Module 6: Python Fundamentals

Introduction to Python

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

Ans: Python is a versatile, high-level programming language known for its simplicity and readability. Created by Guido van Rossum and first released in 1991, Python has become one of the most popular programming languages in the world. It is widely used in various domains, including web development, data science, artificial intelligence, scientific computing, automation, and more.

Key Features of Python

1. **Simple and Easy to Learn:**
   - Python's syntax is designed to be intuitive and resembles the English language, making it beginner-friendly.
   - Its clean and readable code structure reduces the cost of program maintenance.
2. **High-Level Language:**
   - Python abstracts low-level details like memory management, allowing developers to focus on solving problems rather than worrying about system-level operations.
3. **Interpreted Language:**
   - Python code is executed line by line by an interpreter, which makes debugging easier and allows for rapid development and testing.
   - Unlike compiled languages, Python does not need to be compiled before execution.
4. **Dynamically Typed:**
   - Variables in Python do not require explicit declaration of their data type. The type is inferred at runtime, making the language flexible and concise.
5. **Cross-Platform Compatibility:**
   - Python is portable and runs on various platforms, including Windows, macOS, Linux, and more, without requiring changes to the code.
6. **Extensive Standard Library:**
   - Python comes with a rich collection of modules and libraries that provide pre-written code for tasks like file handling, web scraping, database interaction, and more.
7. **Open Source and Community-Driven:**
   - Python is free to use and distribute, and its large, active community contributes to its continuous improvement and support.
8. **Supports Multiple Programming Paradigms:**
   - Python supports procedural, object-oriented, and functional programming styles, giving developers flexibility in how they approach problems.
9. **Integration Capabilities:**

- Python can easily integrate with other languages like C, C++, and Java, as well as with frameworks and tools for various applications.

10. **Scalability:**
    - Python is suitable for both small scripts and large-scale applications, making it a versatile choice for developers.

Example of Python Code

Here's a simple example to demonstrate Python's readability and simplicity:

```
# A simple program to greet the user
name = input("Enter your name: ")
print(f"Hello, {name}! Welcome to Python.")
```

Why Learn Python?

- **Beginner-Friendly:** Ideal for those new to programming.
- **Versatile:** Used in a wide range of applications.
- **High Demand:** Python skills are highly sought after in the job market.
- **Rapid Development:** Enables quick prototyping and development.

Whether you're a beginner or an experienced developer, Python's simplicity and power make it an excellent choice for a wide range of projects.

(Q2) History and evolution of Python.

Ans: Python, one of the most popular and widely used programming languages, has a rich history and has undergone significant evolution since its inception. Here's a detailed look at its journey:

Origins and Early Development (1980s - 1990s)

- **1980s:** Python was conceived in the late 1980s by **Guido van Rossum**, a Dutch programmer working at **Centrum Wiskunde & Informatica (CWI)** in the Netherlands. He aimed to create a successor to the ABC language, addressing its limitations while keeping its strengths.
- **1989:** During the Christmas holidays, Guido van Rossum started working on Python as a hobby project.
- **1991 (Python 0.9.0):** The first official release of Python was made available, introducing key features like exception handling, functions, and core data types (lists, dictionaries, strings).
- **1994 (Python 1.0):** Python 1.0 was officially released, marking its first stable version with features like modules, exception handling, and dynamic typing.

Growth and Maturity (2000s)

- **2000 (Python 2.0):** Python 2 was released with major improvements:

    - List comprehensions
    - Garbage collection using reference counting
    - Unicode support
    - However, Python 2 introduced some design issues that led to the decision to phase it out.
- **2008 (Python 3.0):** A major milestone in Python's evolution.
    - **Not backward compatible** with Python 2
    - Improved **integer division** handling
    - Better Unicode support
    - More consistent and efficient syntax

Modern Era and Popularity (2010s - Present)

- **2010s:** Python gained massive popularity in web development, scientific computing, AI, and automation.
- **2018:** Python became the most popular language in data science and machine learning, thanks to libraries like NumPy, Pandas, TensorFlow, and Scikit-learn.
- **2020:** Python 2 reached its **end of life** on January 1, 2020, pushing all developers to migrate to Python 3.
- **2023-Present:** Python continues to evolve with versions like Python 3.11 and 3.12, improving performance (faster execution) and memory efficiency.

Key Factors Behind Python's Success

1. **Simplicity & Readability** – Easy to learn with a clean syntax.
2. **Versatility** – Used in web development, AI, automation, and more.
3. **Community Support** – Large, active community and extensive libraries.
4. **Cross-Platform Compatibility** – Runs on Windows, macOS, Linux.
5. **Integration Capabilities** – Works well with C, C++, and Java.

Python's future remains bright as it continues to power AI, machine learning, automation, and cloud computing.

(Q3) Advantages of using Python over other programming languages.

Ans: Python stands out among programming languages due to its simplicity, versatility, and extensive community support. Here are the key advantages of using Python over other languages:

1. Readability and Simplicity

- Python's syntax is **clean and easy to read**, making it beginner-friendly.
- Code resembles **natural English**, reducing the learning curve compared to C++, Java, or JavaScript.

2. Versatility and Flexibility

- Can be used for **web development, data science, AI, machine learning, automation, game development, cybersecurity, and more.**
- Works across multiple platforms **(Windows, macOS, Linux, etc.)** without modification.

3. Large Standard Library

- Comes with a **comprehensive standard library**, reducing the need for external dependencies.
- Modules for **file handling**, **networking**, **unit testing**, **databases**, **regular expressions**, **and more** are built-in.

4. Strong Community and Support

- Python has a vast **global community** with extensive documentation, tutorials, and third-party resources.
- **Quick troubleshooting** through forums like Stack Overflow, GitHub, and Python's official website.

5. Extensive Third-Party Libraries & Frameworks

- **Data Science & AI:** NumPy, Pandas, TensorFlow, PyTorch, Scikit-learn
- **Web Development:** Django, Flask, FastAPI
- **Automation & Scripting:** Selenium, PyAutoGUI
- **Cybersecurity:** Scapy, PyCrypto
- **Game Development:** Pygame, Panda3D

6. Cross-Platform and Open Source

- Python is **free and open source**, making it widely accessible.
- Compatible across various operating systems, enabling seamless development.

7. High-Level Language with Dynamic Typing

- **No need for manual memory management** (unlike C or C++).
- **Dynamic typing** allows more flexibility, making coding faster and more efficient.

