. Download from **https://code.visualstudio.com/**.
2. Install and open VS Code.
3. Go to **Extensions (Ctrl+Shift+X)** → Search and install **"Python"** extension by Microsoft.
4. Open your Python file or folder.
5. Select the Python interpreter by pressing **Ctrl+Shift+P → "Python: Select Interpreter"**.
6. You can now run your code using the **Run ▶️ button** or the terminal.

Q5. What are the basic components of SQL syntax?

A. 🐍 Writing and Executing Your First Python Program

🧩 **Step 1: Check Python Installation**

Before writing a program, make sure Python is installed.

- Open **Command Prompt (Windows)** or **Terminal (Mac/Linux)**
- Type:
- python --version

If it shows a version number (like Python 3.12.0), you're ready to go.

🖊️ **Step 2: Write Your First Python Program**

Option 1: Using IDLE (comes with Python)

1. Open **IDLE** (Python's built-in editor).
2. Click **File → New File**.
3. Type this simple program:

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

4. Save the file as **hello.py**.
5. Click **Run → Run Module** (or press **F5**).
6. Output will appear as:

```
Hello, World!
```

Programming Style

Q6. Understanding Python's PEP 8 guidelines.

A. 🐍 Understanding Python's PEP 8 Guidelines

💡 What is PEP 8?

PEP 8 stands for **Python Enhancement Proposal 8**.
It is the **official style guide** for writing **clean, readable, and consistent** Python code.

It was written by **Guido van Rossum**, **Barry Warsaw**, and **Nick Coghlan** to help programmers follow best practices in Python programming.

🎯 Purpose of PEP 8

- To make Python code **more readable and consistent** across projects.
- To help developers **understand each other's code easily**.
- To **maintain coding standards** for large teams and open-source projects.

⚙️ Key PEP 8 Guidelines

1. Indentation

- Use **4 spaces** per indentation level.

```
if True:
    print("Correct indentation")
```

- **Do not use tabs**; always use spaces.

---

2. Line Length

- Keep lines **under 79 characters** long.

- For long statements, use **line continuation** with a backslash \ or parentheses ().

---

3. Blank Lines

- Use blank lines to separate functions, classes, and sections of code for better readability.

```
def func1():
    pass



def func2():
    pass
```

---

4. Imports

- Place all imports at the **top** of the file.
- Each import should be on a **separate line**.

```
import os
import sys
```

---

5. Naming Conventions

- **Variables and functions**: use lowercase letters with underscores → total_sum, calculate_area()
- **Classes**: use PascalCase → StudentData, CarModel
- **Constants**: use all uppercase letters → PI = 3.14

---

6. Spaces Around Operators

- Put spaces around operators and after commas for clarity.

```
x = 10 + 5
print(x, y)
```

---

7. Comments

- Use comments to explain code logic.
- Use **#** for single-line comments.

- # This function adds two numbers

  def add(a, b):

    return a + b

- For multi-line comments, use triple quotes (""" ... """).

---

8. Docstrings

- Write docstrings for functions, classes, and modules to describe their purpose.

  def greet(name):

    """This function greets the person passed as a parameter."""

    print("Hello,", name)

---

9. Avoid Unnecessary Spaces

❌ Wrong:

x= 5

y =10

✅ Correct:

x = 5

y = 10

---

10. Consistent Code Layout

- Always use consistent formatting in your code to make it professional and readable.

Q7. Indentation, comments, and naming conventions in Python.

A. 🐍 Indentation, Comments, and Naming Conventions in Python

🧩 1. Indentation in Python

💡 What is Indentation?

Indentation means **adding spaces at the beginning of a line of code** to define blocks of code.
Unlike many other languages, **Python uses indentation instead of braces {}** to represent code blocks.

⚙️ Rules:

- Use **4 spaces** for each indentation level (as per **PEP 8** guidelines).

- All statements inside a block (like loops, if statements, or functions) must have **the same level of indentation**.

✅ **Example:**

```
if True:
    print("This is indented correctly")
    print("This line is part of the same block")
```

❌ **Incorrect Example:**

```
if True:
print("This will cause an IndentationError")
```

🧠 **Why Important?**

- Indentation defines the structure and flow of a Python program.
- Incorrect indentation leads to **IndentationError**.

💬 **2. Comments in Python**

💡 **What are Comments?**

Comments are lines in the code that are **ignored by the interpreter**.
They are used to **explain code**, **improve readability**, and **help future maintenance**.

🧱 **Types of Comments:**

a) Single-line Comment

- Starts with a # symbol.

```
# This is a single-line comment
print("Hello, World!")  # This prints a message
```

b) Multi-line Comment

- You can use **triple quotes** (''' or """) for multi-line comments.

```
"""
This is a multi-line comment.
It can span several lines.
"""
print("Welcome to Python!")
```

💡 **Use Comments to:**

- Explain what a piece of code does.

- Describe function purpose.
- Temporarily disable a line during debugging.

🏷️ **3. Naming Conventions in Python**

💡 **What are Naming Conventions?**

Naming conventions are **rules for naming variables, functions, classes, and constants** to keep code clear and consistent.

Q8. Writing readable and maintainable code.

A. 🐍 Writing Readable and Maintainable Code in Python

💡 **What Does It Mean?**

Readable code is code that is **easy to understand**, even for someone who didn't write it.
Maintainable code is code that can be **easily updated, fixed, or improved** in the future without breaking other parts of the program.

Writing readable and maintainable code is important because it:

- Saves time during debugging and updates.
- Helps teams work together efficiently.
- Reduces errors and improves code quality.

