(1) Python Fundamentals

❖Introduction to Python

Q-1) Introduction to Python and its Features (simple, high-level, interpreted language).

Python is a simple **high-level, general-purpose, interpreted** programming language.

➢ Simple

Python syntax is clean, minimal, and easy to learn/read. For many tasks you write fewer lines than you might need in a lower-level language.

➢ High-level

Compared to low-level languages, a high-level language lets you write code with abstractions: you don’t deal with hardware-level details (memory addresses, registers, etc.). That makes development easier and faster.

➢ Interpreted

Python code is executed by an interpreter, line by line, at runtime. You don’t compile it to machine code first. This enables rapid development cycles, easier debugging, and interactive coding.

- **Large Standard Library**

  Python comes with a broad standard library as well as supports many third-party packages. This helps in tasks ranging from file I/O, web programming, data manipulation, GUI applications, and more — without reinventing the wheel.

- **Cross-Platform & Portable**

  Python runs on various operating systems (Windows, macOS, Linux, etc.) and generally the same code works across platforms.

- **Free & Open Source**

  Python is freely available and open-source, so anyone can download, use, and contribute to it.

Q-2) History and evolution of Python.

- Python was conceived in the **late 1980s** by Guido van Rossum at the Centrum Wiskunde & Informatica (CWI) in the Netherlands. Implementation started in **December 1989**.

- The very first public release was version **0.9.0 in 1991**. That early version already included key features like functions, exception handling, core data types (lists, dictionaries, strings), modules, and basic object-oriented programming.

- The official “1.0” release came in **1994** — marking Python’s first stable release and starting its growth in usage.

- A major milestone was the release of Python 2.0 in **October 2000**, which added important features like list comprehensions, better memory and garbage-collection support, and Unicode support — making Python more robust for real-world applications.

- Later, Python 3.0 was released on **3 December 2008** — a major redesign that was not fully backward-compatible, intended to clean up and modernize parts of the language.

- Since then, Python has continuously evolved through the 3.x series, gaining new language features, improving performance, and building a large ecosystem and user community.

Q-3) Advantages of using Python over other programming languages.

- **Simple, readable syntax**

  Python’s code looks closer to plain English than many other languages (no curly braces, minimal punctuation), which makes it easier to learn and understand.

- **Fewer lines of code / Less boilerplate**

  You can accomplish in Python what might take many more lines in languages like Java or C++. That leads to faster development and less room for simple errors.

- **Rapid development and prototyping**

  Because of its simplicity and expressiveness, Python is great for building prototypes, testing ideas, or quickly developing applications — helpful especially in startups or fast-changing projects.

- **Large ecosystem of libraries and frameworks**

  Python has many built-in and third-party libraries for web development, data analysis, machine learning, automation, etc. This lets you reuse existing tools instead of reinventing functionality from scratch.

- **Cross-platform and portable**

  Python runs on most operating systems (Windows, Linux, macOS etc.) with little or no change to the code, making it easier to write code once and deploy anywhere.

- **Flexible and versatile for many domains**

  Python supports multiple programming paradigms (object-oriented, procedural, functional) and is used for a wide variety of applications: web apps, data science, scripting/automation, AI/ML, scientific computing, and more.

- **Strong community support and open source**

  Being open source and backed by a large developer community means lots of documentation, shared libraries and frameworks, and continual improvements.

Q-4) Installing Python and setting up the development environment (VS Code).

- Install Python

- The simplest way is to go to the official Python website and download the latest stable Python 3 installer for your OS (Windows, macOS, Linux). Then run the installer. On Windows: make sure to check **"Add Python to PATH"** option so you can run Python from the command line.

- After installation, verify by opening a terminal/command prompt and typing something like:

```
python --version
```

- You should see a Python 3.x version printed.

- **Install & Setup an IDE**

VS Code

- Download and install VS Code from its official website.

- Once installed, open VS Code and install the "Python" extension (by Microsoft) - this gives syntax highlighting, linting, IntelliSense (autocomplete), code formatting, debugging support, etc.