8. Integration Capabilities

- Can easily integrate with **C**, **C++**, **Java**, **and other languages**.
- Supports **APIs**, **web services**, **and cloud computing** with minimal effort.

9. Productivity and Rapid Development

- Ideal for **prototyping and MVP (Minimum Viable Product) development**.
- Requires fewer lines of code compared to Java or C++.

10. Growing Demand and Career Opportunities

- Python is the **most in-demand language** in AI, data science, web development, and automation.
- Companies like **Google**, **Facebook**, **Netflix**, **NASA**, **and Spotify** use Python extensively.

Conclusion

Python's simplicity, powerful libraries, and cross-domain applications make it an excellent choice for both beginners and experienced developers. Whether for **automation, AI, web apps, or data science,** Python remains a top programming language.

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

Ans: To start coding in Python, you need to install Python itself and set up an **Integrated Development Environment (IDE).** Below are the steps to install Python and configure popular IDEs like **Anaconda, PyCharm, and VS Code.**

Step 1: Install Python

Download and Install Python

1. **Visit the official website** → Python.org
2. **Download the latest version** for Windows, macOS, or Linux.
3. **Run the installer** and ensure you check **"Add Python to PATH"** before installing.
4. **Verify installation** by opening a terminal (Command Prompt/PowerShell on Windows or Terminal on macOS/Linux) and typing:

   ```
   python --version
   ```

   or

   ```
   python3 --version
   ```

Step 2: Set Up a Development Environment

Option 1: Anaconda (Best for Data Science & AI)

Anaconda is a Python distribution that comes with pre-installed **libraries** (NumPy, Pandas, TensorFlow, etc.) and **Jupyter Notebook** for interactive coding.

1. **Download Anaconda** → Anaconda.com
2. **Install Anaconda** by following the setup instructions.
3. **Open Anaconda Navigator** and launch **Jupyter Notebook, Spyder, or VS Code.**
4. **Test Installation** by opening Anaconda Prompt and typing:

   ```
   conda list
   ```

   This will display installed packages.

✓ **Best For:** Data science, machine learning, and large-scale scientific computing.

Option 2: PyCharm (Best for Software Development)

PyCharm is a full-featured Python IDE by JetBrains, great for **web development and large projects**.

1. **Download PyCharm** → JetBrains PyCharm
   - Choose **Community Edition (Free)** or **Professional (Paid)**.
2. **Install PyCharm** and open it.
3. **Create a new project** and select the installed Python interpreter.
4. **Write a sample script** and run it.

✅ **Best For:** Professional development, Django, Flask, and full-stack applications.

Option 3: VS Code (Lightweight & Versatile)

VS Code is a free and powerful **code editor** with Python support.

1. **Download VS Code** → code.visualstudio.com
2. **Install the Python Extension:**
   - Open VS Code
   - Go to **Extensions** (Ctrl+Shift+X)
   - Search for "Python" and install it.
3. **Select Python Interpreter:**
   - Press Ctrl+Shift+P, type **Python: Select Interpreter**, and choose your Python installation.
4. **Run Python code** by opening a terminal and executing:

```
python script.py
```

✅ **Best For:** General development, web apps, and automation.

Final Steps: Verify Everything is Working

Once you have installed Python and set up your preferred IDE, test it by running a simple script:

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

Save it as hello.py and run it in the terminal using:

```
python hello.py
```

or

```
python3 hello.py
```

Conclusion

- **Anaconda** → Best for **data science, AI, and Jupyter Notebook users.**
- **PyCharm** → Best for **software development and large projects.**
- **VS Code** → Best for **general use, lightweight, and multi-language support.**

Now you're all set to start coding in Python!

(Q5) Writing and executing your first Python program.

Ans: Once you've installed Python and set up your development environment, it's time to write and run your first Python script!

Step 1: Writing Your First Python Program

Create a simple program that prints a message.

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

This program tells Python to print "Hello, Python World!" to the screen.

Step 2: Running the Python Program

Method 1: Using the Command Line (Recommended for Beginners)

1. **Open Terminal/Command Prompt**
   - Windows: Press Win + R, type **cmd**, and hit Enter.
   - macOS/Linux: Open **Terminal** from Applications or use Ctrl + Alt + T.
2. **Navigate to the folder where your script is saved**
   If your script is in Documents, use:

   ```
   cd Documents
   ```

3. **Run the Python script**

   ```
   python hello.py
   ```

   or

   ```
   python3 hello.py
   ```

   (Use python3 if you're on macOS or Linux and python defaults to an older version.)

✅ **Expected Output:**

```
Hello, Python World!
```

Method 2: Using an IDE (PyCharm, VS Code, or Anaconda Jupyter Notebook)

In PyCharm

1. Open **PyCharm** and create a new Python file (hello.py).
2. Type your Python code (print("Hello, Python World!")).
3. Click **Run** or press Shift + F10.

In VS Code

1. Open **VS Code** and create a new Python file (hello.py).
2. Write the print statement inside the file.
3. Open the terminal inside VS Code (Ctrl + ~) and run:

   ```
   python hello.py
   ```

In Jupyter Notebook (Anaconda Users)

1. Open **Anaconda Navigator** and launch **Jupyter Notebook**.
2. Click **New → Python 3 Notebook**.
3. In a new cell, type:

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

4. Click **Run** (Shift + Enter).

Step 3: Understanding the Code

- print() is a **built-in Python function** that outputs text to the screen.
- "Hello, Python World!" is a **string** enclosed in **double quotes**.
- Running the script executes the print statement and displays the message.

2. Programming Style

(Q1)Understanding Python's PEP 8 guidelines.

Ans: Python Enhancement Proposal 8 (**PEP 8**) is the official **style guide for writing clean, readable, and consistent Python code**. It helps developers maintain uniform coding standards.

Why Follow PEP 8?

✅ Improves code readability and maintainability
✅ Makes collaboration easier in teams
✅ Helps avoid common coding mistakes

(Q2)Indentation, comments, and naming conventions in Python.

Ans:

1. Indentation in Python

Python uses indentation to define **blocks of code** (unlike other languages that use {} brackets).

Rules:

✔ Use **4 spaces per indentation level** (avoid tabs).
✔ Indentation is **mandatory** in Python; incorrect indentation causes errors.