⚙️ **Tips for Writing Readable and Maintainable Python Code**

🧩 **1. Follow PEP 8 Guidelines**

- Use **proper indentation (4 spaces)**.
- Keep line length below **79 characters**.
- Use **meaningful variable names** and proper spacing.

```
# Good
total_marks = math_marks + science_marks

# Bad
t = m + s
```

💬 2. Use Comments Wisely

- Add comments to explain *why* you wrote the code, not *what* it does.
- Keep comments short, clear, and updated.

```
# Calculate total bill after applying discount
total_bill = price - (price * discount)
```

🏷️ 3. Use Meaningful Names

- Variable, function, and class names should describe their purpose.

```
def calculate_area(radius):
    return 3.14 * radius * radius
```

🧱 4. Organize Code into Functions and Modules

- Divide your code into **functions** to avoid repetition.
- Group related functions into **modules or classes** for better organization.

```
def greet_user(name):
    print(f"Hello, {name}!")
```

🔄 5. Avoid Hardcoding Values

- Use variables or constants instead of directly writing numbers or strings.

```
TAX_RATE = 0.05
total = price + (price * TAX_RATE)
```

📄 6. Use Docstrings

- Write a **docstring** (""" ... """) at the beginning of functions, classes, or modules to describe their purpose.

```
def add_numbers(a, b):
    """Return the sum of two numbers."""
    return a + b
```

🧠 7. Handle Errors Gracefully

- Use **try-except** blocks to handle exceptions instead of letting the program crash.

```
try:
    result = int(input("Enter a number: "))
except ValueError:
    print("Invalid input! Please enter a number.")
```

🖌️ **8. Keep Code DRY (Don't Repeat Yourself)**

- Avoid duplicating code; reuse functions or loops whenever possible.

```
# Instead of writing same code twice, use a function
def greet(name):
    print(f"Hello, {name}!")
```

⚡ **9. Consistent Formatting**

- Use the same naming style, indentation, and spacing throughout your project.
- Tools like **Black** or **Autopep8** can automatically format your code.

🤝 **10. Write Modular and Testable Code**

- Write small, independent modules that can be tested separately.
- Helps in debugging and improving code quality.

Core Python Concepts

Q9. Understanding data types: integers, floats, strings, lists, tuples, dictionaries, sets.

A. 🐍 Understanding Python Data Types

In Python, **data types** represent the **kind of value** a variable can hold.
They define how the data is stored and what operations can be performed on it.

🔢 **1. Integers (int)**

- Integers are **whole numbers** (positive or negative) without decimal points.
- Example:

```
x = 10
y = -5
```

- You can perform arithmetic operations like addition, subtraction, etc.

```
result = x + y   # 10 + (-5) = 5
```

💧 2. Floats (float)

- Floats represent **decimal or fractional numbers**.
- Example:

```
pi = 3.14

temperature = -12.5
```

- Used when precision is required (e.g., scientific calculations).

🔤 3. Strings (str)

- Strings are a **sequence of characters** enclosed in **single (' ')** or **double (" ")** quotes.
- Example:

```
name = "Alice"

greeting = 'Hello, World!'
```

- Strings support many operations like:

```
print(name.upper())

print(name[0])

print(greeting[7:12])
```

📋 4. Lists (list)

- Lists are **ordered**, **changeable (mutable)** collections of items.
- Elements can be of **different data types**.
- Example:

```
fruits = ["apple", "banana", "cherry"]

fruits[1] = "mango"

print(fruits)
```

- Lists allow operations like:

```
fruits.append("orange")

fruits.remove("apple")
```

🔒 5. Tuples (tuple)

- Tuples are **ordered** collections like lists, but they are **immutable** (cannot be changed).
- Example:

```
coordinates = (10, 20, 30)
print(coordinates[1])
```

- You **cannot modify** or remove elements once created.

📘 6. Dictionaries (dict)

- Dictionaries store data as **key–value pairs**.
- They are **unordered** (before Python 3.7) and **mutable**.
- Example:

```
student = {
   "name": "John",
   "age": 20,
   "course": "Python"
}
print(student["name"])
student["age"] = 21
```

- Useful for storing structured data.

🧺 7. Sets (set)

- Sets are **unordered collections** of **unique** elements (no duplicates).
- Example:

```
numbers = {1, 2, 3, 3, 4}
print(numbers)
```

- You can perform **set operations** like union, intersection, and difference:

```
a = {1, 2, 3}
b = {3, 4, 5}
print(a.union(b))
print(a.intersection(b))
```

Q10. Python variables and memory allocation.

A. 🐍 Python Variables and Memory Allocation

💡 What is a Variable?

A **variable** is a **name** that is used to store a value in memory.
It acts like a **container** or **label** that refers to data stored in the computer's memory.

In Python, you don't need to declare a variable's type — it is **automatically assigned** when you store a value.

✅ Example:

```
x = 10       # Integer
name = "John" # String
pi = 3.14    # Float
```

Here, x, name, and pi are variables that hold values in memory.

⚙️ Variable Declaration Rules

1. Variable names can include **letters, digits, and underscores (_)**.
2. They **cannot start with a digit**.
3. They are **case-sensitive** (name and Name are different).
4. You **cannot use keywords** (like if, class, while, etc.) as variable names.

✅ Valid Examples:

```
age = 25
first_name = "Alice"
total_marks = 480
```

❌ Invalid Examples:

```
2name = "Bob"    # Starts with a digit
for = 10         # Uses reserved keyword
student-name = 50 # Contains invalid character '-'
```

🧠 Dynamic Typing in Python

Python uses **dynamic typing**, meaning you don't have to specify a variable's type.
The interpreter decides the type automatically based on the assigned value.

```
x = 10       # x is an integer
x = "Hello"   # Now x becomes a string
```

🧱 How Memory Allocation Works

When you create a variable, Python does the following internally:

1. **Creates an object** in memory to store the value.
2. **Assigns a reference (variable name)** to that object.
3. The variable name acts as a **pointer** to the memory location.