- Then open (or create) a project folder, create a .py file and run code — you can run via terminal or VS Code’s Run command.
- You can also manage packages, debug code, and customize formatting / linting / project settings.

Q-5) Writing and executing your first Python program.

- **A text editor or IDE (or simply the built-in shell/IDLE that comes with Python) to write your code.**
  - **Writing “Hello, World!” — the code**
    - Open your editor and type this:

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

    - You can also use single quotes, like:

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

  - **Saving the script**
    - Save the file with a .py extension — e.g. hello.py.
  - **Running / Executing the program**
    - **Via terminal / command prompt**

      (1) Open your terminal (on Linux/Mac) or Command Prompt (on Windows).

      (2) Navigate (cd) to the folder where your hello.py is saved.

      (3) Run the command:

```
python hello.py
```

- **Via built-in shell / IDE (e.g. IDLE)**

  (1) Open IDLE (or another Python-friendly IDE).

  (2) Create a new file, write the print(“Hello, World”) line.

  (3) Save it (.py extension).

  (4) Run the program (in IDLE: Run → Run Module or press F5).

(2) Programming Style

Q-1) Understanding Python's PEP 8 guidelines.

- PEP 8, or Python Enhancement Proposal 8, is the official style guide for writing Python code. Its primary purpose is to enhance readability and consistency across the Python community, making code easier to understand, maintain, and collaborate on.

Key guidelines within PEP 8 include:

- **Indentation:**
  - Use 4 spaces per indentation level. Tabs are discouraged.
- **Line Length:**
  - Limit lines to a maximum of 79 characters to improve readability on various screen sizes.
- **Naming Conventions:**
  - Variables and functions: Use lowercase_with_underscores.
  - Classes: Use CapitalizedWords (CamelCase).
  - Constants: Use ALL_CAPS_WITH_UNDERSCORES.
- **Blank Lines:**
  - Use two blank lines to separate top-level function and class definitions, and one blank line to separate methods within a class.
- **Imports:**
  - Place all import statements at the top of the file.
  - Group imports in the following order: standard library, third-party libraries, local application-specific imports.
  - Use one import statement per module. Avoid `from module import *`.

- **Whitespace:**
    - Use spaces around operators (e.g., a = b + c).
    - Avoid excessive whitespace for alignment unless it significantly improves readability in specific contexts.
- **Comments:**
    - Use comments to explain why code is doing something, not what it is doing if it's already obvious.
    - Inline comments should be used sparingly, separated by at least two spaces from the code, and start with # followed by a single space.
- **Docstrings**:
    - Use PEP 257-style docstrings to document modules, classes, and functions, explaining their purpose, arguments, and return values.
- **Comparisons:**
    - Use is or is not for comparisons with None, True, and False (e.g., if x is None:).

Q-2) Indentation, comments, and naming conventions in Python.

➢ In Python, **indentation** defines code blocks, **comments** explain code for humans, and **naming conventions** promote readability and consistency according to the PEP 8 style guide.

- Indentation
    - Python uses indentation to group statements into blocks of code for control structures like `if` statements, `for` loops, and function definitions.

- All statements within the same block must have the same level of indentation. Inconsistent indentation will cause an IndentationError.

- The recommended style is to use **four spaces** per indentation level.

- **Comments**

  - Comments are notes for human readers and are ignored by the Python interpreter.

  - **Single-Line Comments**: Start with a hash symbol (#) followed by a single space. Everything after the # on that line is considered a comment.

  - # This is a single-line comment.

  - **Docstrings (Documentation Strings)**: Docstrings are enclosed in triple quotes (""" or ''') and are used to document modules, functions, classes, and methods.

  - “””

    Statements

    “””

- **Naming conventions**

  - Following naming conventions, primarily from PEP 8, makes your code consistent and easier for others to understand.