Example (Correct Indentation ✅)

```
def greet():
    print("Hello, Python!")  # 4 spaces indentation
```

Bad Example ❌ (Incorrect Indentation)

```
def greet():
  print("Hello, Python!")  # Only 2 spaces (wrong)
```

2. Comments in Python

Comments help explain your code and make it **easier to understand.**

Types of Comments:

1. **Single-line comment:** Use # at the beginning.
2. **Multi-line comment (docstrings):** Use """ """ for longer explanations.

Example:

```
# This function prints a greeting message

def greet():
    """This function prints 'Hello, Python!'."""
    print("Hello, Python!")
```

✔ **Use comments wisely** – Don't over-comment obvious code.

3. Naming Conventions in Python

PEP 8 provides clear rules for **naming variables, functions, classes, and constants.**

✅**Recommended Naming Styles:**

Type	Naming Style	Example

Variables	snake_case	user_name = "Alice"
Functions	snake_case	def calculate_sum():
Classes	PascalCase	class EmployeeDetails:
Constants	UPPER_CASE	PI = 3.14159
Modules	lowercase_with_underscores	import my_module

Bad Examples ✗

```
def CalculateSum():  # ✗Use lowercase with underscores instead
    pass

class employee_details:  # ✗Class names should be PascalCase
    pass
```

(Q3)Writing readable and maintainable code.

Ans: Good code is **not just about working correctly**; it should also be **easy to read and maintain**.

Tips for Readability:

✔ **Use meaningful variable names:** Avoid x, y, z, use user_age, total_price.
✔ **Break long lines** (>79 characters) using line continuation (\ or parentheses).
✔ **Use white spaces** around operators (+, -, =).
✔ **Group related code** using blank lines.

Example:

```
# Good Example (Readable)
def calculate_discount(price, discount_rate):
    """Returns the final price after applying discount."""
    discounted_price = price - (price * discount_rate)
    return discounted_price

# Calling the function
final_price = calculate_discount(100, 0.1)
print(final_price)
```

✗**Bad Example (Hard to Read)**

```
python
CopyEdit
def calculate_discount(price,discount_rate):return price-
(price*discount_rate)
```

3. Core Python Concepts

(Q1)Understanding data types: integers, floats,strings, lists, tuples, dictionaries,sets.

Ans:

Understanding Data Types in Python

Python has several built-in **data types** used to store different kinds of data.

1. Numeric Types

Data Type	Description	Example
int	Whole numbers	x = 10
float	Decimal numbers	y = 3.14
complex	Complex numbers	z = 2 + 3j

```
a = 42          # Integer
b = 3.14        # Float
c = 2 + 3j      # Complex number
print(type(a), type(b), type(c))
```

2. String (str)

A sequence of characters enclosed in quotes (' ' or " ").

```
text = "Hello, Python!"
print(text[0])  # Output: H (strings are indexed)
print(text[:5]) # Output: Hello (slicing)
```

3. List (list) – Mutable (Changeable) Ordered Collection

- Uses [] brackets.
- Allows **duplicate** values.

```
fruits = ["apple", "banana", "cherry"]
fruits.append("orange")  # Add an item
print(fruits[1])  # Output: banana
```

4. Tuple (tuple) – Immutable (Unchangeable) Ordered Collection

- Uses () brackets.
- **Cannot modify** elements after creation.

```
colors = ("red", "green", "blue")
print(colors[0])  # Output: red
```

5. Dictionary (dict) – Key-Value Pairs

- Uses {} brackets.
- Keys must be unique.

```
person = {"name": "Alice", "age": 25}
print(person["name"])  # Output: Alice
person["age"] = 26     # Modify value
```

6. Set (set) – Unordered, Unique Values

- Uses {} but **no duplicate values.**

```
numbers = {1, 2, 3, 4, 4}  # Duplicate 4 is ignored
numbers.add(5)
print(numbers)  # Output: {1, 2, 3, 4, 5}
```

(Q2)Python variables and memory allocation.

Ans:

1. Variables in Python

- Variables store data in memory.
- No need to declare types explicitly (Python is **dynamically typed**).

```
x = 10  # Integer
y = "Python"  # String
z = 3.14  # Float
```

2. Memory Allocation in Python

- **Immutable types (int, float, str, tuple)** share memory if they have the same value.
- **Mutable types (list, dict, set)** store references in memory.

```
a = 10
b = 10
print(id(a), id(b))  # Same memory address (Immutable sharing)

lst1 = [1, 2, 3]
lst2 = [1, 2, 3]
print(id(lst1), id(lst2))  # Different memory addresses (Mutable)
```

(Q3)Python operators: arithmetic, comparison, logical, bitwise.\

Ans: Python has different types of operators for various operations.

1. Arithmetic Operators

Operator	Description	Example
+	Addition	5 + 3 = 8
-	Subtraction	5 - 3 = 2
*	Multiplication	5 * 3 = 15
/	Division	5 / 3 = 1.666
//	Floor Division	5 // 3 = 1
%	Modulus (Remainder)	5 % 3 = 2
**	Exponentiation	2 ** 3 = 8

```
a = 10
b = 3
print(a // b, a % b, a ** b)  # Output: 3 1 1000
```

2. Comparison (Relational) Operators

Operator	Description	Example
==	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	3 <= 5 (True)

```
x = 10
y = 5
print(x > y, x == y)  # Output: True False
```

3. Logical Operators

Operator	Description	Example

and	Returns True if both conditions are true	(5 > 3 and 10 > 5) → True
or	Returns True if at least one condition is true	(5 > 3 or 10 < 5) → True
not	Reverses the condition	not(5 > 3) → False

```
x = True
y = False
print(x and y)   # Output: False
print(x or y)    # Output: True
print(not x)     # Output: False
```

4. Bitwise Operators (Operate at the bit level)

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

```
a = 5  # 0b0101
b = 3  # 0b0011
print(a & b)  # Output: 1 (0b0001)
print(a | b)  # Output: 7 (0b0111)
print(a ^ b)  # Output: 6 (0b0110)
```

4. Conditional Statements

(Q1) Introduction to conditionalstatements: if, else, elif.

Ans: Conditional statements allow a program to **make decisions** based on conditions. Python provides three main conditional statements:

- if → Executes a block of code **only if** a condition is True.
- elif (else if) → Checks multiple conditions.
- else → Executes when none of the conditions are True.

1. The if Statement

The if statement **checks a condition** and executes a block of code **only if** the condition is True.