Example:

```
a = 10
b = a
```

- Both a and b point to the **same memory location** containing the integer 10.
- If a is reassigned, b remains unchanged.

🧰 id() Function

The built-in id() function returns the **unique memory address (identity)** of an object.

```
x = 10
y = 10
print(id(x))
print(id(y))
```

In Python, small integers and strings are **interned**, so x and y may have the same id.

🔄 Garbage Collection

When a variable is no longer in use (no references point to it), Python automatically **frees up the memory** through a process called **garbage collection**.

Example:

```
x = 50
x = None  # Old memory is released
```

Q11. Python operators: arithmetic, comparison, logical, bitwise.

A. Python Operators

Operators in Python are **special symbols** that perform operations on **variables and values**.
Python supports several types of operators such as **arithmetic, comparison, logical, and bitwise** operators.

---

1 Arithmetic Operators

Used to perform **mathematical calculations**.

Operator	Description	Example	Output
+	Addition	5 + 3	8
-	Subtraction	5 - 3	2
*	Multiplication	5 * 3	15
/	Division (float)	5 / 2	2.5
//	Floor Division (integer)	5 // 2	2
%	Modulus (remainder)	5 % 2	1
**	Exponentiation	2 ** 3	8

---

2 Comparison Operators

Used to **compare values** and return **True or False**.

Operator	Description	Example	Output
==	Equal to	5 == 5	True
!=	Not equal to	5 != 3	True
>	Greater than	5 > 3	True
<	Less than	5 < 3	False
>=	Greater than or equal to	5 >= 5	True
<=	Less than or equal to	5 <= 3	False

---

3 Logical Operators

Used to combine **Boolean conditions** (True or False).

Operator	Description	Example	Output
and	Returns True if **both conditions** are True	(5>3) and (2<4)	True
or	Returns True if **any one condition** is True	(5>3) or (2>4)	True
not	Reverses the Boolean value	not(5>3)	False

4️⃣ Bitwise Operators

Used to perform operations on **binary representations** of integers.

Operator	Description	Example	Output
&	AND	5 & 3 → 0101 & 0011	1
`	`	OR	`5
^	XOR	5 ^ 3 → 0101 ^ 0011	6
~	NOT	~5 → ~0101	-6
<<	Left Shift	5 << 1 → 0101 << 1	10
>>	Right Shift	5 >> 1 → 0101 >> 1	2

Conditional Statements

Q12. Introduction to conditional statements: if, else, elif.

A. 🐍 Introduction to Conditional Statements in Python

💡 What Are Conditional Statements?

Conditional statements allow a program to **make decisions** based on certain conditions.
They let the program execute different code blocks depending on whether a condition is **True** or **False**.

Python provides three main conditional statements:

1️⃣ if Statement

- Executes a block of code **only if a condition is True**.

Syntax:

```
if condition:
    # code to execute if condition is True
```

Example:

```
age = 18
if age >= 18:
    print("You are an adult")
```

Output:

```
You are an adult
```

2️⃣ else Statement

- Executes a block of code **if the condition in if is False**.

Syntax:

```
if condition:
    # code if True
else:
    # code if False
```

Example:

```
age = 16
if age >= 18:
    print("You are an adult")
else:
    print("You are a minor")
```

Output:

```
You are a minor
```

3️⃣ elif Statement (Else If)

- Stands for **“else if”**.
- Allows testing **multiple conditions** in sequence.
- Only the **first True condition** block is executed.

Syntax:

```
if condition1:
    # code if condition1 is True
elif condition2:
    # code if condition2 is True
```

```
else:
    # code if all conditions are False
```

Example:

```
marks = 75
if marks >= 90:
    print("Grade: A")
elif marks >= 70:
    print("Grade: B")
elif marks >= 50:
    print("Grade: C")
else:
    print("Grade: F")
```

Output:

```
Grade: B
```

Q13. Nested if-else conditions.

A. 🐍 Nested if-else Conditions

💡 What is a Nested if-else?

A **nested if-else** is an **if or else statement inside another if or else statement**.
It allows you to check **multiple levels of conditions** within a single block of code.

⚙️ Syntax

```
if condition1:
    if condition2:
        # code executed if both condition1 and condition2 are True
    else:
        # code executed if condition1 is True but condition2 is False
else:
    # code executed if condition1 is False
```

📝 Example 1: Checking Age and Student Status

```
age = 20
is_student = True

if age >= 18:
    if is_student:
        print("You are an adult student")
    else:
        print("You are an adult but not a student")
else:
    print("You are a minor")
```

Output:

```
You are an adult student
```

📝 Example 2: Grading System

```
marks = 85

if marks >= 90:
    print("Grade: A")
else:
    if marks >= 75:
        print("Grade: B")
    else:
        if marks >= 50:
            print("Grade: C")
        else:
            print("Grade: F")
```

Output:

```
Grade: B
```

Looping (For, While)

Q14. Introduction to for and while loops.

A. 🐍 Introduction to Loops in Python

💡 What is a Loop?

A **loop** is used to **execute a block of code repeatedly** as long as a certain condition is true.
Loops help avoid **writing repetitive code**.

Python has two main types of loops:

1️⃣ for Loop

- Used to **iterate over a sequence** like a list, tuple, string, or range of numbers.
- Executes a block of code for **each item** in the sequence.

⚙️ Syntax:

```
for variable in sequence:
    # code to execute
```

📝 Example:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Output:

```
apple
banana
cherry
```

Example with range():

```
for i in range(5):  # 0 to 4
    print(i)
```

Output:

```
0
1
2
3
4
```

2️⃣ while Loop

- Repeats a block of code **as long as a condition is True**.
- The condition is checked **before each iteration**.