Type	**Convention**	**Example**
Variables	snake_case (lowercase with underscores)	user_name, total_count

Functions/Methods	snake_case (lowercase with underscores)	calculate_total(), get_data()
Classes	CapWords (CamelCase)	MyClass, HTTPRequest
Modules/Packages	Short, lowercase names (underscores optional in modules)	my_module.py, mypackage
Constants	UPPER_CASE_WITH_UNDERSCORES	MAX_OVERFLOW, PI

- **Start with letter or underscore**: Variable names cannot start with a number.
- **Case-sensitive**: name, Name, and NAME are three different variables.
- **Be descriptive**: Use meaningful, intention-revealing names. A name like days_since_last_login is better than dsl.

Q-3) Writing readable and maintainable code.

- Writing readable and maintainable Python code means structuring your programs so that they are easy to understand, update, and reuse by you and other developers. Readability and maintainability are crucial because code is read more often than it is written.
- Use meaningful and descriptive names for variables, functions, and classes to convey intent clearly. Avoid vague or single-letter names (except simple loop counters). Break complex logic into small, focused functions and organize related code into modules or packages, making it easier to test and maintain.
- Write clear docstrings and comments to explain why certain decisions are made, not just what the code does. Limit line length and use consistent indentation and whitespace to improve readability. Handle errors explicitly with proper exception handling and write unit tests to verify behavior and catch bugs early.

(3) Core Python Concepts

Q-1) Understanding data types: integers, floats, strings, lists, tuples, dictionaries, sets.

- Python has several built-in data types to store various kinds of data. Understanding these fundamental types is key to programming effectively in Python.

Data Type	Description	Example Literal
Integers (int)	Whole numbers, positive or negative, without decimals.	42
Floats (float)	Numbers with a decimal point or exponential notation.	3.14
Strings (str)	A sequence of Unicode characters used to record text data. Enclosed in single, double, or triple quotes.	"hello world"
Lists (list)	An ordered collection of items, which can be of different types. Enclosed in square brackets.	[1, "two", 3.0]
Tuples (tuple)	An ordered, immutable collection of items. Enclosed in parentheses.	(1, "two", 3.0)
Dictionaries (dict)	A collection of data stored in key-value pairs. Enclosed in curly braces.	{"a": 1, "b": 2}
Sets (set)	An unordered collection of unique items. Enclosed in curly braces.	{1, 2, 3}

Q-2) Python variables and memory allocation.

- **<u>Variables</u>**

    - A variable is a name which can store a value.
    - Unlike other programming languages, Python has no command for declaring a variable.
    - A variable is created at the moment you first assign a value to it.
    - E.g. number=20

      age = 21
    - A variable can have a short name (like a and b) or a more descriptive name (age,username,product_price).
    - A variable name must start with a letter or the underscore character.
    - A variable name cannot start with a number.
    - A variable name can only contain alphanumeric characters and underscores (A-z, 0-9, and _)
    - Variable names are case-sensitive (age, Age and AGE are three different variables)

- **<u>Memory allocation</u>**

- Python uses both a stack and a heap for memory management.

    - **Stack Memory**: Used for function calls and local variables (the references/pointers themselves). Memory on the stack is allocated and freed automatically when a function starts and finishes executing (LIFO - Last-In, First-Out).

    - **Heap Memory**: This is where all Python objects (e.g., integers, strings, lists, dictionaries, class instances) are stored. This memory is dynamically allocated at runtime and managed by the Python memory manager and garbage collector.

- Python's primary memory management mechanisms are

    - **Reference Counting**: Each object in memory has a counter that tracks how many variables (references) point to it. When the reference count drops to zero, the object is automatically deallocated (freed) from memory. This is a real-time process.

    - **Garbage Collection (Generational)**: Reference counting alone cannot detect circular references (where objects refer to each other, so their counts never reach zero). Python employs a separate, periodic generational garbage collector to identify and reclaim this "garbage" memory.