Syntax:

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

Example:

```
age = 18

if age >= 18:
    print("You are eligible to vote.")
```

✅ **Output:**

```
You are eligible to vote.
```

2. The if-else Statement

If the if condition is **False**, the else block executes.

Example:

```
age = 16

if age >= 18:
    print("You are eligible to vote.")
else:
    print("You are not eligible to vote.")
```

✅ **Output:**

```
You are not eligible to vote.
```

3. The elif (else if) Statement

Used when there are **multiple conditions** to check.

Example:

```
score = 85

if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
elif score >= 70:
    print("Grade: C")
else:
    print("Grade: F")
```

✅ **Output:**

```
Grade: B
```

(Q2) Nested if-else conditions.

Ans: When an if statement is **inside another if or else block**, it's called a **nested if statement.**

Example:

```
num = 10

if num > 0:
    print("Positive number")
    if num % 2 == 0:
        print("Even number")
    else:
        print("Odd number")
else:
    print("Negative number")
```

✅ **Output:**

```
Positive number
Even number
```

5. Using Logical Operators in Conditions

You can combine multiple conditions using **and, or, and not.**

Example (and and or):

```
python
CopyEdit
age = 25
income = 50000

if age > 18 and income > 30000:
    print("Eligible for a loan")
else:
    print("Not eligible")
```

✅ **Output:**

```
css
CopyEdit
Eligible for a loan
```

Example (not):

```
is_logged_in = False

if not is_logged_in:
    print("Please log in")
```

✅ **Output:**

```
Please log in
```

5. Looping (For, While)

(Q1) Introduction to for and while loops.

Ans: Loops are an essential part of programming that allow you to execute a block of code multiple times without repeating it manually. Python provides two main types of loops: for loops and while loops.

(Q2) How loops work in Python.

Ans:

1 for Loops

- Used when you need to iterate over a sequence (e.g., a list, tuple, string, or range).
- Automatically handles iteration through elements.

Example:

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

Output:

```
apple
banana
cherry
```

2 while Loops

- Used when you want to repeat an action **as long as a condition is true.**
- Requires a condition that must eventually become false to prevent infinite loops.

Example:

```
count = 1
while count <= 5:
    print("Iteration:", count)
    count += 1
```

Output:

```
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
Iteration: 5
```

(Q3) Using loops with collections (lists, tuples, etc.)

Ans: Python loops work seamlessly with collections like **lists, tuples, dictionaries, and sets.**

Looping through a List

```
numbers = [10, 20, 30, 40]
for num in numbers:
    print(num)
```

Looping through a Tuple

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

Looping through a Dictionary

```
person = {"name": "Alice", "age": 25, "city": "New York"}
for key, value in person.items():
    print(f"{key}: {value}")
```

Looping through a Set

```
unique_numbers = {1, 2, 3, 4, 5}
for num in unique_numbers:
    print(num)
```

6. Generators and Iterators

(Q1) Understanding how generators work in Python.

Ans: Generators are a special type of iterator that allow you to **generate values on the fly** instead of storing them in memory. They are useful when dealing with large datasets or when you need to iterate over a sequence **without creating a list in memory.**

How Generators Work

Generators are created using functions that include the yield keyword instead of return. When a function with yield is called, it does **not execute immediately;** instead, it returns a **generator object.**

Example of a Generator

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

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

If you call next(gen) again after the last value is yielded, it will raise a Stop Iteration error.

(Q2) Difference between yield and return.

Ans:

Feature	yield	return
Function Type	Creates a generator	Regular function
Execution	Suspends function execution and remembers state	Ends function execution completely
Return Type	Returns a generator object	Returns a single value
Use Case	Efficient for large data and streaming	Used for simple function results

Example of yield vs. return

```
def using_return():
    return 1  # Returns the value and exits

def using_yield():
    yield 1  # Suspends execution, can continue later

print(using_return())  # Output: 1
print(using_yield())   # Output: <generator object>
```

To get values from a generator, you must iterate using next() or a loop.

(Q3) Understanding iterators and creating custom iterators.

Ans:

1. Iterator

An **iterator** is an object that implements two methods:

- __iter__() – Returns the iterator object itself.
- __next__() – Returns the next value in the sequence.

Built-in iterators in Python include lists, tuples, and dictionaries:

```
my_list = [10, 20, 30]
it = iter(my_list)  # Get an iterator from a list

print(next(it))  # Output: 10
print(next(it))  # Output: 20
print(next(it))  # Output: 30
```

2. Creating a Custom Iterator

You can create a custom iterator by defining a class with __iter__() and __next__() methods.

```
class CountDown:
    def __init__(self, start):
        self.num = start

    def __iter__(self):
        return self  # The iterator object itself

    def __next__(self):
        if self.num <= 0:
            raise Stop Iteration  # Stops iteration when the condition is
met
        self.num -= 1
        return self.num + 1

counter = CountDown(5)
for num in counter:
    print(num)
```

Output:

```
5
4
3
2
1
```

7. Functions and Methods

(Q1) Defining and calling functions in Python.

Ans: Functions in Python are blocks of reusable code that perform a specific task. They help in making code modular and easier to maintain.

Defining a Function

A function is defined using the def keyword, followed by the function name and parentheses ().

```
def greet():
    print("Hello, welcome to Python!")

greet()  # Calling the function
```

Function with Parameters

Functions can take arguments to make them more dynamic.

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

```
greet("Alice")  # Output: Hello, Alice!
```

(Q2) Function arguments (positional, keyword, default).

Ans: Python supports several types of function arguments:

1. Positional Arguments

Values are passed in order, and their position matters.

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

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

2. Default Arguments

If an argument is not provided, it takes a default value.

```
def greet(name="Guest"):
    print(f"Hello, {name}!")

greet()          # Output: Hello, Guest!
greet("Alice")   # Output: Hello, Alice!
```

3. Keyword Arguments

Arguments can be passed with their parameter names.

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

introduce(age=25, name="John")
# Output: My name is John and I am 25 years old.
```

4. Variable-Length Arguments (*args and **kwargs)

- *args allows multiple positional arguments as a tuple.
- **kwargs allows multiple keyword arguments as a dictionary.

```
def sum_all(*args):
    return sum(args)

print(sum_all(1, 2, 3, 4, 5))  # Output: 15

def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_info(name="Alice", age=25, city="New York")
```

(Q3) Scope of variables in Python.

Ans: Variable scope determines where a variable can be accessed.

1. Local Scope

A variable inside a function is **local** and cannot be accessed outside.

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

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

2. Global Scope

A variable outside a function is **global** and can be accessed anywhere.