⚙️ Syntax:

```
while condition:
  # code to execute
```

📝 Example:

```
i = 1
while i <= 5:
  print(i)
  i += 1
```

Output:

```
1
2
3
4
5
```

Q15. How loops work in Python.

A. 🐍 How Loops Work in Python

Loops in Python are used to **repeat a block of code** multiple times. Python has two main types of loops: **for loop** and **while loop**. Each works in a slightly different way.

1️⃣ For Loop

💡 How it Works:

- The **for loop** iterates over a **sequence** (like a list, tuple, string, or range).
- Python assigns **each element** of the sequence to a **loop variable**, executes the loop body, and then moves to the next element.
- The loop **ends automatically** after the last element.

⚙️ Flow:

Start → Take first element → Execute code block → Next element → Repeat → End

Example:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
  print(fruit)
```

Output:

```
apple
banana
cherry
```

2️⃣ While Loop

💡 How it Works:

- The **while loop** repeats a block of code **as long as a condition is True**.
- Before each iteration, Python **checks the condition**.
- When the condition becomes False, the loop **stops automatically**.

⚙️ Flow:

Start → Check condition → True? → Execute code → Repeat → False? → End

Example:

```
i = 1
while i <= 3:
  print(i)
  i += 1
```

Output:

```
1
2
3
```

🔁 Loop Control Statements

- **break** → Immediately exits the loop.
- **continue** → Skips the current iteration and moves to the next.
- **else (optional)** → Executes a block when the loop ends normally (without break).

Example:

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

Output:

```
0
1
2
```

Q16. Using loops with collections (lists, tuples, etc.).

A. Using Loops with Collections in Python

In Python, **collections** like **lists, tuples, sets, and dictionaries** store multiple items. Loops allow you to **process each item one by one** without writing repetitive code.

---

1 Using for Loop with a List

A list is **ordered and mutable**. You can iterate over it easily using a for loop.

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Output:

```
apple
banana
cherry
```

---

2 Using for Loop with a Tuple

Tuples are **ordered but immutable**. You can loop over them like lists.

```
numbers = (1, 2, 3, 4, 5)
for num in numbers:
    print(num * 2)
```

Output:

```
2
4
6
8
10
```

3️⃣ Using for Loop with a Set

Sets are **unordered collections with unique elements**. Looping works, but the order is unpredictable.

```
colors = {"red", "green", "blue"}
for color in colors:
    print(color)
```

Output (example, order may vary):

```
green
blue
red
```

4️⃣ Using for Loop with a Dictionary

Dictionaries store **key–value pairs**. You can loop over:

- **Keys only** (default)
- **Values only**
- **Both keys and values**

```
student = {"name": "Alice", "age": 20, "course": "Python"}


# Loop through keys
for key in student:
    print(key)


# Loop through values
for value in student.values():
    print(value)
```

```
# Loop through key-value pairs
for key, value in student.items():
    print(key, ":", value)
```

Output:

```
name
age
course
Alice
20
Python
name : Alice
age : 20
course : Python
```

5️⃣ Using while Loop with Collections

- While loops can also be used with collections using **indexing**.

```
numbers = [10, 20, 30, 40]
i = 0
while i < len(numbers):
    print(numbers[i])
    i += 1
```

Output:

```
10
20
30
40
```

Generators and Iterators

Q17. Understanding how generators work in Python.

A. 🐍 Understanding Generators in Python

💡 What is a Generator?

A **generator** is a **special type of function** that **returns an iterator** which **yields items one at a time** instead of returning all of them at once.

Generators are **memory-efficient**, especially when working with **large data sets**, because they **produce items on-the-fly** rather than storing everything in memory.

⚙️ How Generators Work

1. A generator function uses the yield keyword instead of return.
2. When called, it **does not execute the function immediately**; it returns a generator object.
3. Each call to next() on the generator **executes the function until the next yield**, pauses, and returns the value.
4. Execution **resumes from where it left off** the next time next() is called.

📝 Example 1: Simple Generator

```
def simple_gen():
    yield 1
    yield 2
    yield 3

gen = simple_gen()

print(next(gen))  # Output: 1
print(next(gen))  # Output: 2
print(next(gen))  # Output: 3
```

- The function simple_gen() **yields values one by one**.
- After the last yield, calling next() again will raise StopIteration.

📝 **Example 2: Generator with Loop**

```
def squares(n):
    for i in range(1, n+1):
        yield i ** 2

for square in squares(5):
    print(square)
```

Output:

```
1
4
9
16
25
```

- Each square is **generated on-the-fly** as the loop iterates.
- This avoids storing all 25 squares in memory at once.

Q18. Difference between yield and return.

A. 🐍 Difference Between yield and return

Both yield and return are used in functions to send values back, but they **behave differently**.

1️⃣ return

- **Ends the function immediately**.
- Sends a **single value** back to the caller.
- The function cannot resume after returning.

Example:

```
def my_func():
    return 10
    print("This will never run")  # Code after return is ignored

print(my_func())  # Output: 10
```

Key Points:

- Function execution **stops** after return.
- Returns a **value** (not an iterator).
- Suitable when you want **one-time result**.

2️⃣ yield

- **Pauses the function** instead of ending it.
- Returns a **generator object** that can produce multiple values, one at a time.
- Execution **resumes** from the point of yield the next time the generator is called.

Example:

```
def my_gen():
    yield 1
    yield 2
    yield 3

gen = my_gen()
print(next(gen))  # Output: 1
print(next(gen))  # Output: 2
print(next(gen))  # Output: 3
```

Key Points:

- Function execution **pauses at yield** and can **resume later**.
- Returns a **generator** (iterator) instead of a single value.
- Memory-efficient for **large sequences**.

Q19. Difference between yield and return.

A. 🐍 Understanding Iterators in Python

💡 What is an Iterator?

An **iterator** is an object in Python that **allows you to traverse through all the elements of a collection one at a time**.

- You can get the **next element** using the built-in next() function.
- Iterators remember their **current position**, so you don't have to manage indexes manually.

- Most Python **collections (lists, tuples, dictionaries, sets)** are iterable, meaning they can return an iterator using iter().

⚙️ **Basic Iterator Example**

```
numbers = [10, 20, 30]


# Get an iterator from the list
num_iter = iter(numbers)


print(next(num_iter))  # 10
print(next(num_iter))  # 20
print(next(num_iter))  # 30
# print(next(num_iter))  # Would raise StopIteration
```

Key Points:

- iter() returns an iterator object.
- next() fetches the **next element**.
- When elements are exhausted, StopIteration is raised.

🏗️ **Creating Custom Iterators**

You can create your **own iterator** by defining a class with two methods:

1. __iter__() → Returns the iterator object itself.
2. __next__() → Returns the next value and raises StopIteration when done.

📝 **Example: Custom Iterator for Even Numbers**

```
class EvenNumbers:
  def __init__(self, limit):
    self.limit = limit
    self.num = 0

  def __iter__(self):
    return self  # Iterator object returns itself
```

```
    def __next__(self):
        if self.num > self.limit:
            raise StopIteration
        current = self.num
        self.num += 2
        return current

# Using the custom iterator
evens = EvenNumbers(10)
for n in evens:
    print(n)
```

Output:

```
0
2
4
6
8
10
```

Explanation:

- EvenNumbers is a custom iterator that generates **even numbers up to a limit**.
- for loop internally calls iter() and next() automatically.

Functions and Methods

Q20. Defining and calling functions in Python.

A. 🐍 Defining and Calling Functions in Python

💡 What is a Function?

A **function** is a **block of reusable code** that performs a specific task.
Functions help:

- **Avoid repetition** of code.

- **Organize** code for better readability.
- **Break complex problems** into smaller, manageable pieces.

1️⃣ Defining a Function

- Use the def keyword followed by the **function name** and parentheses ().
- The **code block** is indented.

Syntax:

```
def function_name(parameters):
    # code block
    return value  # optional
```

Example:

```
def greet():
    print("Hello, World!")
```

2️⃣ Calling a Function

- To **execute** a function, write its **name followed by parentheses**.

```
greet()  # Output: Hello, World!
```

3️⃣ Function with Parameters

- Functions can accept **input values** called **parameters**.
- These allow functions to work with **different data** each time they are called.

Example:

```
def greet_user(name):
    print(f"Hello, {name}!")


greet_user("Alice")  # Output: Hello, Alice!
greet_user("Bob")    # Output: Hello, Bob!
```

4️⃣ Function with Return Value

- Functions can **return a value** using the return keyword.
- The returned value can be **stored in a variable**.

Example:

```
def add_numbers(a, b):
    return a + b

result = add_numbers(5, 3)
print(result)  # Output: 8
```

Q21. Function arguments (positional, keyword, default).

A. Function Arguments in Python

When you define a function, you can pass **values to it** called **arguments**. Python supports **positional, keyword, and default arguments** to make functions flexible.

---

1 Positional Arguments

- The **values are assigned to parameters based on their position**.
- Order matters: the first value goes to the first parameter, the second to the second, and so on.

Example:

```
def greet(name, age):
    print(f"Hello {name}, you are {age} years old.")

greet("Alice", 25)  # Output: Hello Alice, you are 25 years old.
greet(25, "Alice")  # Output: Hello 25, you are Alice years old. ❌
```

Key Point: Always pass arguments in the **same order as parameters**.

---

2 Keyword Arguments

- Arguments are passed as **key=value**, so order **doesn't matter**.

Example:

```
def greet(name, age):
    print(f"Hello {name}, you are {age} years old.")

greet(age=25, name="Alice")  # Output: Hello Alice, you are 25 years old.
```

Key Point: Makes the code **more readable** and **order-independent**.

3️⃣ Default Arguments

- You can **assign default values** to parameters.
- If no value is passed, the function uses the **default value**.

Example:

```
def greet(name, age=18):
    print(f"Hello {name}, you are {age} years old.")


greet("Alice")      # Output: Hello Alice, you are 18 years old.
greet("Bob", 25)    # Output: Hello Bob, you are 25 years old.
```

Key Point: Default arguments must **come after non-default arguments**.

🔄 Combination Example

You can combine **positional, keyword, and default arguments**:

```
def info(name, age=18, country="USA"):
    print(f"{name}, {age} years old, from {country}")


info("Alice")                 # Positional with default values
info("Bob", country="Canada")     # Mix of positional and keyword
info(name="Charlie", age=30)      # Keyword arguments
```

Output:

Alice, 18 years old, from USA

Bob, 18 years old, from Canada

Charlie, 30 years old, from USA

Q22. Scope of variables in Python.

A. 🐍 Scope of Variables in Python

💡 What is Variable Scope?

Scope refers to the **region of the program where a variable is recognized and can be accessed**.
Python has **four types of variable scope**, often remembered by the acronym **LEGB**:

1️⃣ Local Scope (L)

- Variables created **inside a function** are **local to that function**.
- They **cannot be accessed outside** the function.

Example:

```
def my_func():
    x = 10  # Local variable
    print(x)

my_func()   # Output: 10
# print(x)  # ❌ Error: x is not defined outside the function
```

2️⃣ Enclosing (Nonlocal) Scope (E)

- Applies to **nested functions**.
- A variable in the **outer function** can be accessed by the **inner function**, but not vice versa.

Example:

```
def outer():
    y = 20
    def inner():
        print(y)  # Accessing enclosing variable
    inner()

outer()  # Output: 20
```

- To **modify** the enclosing variable inside the inner function, use nonlocal:

```
def outer():
    y = 20
    def inner():
        nonlocal y
        y += 5
        print(y)
    inner()
```

```
outer()  # Output: 25
```

3️⃣ Global Scope (G)

- Variables defined **outside all functions** have **global scope**.
- They can be accessed **anywhere in the program**.

Example:

```
z = 50  # Global variable

def my_func():
    print(z)

my_func()   # Output: 50
print(z)    # Output: 50
```

- To **modify a global variable inside a function**, use the global keyword:

```
a = 10

def update():
    global a
    a += 5

update()
print(a)  # Output: 15
```

4️⃣ Built-in Scope (B)

- Python has **built-in variables and functions** like print(), len(), range() etc.
- These are available **anywhere in the program** without defining them.

```
print(len("Python"))  # len() is a built-in function
```

Q23. Built-in methods for strings, lists, etc.

A. Built-in Methods in Python

Python provides **many built-in methods** for **strings, lists, tuples, sets, and dictionaries**. These methods **help perform common operations easily**.

---

1 String Methods (str)

Strings have **methods for manipulation, searching, and formatting**.

Method	Description	Example
.upper()	Converts to uppercase	"hello".upper() → "HELLO"
.lower()	Converts to lowercase	"HELLO".lower() → "hello"
.strip()	Removes whitespace from ends	" hi ".strip() → "hi"
.replace(old, new)	Replaces substring	"cat".replace("c","b") → "bat"
.split(sep)	Splits string into a list	"a,b,c".split(",") → ['a','b','c']
.join(iterable)	Joins iterable elements into string	"-".join(['a','b','c']) → "a-b-c"
.find(sub)	Returns index of substring	"hello".find("e") → 1
.startswith(sub)	Checks start of string	"hello".startswith("h") → True
.endswith(sub)	Checks end of string	"hello".endswith("o") → True

---

2 List Methods (list)

Lists are **mutable sequences**. Methods help **add, remove, sort, and search items**.

Method	Description	Example
.append(x)	Add element at end	[1,2].append(3) → [1,2,3]
.extend(iterable)	Add multiple elements	[1,2].extend([3,4]) → [1,2,3,4]
.insert(i,x)	Insert element at index	[1,3].insert(1,2) → [1,2,3]
.remove(x)	Remove first occurrence	[1,2,3].remove(2) → [1,3]
.pop(i)	Remove and return element	[1,2,3].pop() → 3
.index(x)	Return index of first occurrence	[1,2,3].index(2) → 1
.count(x)	Count occurrences	[1,2,2].count(2) → 2
.sort()	Sort list	[3,1,2].sort() → [1,2,3]
.reverse()	Reverse list	[1,2,3].reverse() → [3,2,1]

Method	Description	Example
.copy()	Return shallow copy	[1,2].copy() → [1,2]
.clear()	Remove all elements	[1,2].clear() → []

---

3 Tuple Methods (tuple)

Tuples are **immutable**, so they have **few methods**.

Method	Description	Example
.count(x)	Count occurrences	(1,2,2).count(2) → 2
.index(x)	Return index of first occurrence	(1,2,3).index(2) → 1

---

4 Set Methods (set)

Sets are **unordered collections of unique elements**.

Method	Description	Example
.add(x)	Add element	{1,2}.add(3) → {1,2,3}
.remove(x)	Remove element (error if not exists)	{1,2,3}.remove(2) → {1,3}
.discard(x)	Remove element if exists	{1,2,3}.discard(4) → {1,2,3}
.pop()	Remove and return arbitrary element	{1,2}.pop() → 1
.union(s)	Return union	{1,2}.union({2,3}) → {1,2,3}
.intersection(s)	Return intersection	{1,2}.intersection({2,3}) → {2}
.difference(s)	Return difference	{1,2}.difference({2,3}) → {1}
.clear()	Remove all elements	{1,2}.clear() → set()

---

5 Dictionary Methods (dict)

Dictionaries store **key–value pairs**.

Method	Description	Example
.keys()	Return all keys	{"a":1}.keys() → dict_keys(['a'])
.values()	Return all values	{"a":1}.values() → dict_values([1])
.items()	Return key-value pairs	{"a":1}.items() → dict_items([('a',1)])

Method	Description	Example
.get(k, default)	Get value for key	{"a":1}.get("b", 0) → 0
.update(d)	Merge another dict	{"a":1}.update({"b":2}) → {"a":1,"b":2}
.pop(k)	Remove key and return value	{"a":1}.pop("a") → 1
.clear()	Remove all items	{"a":1}.clear() → {}

Control Statements (Break, Continue, Pass)

Q24. Understanding the role of break, continue, and pass in Python loops.

A. Role of break, continue, and pass in Python Loops

These **special statements** help **control the flow** of loops in Python.

---

1 break

- **Stops the loop immediately**, even if the loop condition is still True.
- Often used to **exit early** when a certain condition is met.

Example:

```
for i in range(1, 6):
    if i == 3:
        break
    print(i)
```

Output:

```
1
2
```

Explanation: Loop stops when i == 3.

---

2 continue

- **Skips the current iteration** and moves to the **next iteration** of the loop.
- Useful to **ignore certain cases** but continue looping.

Example:

```
for i in range(1, 6):
```

```
    if i == 3:
        continue
    print(i)
```

Output:

1

2

4

5

Explanation: Iteration for i == 3 is skipped.

3️⃣ pass

- Does **nothing**; it's a **placeholder**.
- Useful when **a statement is required syntactically**, but you don't want any action yet.

Example:

```
for i in range(1, 6):
    if i == 3:
        pass  # Do nothing for now
    print(i)
```

Output:

1

2

3

4

5

Explanation: Loop runs normally; pass has no effect.

String Manipulation

Q25. Understanding how to access and manipulate strings.

A. 🐍 Accessing and Manipulating Strings in Python

Strings in Python are **sequences of characters**. You can **access, modify, and perform operations** on strings easily.

1️⃣ Accessing Characters in a String

- Strings support **indexing** (like lists).
- Index starts at **0** for the first character.
- Negative index -1 refers to the **last character**.

Example:

```
text = "Python"

print(text[0])   # Output: P
print(text[3])   # Output: h
print(text[-1])  # Output: n
```

2️⃣ Slicing Strings

- Slicing extracts **substrings** using [start:end:step].
- start → starting index (inclusive)
- end → ending index (exclusive)
- step → step size (optional)

Example:

```
text = "PythonProgramming"

print(text[0:6])    # Output: Python
print(text[6:])     # Output: Programming
print(text[:6])     # Output: Python
print(text[::2])    # Output: PtoPormig (every 2nd character)
print(text[::-1])   # Output: gnimmargorPmhtyP (reverse string)
```

3️⃣ String Concatenation

- Combine strings using + or .join()

```
first = "Hello"
second = "World"
```

```
print(first + " " + second)  # Output: Hello World
print(" ".join([first, second]))  # Output: Hello World
```

---

4 String Repetition

- Multiply a string using *

```
print("Hi! " * 3)  # Output: Hi! Hi! Hi!
```

---

5 String Methods for Manipulation

Some common **string methods**:

Method	Description	Example
.upper()	Convert to uppercase	"hello".upper() → "HELLO"
.lower()	Convert to lowercase	"HELLO".lower() → "hello"
.strip()	Remove whitespace	" hi ".strip() → "hi"
.replace(old,new)	Replace substring	"cat".replace("c","b") → "bat"
.split(sep)	Split into list	"a,b,c".split(",") → ['a','b','c']
.join(iterable)	Join list into string	"-".join(['a','b','c']) → "a-b-c"
.find(sub)	Index of substring	"hello".find("e") → 1
.startswith(sub)	Check start	"hello".startswith("h") → True
.endswith(sub)	Check end	"hello".endswith("o") → True

---

6 String Formatting

- Use **f-strings**, .format(), or % operator for inserting variables.

```
name = "Alice"
age = 25

# f-string (Python 3.6+)
print(f"My name is {name} and I am {age} years old.")

# format()
```

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


# % operator
print("My name is %s and I am %d years old." % (name, age))
```

Q26. How do you manage privileges using these commands?

A. Basic String Operations in Python

Strings in Python are **sequences of characters**, and you can perform several basic operations on them.

---

1 Concatenation

- **Combine two or more strings** using the + operator.

Example:

```
str1 = "Hello"
str2 = "World"
result = str1 + " " + str2
print(result)
```

Output:

```
Hello World
```

---

2 Repetition

- **Repeat a string** using the * operator.

Example:

```
greet = "Hi! "
print(greet * 3)
```

Output:

```
Hi! Hi! Hi!
```

---

3 String Methods

Python provides **built-in methods** to manipulate strings.

Method	Description	Example
.upper()	Convert all characters to uppercase	"hello".upper() → "HELLO"
.lower()	Convert all characters to lowercase	"HELLO".lower() → "hello"
.strip()	Remove leading and trailing whitespace	" hi ".strip() → "hi"
.replace(old, new)	Replace a substring	"cat".replace("c","b") → "bat"
.split(sep)	Split string into a list	"a,b,c".split(",") → ['a','b','c']
.join(iterable)	Join list of strings	"-".join(['a','b','c']) → "a-b-c"
.find(sub)	Find index of substring	"hello".find("e") → 1
.startswith(sub)	Check start of string	"hello".startswith("h") → True
.endswith(sub)	Check end of string	"hello".endswith("o") → True

---

4️⃣ **Examples of Using Methods**

```
text = "  python programming  "

print(text.upper())       # "  PYTHON PROGRAMMING  "
print(text.lower())       # "  python programming  "
print(text.strip())       # "python programming"
print(text.replace("python", "Java"))  # "  Java programming  "
print(text.split())       # ['python', 'programming']
```

Q27. String slicing.

A. 🐍 String Slicing in Python

💡 **What is String Slicing?**

Slicing allows you to **extract a part of a string** (substring) by specifying **start and end positions**, and optionally a **step**.

- Syntax:

  string[start:end:step]

- **start** → index to start (inclusive)
- **end** → index to end (exclusive)
- **step** → number of steps to jump (optional, default is 1)

---

1 Basic Slicing

```
text = "PythonProgramming"

print(text[0:6])   # Output: Python (index 0 to 5)
print(text[6:])    # Output: Programming (from index 6 to end)
print(text[:6])    # Output: Python (from start to index 5)
```

---

2 Slicing with Step

```
text = "PythonProgramming"

print(text[::2])   # Output: PtoPormig (every 2nd character)
print(text[1::2])  # Output: yhnrgamn (every 2nd character starting from index 1)
```

---

3 Negative Indexing

- Negative index counts **from the end** (-1 is last character).

```
text = "Python"

print(text[-1])     # Output: n (last character)
print(text[-6:-1])  # Output: Pytho (from index -6 to -2)
print(text[::-1])   # Output: nohtyP (reverses the string)
```

---

4 Examples of Useful Slices

```
text = "PythonProgramming"

print(text[:])     # Entire string → PythonProgramming
print(text[3:10])  # From index 3 to 9 → honProg
print(text[::3])   # Every 3rd character → PhPorm
print(text[::-2])  # Every 2nd character in reverse → gnmrPyth
```

Transaction Control Language (TCL)

Q28. How functional programming works in Python.