    - **Memory Pools (pymalloc)**: For efficiency with small objects (<= 512 bytes), CPython uses a specialized allocator called pymalloc. This system pre-allocates large chunks of memory (arenas) and divides them into pools and blocks to reduce the overhead of requesting memory from the operating system for every small object.

Q-3) Python operators: arithmetic, comparison, logical, bitwise.

- To perform specific operations we need to use some symbols ..that symbols are operator

- **Arithmetic Operators**

  - Arithmetic operators are used to perform common mathematical operations.

Operator	Name	Description	Example	Result
+	Addition	Adds two operands	5 + 2	7
-	Subtraction	Subtracts the right operand from the left	5 - 2	3
*	Multiplication	Multiplies two operands	5 * 2	10
/	Division	Divides the left operand by the right, returns a float	10 / 3	3.33...
%	Modulus	Returns the remainder of the division	10 % 3	1
**	Exponentiation	Raises the left operand to the power of the right	2 ** 3	8
//	Floor division	Divides and returns the integer part of the result (floors the result)	10 // 3	3

- **Comparison Operators**

    - Comparison operators (also known as relational operators) compare the values of two operands and return a boolean result (True or False).

Operator	Name	Description	Example	Result
==	Equal to	Returns True if both values are equal	5 == 5	True
!=	Not equal to	Returns True if the values are not equal	5 != 2	True
>	Greater than	Returns True if the left operand is greater than the right	5 > 2	True
<	Less than	Returns True if the left operand is less than the right	2 < 5	True
>=	Greater than or equal to	Returns True if the left operand is greater than or equal to the right	6 >= 6	True
<=	Less than or equal to	Returns True if the left operand is less than or equal to the right	2 <= 3	True

- **Logical Operators**

  - Logical operators are used to combine multiple conditional statements and determine the final truth value.

Operator	Description	Example	Result
and	Returns True if both statements are true	(5 > 2) and (4 < 5)	True
or	Returns True if at least one statement is true	(2 == 2) or (3 != 3)	True
not	Reverses the result, returns False if the result is true	not(2 < 1)	True

- **Bitwise Operators**

  - Bitwise operators act on operands as if they were strings of binary digits (bits), performing operations bit by bit.

Operator	Name	Description	Example (a=10, b=5)	Result
&	Bitwise AND	Sets each bit to 1 if both bits are 1	a & b	0 (binary 0000 0000)
|	Bitwise OR	Sets each bit to 1 if at least one of the two bits is 1	a | b	15 (binary 0000 1111)
^	Bitwise XOR	Sets each bit to 1 if only one of the two bits is 1	a ^ b	15 (binary 0000 1111)

~	Bitwise NOT	Inverts all the bits	~a	-11 (depends on number representation)
<<	Left shift	Shifts the bits to the left, filling empty spots with zero	a << 2	40 (binary 0010 1000)
>>	Right shift	Shifts the bits to the right	a >> 2	2 (binary 0000 0010)

- **Lab**

Q-4) How does the Python code structure work?

- Python code structure fundamentally relies on indentation to define code blocks and is organized using modules, functions, and classes to promote modularity and readability. The code is executed top-to-bottom when run as a script.

- **Core Structural Principles**

  - **Indentation:** Python uses whitespace (four spaces by convention) for indentation to define the scope of control flow statements (like if, for, while) and the body of functions and classes. This replaces the curly braces used in many other languages.

  - **Modularity (Modules and Packages):** Python programs are typically broken down into separate files called modules (.py files). These modules can contain functions, classes, and variables that can be reused in other parts of the program using import statements. A collection of modules in a directory with an __init__.py file is a package.

- **Statements:** The program is a sequence of individual statements, which can be simple (e.g., assignment) or complex (e.g., a function definition). Newlines mark the end of a statement.

- **Functions and Classes:**

    - **Functions** (def keyword) are named, reusable blocks of code that perform specific tasks. They help in abstracting logic and reducing code duplication.

    - **Classes** (class keyword) are blueprints for creating objects (instances) that bundle data (attributes) and behaviors (methods). This is central to Python's object-oriented nature.