```
x = 50  # Global variable

def show():
    print(x)  # Accessible inside the function

show()
print(x)  # Accessible outside the function
```

3. Modifying Global Variables Inside a Function

To modify a global variable inside a function, use the global keyword.

```
x = 10

def modify():
    global x
    x = 20  # Modifies the global variable

modify()
print(x)  # Output: 20
```

4. Nonlocal Scope (For Nested Functions)

The nonlocal keyword allows modifying a variable in the enclosing function.

```
def outer():
    y = 5
    def inner():
        nonlocal y
        y = 10  # Modifies the enclosing variable
    inner()
    print(y)  # Output: 10

outer()
```

(Q4) Built-in methods for strings, lists, etc.

Ans: Python provides various built-in methods to manipulate data types.

String Methods

```
text = "hello world"
print(text.upper())      # HELLO WORLD
print(text.lower())      # hello world
print(text.title())      # Hello World
print(text.replace("hello", "hi"))  # hi world
print(text.split())      # ['hello', 'world']
```

List Methods

```
nums = [1, 2, 3, 4, 5]
nums.append(6)  # Adds element
nums.pop()      # Removes last element
nums.insert(1, 10)  # Inserts 10 at index 1
nums.sort()     # Sorts list
print(nums)  # Output: [1, 2, 3, 4, 10]
```

Dictionary Methods

```
person = {"name": "Alice", "age": 25}
print(person.keys())   # dict_keys(['name', 'age'])
print(person.values()) # dict_values(['Alice', 25])
print(person.items())  # dict_items([('name', 'Alice'), ('age', 25)])
```

Set Methods

```
nums = {1, 2, 3, 4}
nums.add(5)
nums.remove(3)
print(nums)  # Output: {1, 2, 4, 5}
```

8. Control Statements (Break, Continue, Pass)

(Q1) Understanding the role of break, continue, and pass in Python loops.

Ans: In Python, loops (for and while) can be controlled using **break**, **continue**, and **pass** statements. These help in modifying the flow of loop execution.

1. break Statement

The break statement is used to **exit the loop prematurely** when a condition is met.

Example: Stopping a loop when a specific value is found

```
for num in range(1, 10):
    if num == 5:
        print("Stopping at", num)
        break  # Exits the loop
    print(num)

# Output:
# 1
```

```
# 2
# 3
# 4
# Stopping at 5
```

✅ **Use Case:** When you need to terminate a loop as soon as a condition is met.

2. continue Statement

The continue statement **skips the rest of the current iteration** and moves to the next one.

Example: Skipping even numbers in a loop

```
for num in range(1, 6):
    if num % 2 == 0:
        continue  # Skips even numbers
    print(num)

# Output:
# 1
# 3
# 5
```

✅ **Use Case:** When you want to skip specific iterations but continue looping.

3. pass Statement

The pass statement is a **placeholder** and does nothing. It is used when a statement is required syntactically but no action is needed.

Example: Using pass inside a loop

```
for num in range(1, 6):
    if num == 3:
        pass  # Placeholder, does nothing
    print(num)

# Output:
# 1
# 2
# 3
# 4
# 5
```

✅ **Use Case:** When you have not yet implemented logic inside a loop but want to maintain the structure.

Summary of Differences

Statement	Function	Statement
break	Exits the loop completely.	break
continue	Skips the current iteration and moves to the next one.	continue
pass	Does nothing; acts as a placeholder.	pass

9. String Manipulation

(Q1)Understanding how to access and manipulate strings.

Ans: Strings in Python are sequences of characters enclosed in **single (')**, **double (")**, **or triple quotes (''' """)**.

```
text = "Hello, Python!"
```

(Q2)Basic operations: concatenation, repetition,string methods(upper(), lower(), etc.).

Ans:

1. Concatenation (+)

Joining two or more strings together.

```
first = "Hello"
second = "World"
result = first + " " + second
print(result)  # Output: Hello World
```

2. Repetition (*)

Repeating a string multiple times.

```
word = "Python "
print(word * 3)  # Output: Python Python Python
```

3. String Length (len())

Finding the length of a string.

```
text = "Python"
print(len(text))  # Output: 6
```

String Methods

Python provides built-in methods to manipulate strings.

1. Changing Case

```
text = "Hello Python"
print(text.upper())  # HELLO PYTHON
print(text.lower())  # hello python
print(text.title())  # Hello Python
print(text.capitalize())  # Hello python
```

2. Removing Whitespaces

```
text = "  Python  "
print(text.strip())  # "Python" (Removes spaces from both sides)
print(text.lstrip())  # "Python  " (Removes left spaces)
print(text.rstrip())  # "  Python" (Removes right spaces)
```

3. Replacing & Splitting

```
text = "I love Python"
print(text.replace("love", "like"))  # Output: I like Python

words = "apple,banana,orange"
print(words.split(","))  # Output: ['apple', 'banana', 'orange']
```

4. Checking Substrings

```
text = "Hello, Python"
print("Python" in text)  # True
print("Java" not in text)  # True
```

(Q1)String slicing.

Ans: Slicing allows you to extract parts of a string using string[start:end:step].

1. Basic Slicing

```
text = "Python"
print(text[0:4])  # Output: Pyth
print(text[:3])   # Output: Pyt (same as text[0:3])
print(text[2:])   # Output: thon (from index 2 to end)
```

2. Negative Indexing

```
text = "Python"
print(text[-3:])  # Output: hon
print(text[:-2])  # Output: Pyth
```

3. Step Slicing

```
text = "Python"
print(text[::2])  # Output: Pto (every second letter)
print(text[::-1])  # Output: nohtyP (reverses string)
```

10. Advanced Python (map(), reduce(), filter(), Closures and Decorators)

(Q1) How functional programming works in Python.

Ans: Functional programming is a programming paradigm that treats functions as **first-class citizens**, meaning functions can be assigned to variables, passed as arguments, and returned from other functions. It emphasizes **pure functions, immutability, and higher-order functions.**

(Q2) Using map(), reduce(), and filter() functions for processing data.

Ans: Python provides built-in **higher-order functions** like map(), filter(), and reduce() to apply functions to iterables efficiently.

1. map() – Transform Data

map() applies a function to **each item** in an iterable and returns a new iterable.

Example: Converting a list of numbers to their squares

```
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x ** 2, numbers))
print(squared)  # Output: [1, 4, 9, 16, 25]
```

2. filter() – Select Data

filter() applies a function to each element and **keeps only elements that return True.**

Example: Filtering even numbers from a list

```
numbers = [1, 2, 3, 4, 5, 6]
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(evens)  # Output: [2, 4, 6]
```

3. reduce() – Aggregate Data

reduce() **reduces** an iterable to a single value using a function. It's part of functools and is used for operations like sum, multiplication, etc.

Example: Summing up all numbers in a list

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]
total = reduce(lambda x, y: x + y, numbers)
print(total)  # Output: 15
```

(Q3) Introduction to closures and decorators.