A. Functional Programming in Python

What is Functional Programming (FP)?

Functional programming is a **programming paradigm** where **functions are treated as first-class objects**.
Key principles of FP include:

1. **Immutability:** Data should not be modified (avoid changing variables).
2. **Pure functions:** Functions always produce the same output for the same input and have **no side effects**.
3. **Functions as first-class objects:** Functions can be **passed as arguments, returned from other functions, and assigned to variables**.
4. **Higher-order functions:** Functions that **take other functions as input** or **return functions**.

Python supports functional programming alongside **object-oriented** and **imperative programming**.

---

1 Pure Functions

A **pure function** produces the same output for the same input and has **no side effects**.

```
def add(a, b):
    return a + b


print(add(5, 3))  # Output: 8
print(add(5, 3))  # Output: 8 (always same)
```

---

2 First-Class Functions

Functions can be **assigned to variables**, **passed as arguments**, or **returned from other functions**.

```
def greet(name):
    return f"Hello, {name}!"


say_hello = greet  # Assign function to a variable
print(say_hello("Alice"))  # Output: Hello, Alice!
```

3️⃣ Higher-Order Functions

Functions that **take functions as input** or **return functions**.

Example: Using map(), filter(), reduce()

```
# map() → applies function to each element
nums = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, nums))
print(squared)  # Output: [1, 4, 9, 16, 25]

# filter() → selects elements based on condition
even_nums = list(filter(lambda x: x % 2 == 0, nums))
print(even_nums)  # Output: [2, 4]

# reduce() → cumulative computation (from functools import reduce)
from functools import reduce
sum_nums = reduce(lambda x, y: x + y, nums)
print(sum_nums)  # Output: 15
```

4️⃣ Anonymous Functions (Lambda)

- **Lambda functions** are **small, unnamed functions** used in functional programming.

```
square = lambda x: x**2
print(square(5))  # Output: 25
```

5️⃣ Advantages of Functional Programming in Python

- Promotes **readable and modular code**.
- Makes code **less prone to bugs** because of immutability and pure functions.
- Works well with **data transformations**, pipelines, and higher-order functions.

Q29. Using map(), reduce(), and filter() functions for processing data.

A. Using map(), filter(), and reduce()

These are **functional programming tools** in Python used to **process data in a clean and concise way**.

1️⃣ map()

- **Purpose:** Apply a function to **each item** of an iterable (like list, tuple) and return a new iterable.

Syntax:

```
map(function, iterable)
```

Example:

```
nums = [1, 2, 3, 4, 5]


# Square each number
squared = list(map(lambda x: x**2, nums))
print(squared)  # Output: [1, 4, 9, 16, 25]
```

Key Point: map() returns a **map object**, so convert to list or tuple if needed.

2️⃣ filter()

- **Purpose:** Select elements from an iterable that **meet a condition**.

Syntax:

```
filter(function, iterable)
```

Example:

```
nums = [1, 2, 3, 4, 5]


# Keep only even numbers
even_nums = list(filter(lambda x: x % 2 == 0, nums))
print(even_nums)  # Output: [2, 4]
```

Key Point: filter() also returns a **filter object**, convert to list for viewing.

3️⃣ reduce()

- **Purpose:** Apply a function **cumulatively** to items of an iterable, reducing it to a single value.

Syntax:

```
from functools import reduce
reduce(function, iterable)
```

Example:

```
from functools import reduce


nums = [1, 2, 3, 4, 5]


# Sum all numbers
sum_nums = reduce(lambda x, y: x + y, nums)
print(sum_nums)  # Output: 15


# Multiply all numbers
product = reduce(lambda x, y: x * y, nums)
print(product)  # Output: 120
```

Key Point: reduce() is useful for **cumulative operations** like sum, product, max, etc.

Q30. Introduction to closures and decorators.

A. 🧬 Closures and Decorators in Python

Python supports **advanced functional programming features** like **closures** and **decorators**, which are useful for **writing cleaner and reusable code**.

1️⃣ Closures

💡 What is a Closure?

A **closure** is a **function that remembers the environment in which it was created**, even if it is called outside that scope.

- **Inner function** remembers the **variables from outer function**.
- Useful for **encapsulation and maintaining state**.

Example:

```
def outer_func(x):
    def inner_func(y):
        return x + y  # inner function remembers x
    return inner_func
```

```
closure = outer_func(10)  # outer_func returns inner_func
print(closure(5))        # Output: 15
```

Explanation:

- inner_func **remembers x=10** even after outer_func has finished execution.
- This is a **closure**.

2️⃣ Decorators

💡 What is a Decorator?

A **decorator** is a **function that takes another function as input**, **adds extra functionality**, and **returns a new function**.

- Often used for **logging, authentication, timing, or preprocessing**.

Basic Syntax:

```
def decorator(func):
    def wrapper():
        print("Before function call")
        func()
        print("After function call")
    return wrapper
```

Example:

```
def say_hello():
    print("Hello!")

# Apply decorator
decorated = decorator(say_hello)
decorated()
```

Output:

```
Before function call
Hello!
After function call
```

Using @ Syntax (Pythonic way):

```
@decorator
```

```
def say_hello():
    print("Hello!")


say_hello()
```

Explanation:

- @decorator is a shorthand for say_hello = decorator(say_hello).
- It **enhances the original function without modifying it**.

3️⃣ Key Points

Concept	Description
Closure	Inner function remembers variables from outer function even after outer function finishes.
Decorator	Function that **adds functionality** to another function **without changing its code**.
Use Case	Closures → encapsulate state; Decorators → logging, timing, validation, etc.