Ans:

Closures in Python

A **closure** is a function defined inside another function that remembers the variables from the enclosing scope **even after the outer function has finished executing.**

Example of a Closure

```
def outer_function(msg):
    def inner_function():
        print(msg)  # msg is remembered even after outer_function is done
    return inner_function

greet = outer_function("Hello, Python!")
greet()  # Output: Hello, Python!
```

✅**Use Case:** Closures help in **data hiding** and maintaining **state across function calls.**

Decorators in Python

A **decorator** is a function that takes another function as input and enhances its behavior **without modifying the original function.**

Creating a Simple Decorator

```
def decorator_function(original_function):
    def wrapper_function():
        print("Wrapper executed before", original_function.__name__)
        return original_function()
    return wrapper_function

@decorator_function  # Using the decorator
def say_hello():
    print("Hello!")

say_hello()
```

Output:

```
Wrapper executed before say_hello
Hello!
```

✅**Use Case:** Decorators are used for **logging, authentication, caching, timing functions, etc.**

11. Python – Collections, functions and Modules

Accessing List

(Q1)Understanding how to create and access elements in a list.

Ans:

Creating a list

```
my_list = ["apple", "banana", "cherry", "date", "elderberry", "fig",
"grape"]
```

(Q2) Indexing in lists (positive and negative indexing).

Ans:

Accessing elements using positive indexing

```
print("First element (index 0):", my_list[0])

print("Third element (index 2):", my_list[2])
```

Accessing elements using negative indexing

```
print("Last element (index -1):", my_list[-1])

print("Second last element (index -2):", my_list[-2])
```

(Q3) Slicing a list: accessing a range of elements.

Ans:

```
print("Elements from index 1 to 4:", my_list[1:5]) # Index 5 is excluded

print("First three elements:", my_list[:3]) # From start to index 2

print("Elements from index 3 to end:", my_list[3:])

print("Full list using slicing:", my_list[:])

print("List in reverse order:", my_list[::-1])

# Accessing every second element
print("Every second element:", my_list[::2])

# Last 3 elements of the list
print("Last 3 elements:", my_list[-3:])
```

2. List Operations

(Q1) Common list operations: concatenation, repetition, membership.

Ans:

Creating two lists

```
list1 = ["apple", "banana", "cherry"]
```

```
list2 = ["date", "elderberry", "fig"]
```

1. List Concatenation (+ operator)

```
concatenated_list = list1 + list2

print("Concatenated List:", concatenated_list)
```

2. List Repetition (* operator)

```
repeated_list = list1 * 2

print("Repeated List:", repeated_list)
```

3. Membership Test (in operator)

```
print("Is 'banana' in list1?", "banana" in list1)

print("Is 'grape' in list1?", "grape" in list1)
```

(Q2)Understanding list methods like append(), insert(), remove(), pop().

Ans:

1. Using append() - Adds an element to the end

```
list1 = ["apple", "banana", "cherry"]

list1.append("grape")

print("List after append:", list1)
```

2. Using insert() - Inserts an element at a specific index

```
list1.insert(1, "blueberry")

print("List after insert at index 1:", list1)
```

3. Using remove() - Removes the first occurrence of a specified element

```
list1.remove("cherry")

print("List after removing 'cherry':", list1)
```

4. Using pop() - Removes and returns an element (default: last element)

```
popped_element = list1.pop() # Removes last element

print("Popped element:", popped_element)

print("List after pop:", list1)
```

 # Pop at a specific index

```
popped_index_element = list1.pop(2) # Removes element at index 2

print("Element popped from index 2:", popped_index_element)

print("List after popping index 2:", list1)
```

3. Working with Lists

(Q1)Iterating over a list using loops.

Ans: You can iterate over a list using a `for` loop or a `while` loop.

Example: Using a for Loop

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

print("Iterating using a for loop:")
for fruit in fruits:
    print(fruit)
```

Example: Using a while Loop

```
print("Iterating using a while loop:")
i = 0
while i < len(fruits):
        print(fruits[i])
        i += 1
```

(Q2)Sorting and reversing a list using sort(), sorted(), and reverse().

Ans:

Using sort() (Modifies the list in-place)

```
numbers = [5, 2, 9, 1, 5, 6]
numbers.sort()
print("Sorted list (ascending):", numbers)

numbers.sort(reverse=True)
print("Sorted list (descending):", numbers)
```

Using sorted() (Returns a new sorted list)

```
numbers = [3, 8, 1, 6, 0, 7]
sorted_numbers = sorted(numbers)
print("Original list:", numbers)
print("Sorted list:", sorted_numbers)

sorted_descending = sorted(numbers, reverse=True)
print("Sorted in descending order:", sorted_descending)
```

Using reverse() (Reverses the list in-place)

```
fruits = ["apple", "banana", "cherry", "date"]
fruits.reverse()
print("Reversed list:", fruits)
```

(Q3) Basic list manipulations: addition, deletion, updating, and slicing.

Ans:

1 Adding Elements

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

# Using append()
my_list.append(40)
print("After append:", my_list)

# Using insert()
my_list.insert(1, 15)
print("After insert at index 1:", my_list)

# Using extend()
my_list.extend([50, 60])
print("After extend:", my_list)
```

2 Deleting Elements

```
# Using remove()
my_list.remove(30)
print("After removing 30:", my_list)

# Using pop() - Removes last element
popped_element = my_list.pop()
print("Popped element:", popped_element)
print("After pop:", my_list)

# Using del
del my_list[1]  # Deletes the element at index 1
print("After deleting index 1:", my_list)

# Using clear() - Empties the list
my_list.clear()
print("After clear:", my_list)
```

3 Updating Elements

```
my_list = [10, 20, 30, 40]
my_list[2] = 35  # Updating index 2
print("Updated list:", my_list)
```

4 Slicing a List

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

print("Slice from index 2 to 5:", numbers[2:6])  # Excludes index 6
print("First 4 elements:", numbers[:4])
print("Elements from index 3 onwards:", numbers[3:])
print("Last 3 elements:", numbers[-3:])
print("Every second element:", numbers[::2])
print("List in reverse:", numbers[::-1])
```

4. Tuple

(Q1) Introduction to tuples, immutability.

Ans: A **tuple** in Python is an **ordered**, **immutable** collection of elements.

- **Immutable** means **you cannot modify (add, remove, or update) elements** once the tuple is created.
- Tuples are defined using **parentheses ()**, unlike lists, which use [].

Example: Creating a Tuple

```
# Creating a tuple
my_tuple = (1, 2, 3, "apple", "banana")
print("Tuple:", my_tuple)

# Attempting to modify a tuple (this will raise an error)
# my_tuple[1] = 100  # TypeError: 'tuple' object does not support item
assignment
```

Use Cases of Tuples:

- Used when data **should not be changed** (e.g., coordinates, database records).
- Faster than lists due to **immutability**.

(Q2) Creating and accessing elements in a tuple.

Ans:

Creating a Tuple

```
# Tuple with different data types
tuple1 = (10, 20, 30)
tuple2 = ("apple", "banana", "cherry")
tuple3 = (True, False, "Hello", 5.5)

# Single-element tuple (must include a comma!)
single_element_tuple = (5,)  # Without the comma, it's just an integer!
print("Single-element tuple:", single_element_tuple)
```

Accessing Tuple Elements

```
# Using positive indexing
print("First element:", tuple1[0])
print("Second element:", tuple2[1])
```

```
# Using negative indexing
print("Last element:", tuple3[-1])
print("Second last element:", tuple3[-2])
```

Tuple Slicing

```
numbers = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
print("Slice from index 2 to 5:", numbers[2:6])  # Elements from index 2 to
5
print("Last three elements:", numbers[-3:])
print("Every second element:", numbers[::2])
print("Tuple in reverse:", numbers[::-1])
```

(Q3) Basic operations with tuples: concatenation, repetition, membership.

Ans:

1 Concatenation (+)

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)

# Joining two tuples
concatenated_tuple = tuple1 + tuple2
print("Concatenated tuple:", concatenated_tuple)
```

2 Repetition (*)

```
my_tuple = ("Hello",) * 3  # Repeat tuple 3 times
print("Repeated tuple:", my_tuple)
```

3 Membership (in)

```
fruits = ("apple", "banana", "cherry")

print("Is 'banana' in tuple?", "banana" in fruits)  # True
print("Is 'grape' in tuple?", "grape" in fruits)  # False
```

5. Accessing Tuples

(Q1) Accessing tuple elements using positive and negative indexing.

Ans: Tuples in Python allow element access using **positive** and **negative** indexing:

- **Positive Indexing (0 to n-1)** → Starts from the left (0-based index).
- **Negative Indexing (-1 to -n)** → Starts from the right (-1 is the last element).

Example: Accessing Elements Using Indexing

```
# Creating a tuple
my_tuple = ("apple", "banana", "cherry", "date", "elderberry")

# Accessing using positive indexing
print("First element:", my_tuple[0])  # apple
```

```
print("Third element:", my_tuple[2])  # cherry

# Accessing using negative indexing
print("Last element:", my_tuple[-1])   # elderberry
print("Second last element:", my_tuple[-2])  # date
```

 Output:

```
First element: apple
Third element: cherry
Last element: elderberry
Second last element: date
```

(Q2) Slicing a tuple to access ranges of elements.

Ans: Tuple slicing allows you to extract parts of a tuple using the syntax:
`tuple[start:end:step]`

- **start** → Beginning index (inclusive)
- **end** → Ending index (exclusive)
- **step** → Skipping elements (default is 1)

Example: Tuple Slicing

```
numbers = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

print("Elements from index 2 to 5:", numbers[2:6])  # (2, 3, 4, 5)
print("First four elements:", numbers[:4])  # (0, 1, 2, 3)
print("Elements from index 3 to end:", numbers[3:])  # (3, 4, 5, 6, 7, 8,
9)
print("Last three elements:", numbers[-3:])  # (7, 8, 9)
print("Every second element:", numbers[::2])  # (0, 2, 4, 6, 8)
print("Tuple in reverse:", numbers[::-1])  # (9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
```

Output:

```
Elements from index 2 to 5: (2, 3, 4, 5)

First four elements: (0, 1, 2, 3)

Elements from index 3 to end: (3, 4, 5, 6, 7, 8, 9)

Last three elements: (7, 8, 9)

Every second element: (0, 2, 4, 6, 8)

Tuple in reverse: (9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
```

6. Dictionaries

(Q1) Introduction to dictionaries: key-value pairs.

Ans: A **dictionary** in Python is an **unordered**, **mutable** collection of **key-value pairs**.

- Defined using **curly braces {}.**
- Keys must be **unique** and **immutable** (strings, numbers, or tuples).
- Values can be **any data type** (strings, lists, other dictionaries, etc.).

Example: Creating a Dictionary

```
# Creating a dictionary
student = {
    "name": "Alice",
    "age": 20,
    "grade": "A",
    "subjects": ["Math", "Science"]
}

print("Student Dictionary:", student)
```

 Output:

```
Student Dictionary: {'name': 'Alice', 'age': 20, 'grade': 'A', 'subjects':
['Math', 'Science']}
```

(Q2) Accessing, adding, updating, and deleting dictionary elements.

Ans: ***1 Accessing Elements***

```
# Using keys
print("Name:", student["name"])  # Alice

# Using get() (avoids KeyError if key is missing)
print("Age:", student.get("age"))  # 20
print("GPA (non-existent key):", student.get("GPA", "Not Available"))  #
Default value
```

2 Adding Elements

```
# Adding a new key-value pair
student["GPA"] = 3.8
print("After Adding GPA:", student)
```

3 Updating Elements

```
# Updating an existing key
student["grade"] = "A+"
print("After Updating Grade:", student)
```

4 Deleting Elements

```
# Using del
del student["age"]
print("After Deleting Age:", student)

# Using pop() - Removes & returns the value
removed_value = student.pop("GPA")
print("Removed GPA:", removed_value)
print("After pop:", student)

# Using popitem() - Removes last inserted key-value pair (Python 3.7+)
student.popitem()
print("After popitem():", student)
```

```
# Using clear() - Removes all elements
student.clear()
print("After clear:", student)
```

(Q3) Dictionary methods like keys(), values(), and items().

Ans:

keys() - Get all keys

```
student = {"name": "Alice", "age": 20, "grade": "A"}
print("Keys:", student.keys())  # dict_keys(['name', 'age', 'grade'])
```

2 values() - Get all values

```
print("Values:", student.values())  # dict_values(['Alice', 20, 'A'])
```

3 items() - Get all key-value pairs as tuples

```
print("Items:", student.items())
# dict_items([('name', 'Alice'), ('age', 20), ('grade', 'A')])
```

7. Working with Dictionaries

(Q1) Iterating over a dictionary using loops.

Ans:

You can iterate over a dictionary using loops to access its keys, values, or both.

1 Iterating Over Keys

```
student = {
    "name": "Alice",
    "age": 20,
    "grade": "A"
}

print("Iterating over keys:")
for key in student:
    print(key)
```

2 Iterating Over Values

```
print("\nIterating over values:")
for value in student.values():
    print(value)
```

3 Iterating Over Key-Value Pairs

```
print("\nIterating over key-value pairs:")
for key, value in student.items():
    print(f"{key}: {value}")
```

Output:

```
Iterating over keys:
```

```
name
age
grade

Iterating over values:
Alice
20
A

Iterating over key-value pairs:
name: Alice
age: 20
grade: A
```

(Q2) Merging two lists into a dictionary using loops or zip().

Ans:

1 Using a Loop

You can iterate over both lists simultaneously and create key-value pairs using the zip() function or by manually looping through the lists.

```
keys = ["name", "age", "grade"]
values = ["Alice", 20, "A"]

# Using a loop to merge the lists into a dictionary
merged_dict = {}
for i in range(len(keys)):
    merged_dict[keys[i]] = values[i]

print("Merged Dictionary using loop:", merged_dict)
```

2 Using zip()

```
# Using zip() to combine lists into key-value pairs
merged_dict_zip = dict(zip(keys, values))
print("Merged Dictionary using zip():", merged_dict_zip)
```

Output:

```
Merged Dictionary using loop: {'name': 'Alice', 'age': 20, 'grade': 'A'}
Merged Dictionary using zip(): {'name': 'Alice', 'age': 20, 'grade': 'A'}
```

(Q3) Counting occurrences of characters in a string using dictionaries.

Ans: You can count how many times each character appears in a string by iterating through the string and updating the dictionary.

```
# String to count characters
text = "hello world"

# Initialize an empty dictionary
char_count = {}

# Iterating through the string
for char in text:
```

```
    if char in char_count:
        char_count[char] += 1
    else:
        char_count[char] = 1

print("Character occurrences:", char_count)
```

Output:

```
Character occurrences: {'h': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 1, 'w': 1, 'r': 1, 'd': 1}
```

8. Functions

(Q1)Defining functions in Python.

Ans: In Python, a function is defined using the def keyword, followed by the function name and parentheses. The function body is indented below the definition.

Syntax:

```
def function_name(parameters):
    # Function body
    # Code to execute
```

(Q2)Different types of functions: with/without parameters, with/without return values.

Ans: A function can be defined without any parameters and without returning any value. It simply performs an action when called.

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

# Calling the function
greet()
```

Output:

```
Hello, World!
```

(Q3)Anonymous functions (lambda functions).

Ans: You can pass parameters to a function to make it more dynamic. The function can then use these parameters in its execution, but it doesn't return any value.

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

# Calling the function with an argument
greet_person("Alice")
greet_person("Bob")
```

Output:

```
Hello, Alice!
```

```
Hello, Bob!
```

9. Modules

(Q1) Introduction to Python modules and importing modules.

Ans: A **module** in Python is a file containing Python code (functions, variables, classes, etc.). It allows you to organize code logically, reuse it, and avoid repetition.

What is a Module?

A module is simply a Python file with the extension .py. You can create your own modules, or you can use built-in ones (like math, random, etc.).

Importing Modules

To use a module, you can import it into your program using the import keyword.

```
import module_name  # Importing the whole module
```

Alternatively, you can import specific functions or objects from a module:

```
from module_name import function_name  # Import specific function
```

You can also give a module an alias to make it easier to refer to:

```
import module_name as alias  # Import with alias
```

(Q2) Standard library modules: math, random.

Ans:

1 math Module

The math module provides mathematical functions for tasks like square roots, trigonometry, logarithms, and more.

```
import math

# Using some functions from the math module
print("Square root of 16:", math.sqrt(16))  # 4.0
print("Pi:", math.pi)  # 3.141592653589793
print("Factorial of 5:", math.factorial(5))  # 120
```

2 random Module

The random module provides functions for generating random numbers or making random selections.

```
import random

# Random number generation
print("Random integer between 1 and 10:", random.randint(1, 10))  # e.g., 7
```

```
# Random float between 0 and 1
print("Random float between 0 and 1:", random.random())  # e.g., 0.657

# Randomly select an element from a list
fruits = ["apple", "banana", "cherry", "date"]
print("Random fruit:", random.choice(fruits))  # e.g., 'banana'

# Shuffle the list randomly
random.shuffle(fruits)
print("Shuffled list:", fruits)  # e.g., ['banana', 'cherry', 'apple', 'dat
```

(Q3) Creating custom modules.

Ans:

1 How to Create a Custom Module

A custom module is simply a Python file that contains functions, classes, or variables. For example, create a file named my_module.py.

Contents of my_module.py:

```
# Defining a function in the custom module
def greet(name):
    return f"Hello, {name}!"

# Defining a variable
pi = 3.14159
```

2 Importing Your Custom Module

Once you've created a module, you can import it into another script or the Python interpreter to use its functionality.

Using the custom module in another file (main.py):

```
import my_module  # Importing the custom module

# Using the function from the custom module
print(my_module.greet("Alice"))

# Using the variable from the custom module
print("Value of Pi:", my_module.pi)
```

Alternatively, you can import specific functions or variables from the module:

```
from my_module import greet, pi

print(greet("Bob"))
print("Value of Pi:", pi)
```