- **if __name__ == "__main__": block**: This common structure defines the script's entry point. Code within this block runs only when the file is executed as a standalone script, not when it's imported as a module into another file. This separates reusable library code from script-specific execution logic.

Q-5) How to create variables in Python?

- In Python, you create a variable the moment you first assign a value to it using the equals sign (`=`). No special command or data type declaration is needed, as Python is dynamically typed.

- **Syntax**

  variable_name = value

```
# Assigning an integer (whole number)
user_age = 30
# Assigning a string (text) using double quotes
greeting = "Hello, World!"
# Assigning a string using single quotes (both are the same)
status = 'pending'
# Assigning a float (decimal number)
tax_rate = 0.05
# Assigning a boolean (True/False)
is_active = True
```

- **Naming Rules and Conventions**

  - Must start with a letter or an underscore (_).
  - Cannot start with a number.
  - Can only contain alpha-numeric characters and underscores (A-z, 0-9, and _).
  - Are case-sensitive (age, Age, and AGE are three different variables).
  - Cannot use any of Python's reserved keywords (e.g., if, while, class).

- **Best Practice (Convention)**

  - Python developers typically use `snake_case` (all lowercase with underscores separating words) for variable names (e.g., `first_name`, `total_cost`).

- **Assigning Multiple Variables:**
    - Python allows you to assign values to multiple variables in a single line
    - **Assign one value to multiple variables:**

      x = y = z = 0
    - **Assign different values to multiple variables:**

      item_name, item_price, tax_rate = "Laptop", 1000, 0.05

Q-6) How to take user input using the input() function.

- To take user input in Python, you use the built-in `input()` function. The function pauses program execution and waits for the user to type something and press the Enter key. The entered value is then returned as a string.

- **Basic Syntax**

➢ The basic syntax is `variable_name = input("prompt message: ")`

- **"Prompt message: "**: This string (the *prompt*) is displayed to the user to let them know what information is expected. It is good practice to include one.

- **input()**: This function captures whatever the user types.

  - **variable_name** =: The input captured is assigned to a variable for later use in the program.

- **Example: Getting a Name (String Input)**

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

  - **When you run this code:**

  1. The message "Enter your name: " is displayed.

  2. The program stops and waits for you to type (e.g., "Alice") and press Enter.

  3. The variable name stores the string "Alice".

  4. The final line prints the greeting "Hello, Alice!".

Q-7) How to take user input using the input() function.

- In Python you can check the type of a variable at runtime using the built-in type() function — it tells you what class/type the object currently has. Python is dynamically typed, so this information is determined while the program runs.

- **Example:**

```
x = 10
print(type(x))   # Output: <class 'int'>

y = "Hello"
print(type(y))   # Output: <class 'str'>
```

```
z = [1, 2, 3]
print(type(z))   # Output: <class 'list'>
```

- When you call type(variable), it returns the type object for that variable — e.g., an integer, string, list, etc.

(4) Conditional Statements

Q-1) Introduction to conditional statements: if, else, elif.

- Conditional statements, a fundamental concept in programming, allow a program to make decisions and control its flow of execution based on whether a condition is true or false.

- The primary keywords used for this purpose in many languages, including Python, are `if`, `else`, and `elif` (short for "else if").

- **The if Statement**

  - The if statement executes a block of code if a condition is True. For example, if age = 20, the condition age >= 18 is true, and a message is printed.

  - **Syntax :**

```
If condition:
        statements
```

- **The else Statement**

  - The else statement provides an alternative block of code to execute if the if condition is False. If age = 15, the if condition is false, so the else block runs, printing a different message.

  - **Syntax :**

```
if condition:
        statements
else:
        statement(s)
```

- **<u>The elif Statement</u>**

  - The elif statement allows checking multiple conditions in sequence. The program executes the block for the first true condition encountered. With score = 75, the program checks conditions until score >= 70 is true, printing "Grade: C".

  - **Syntax :**

```
If condition:
          statement
Elif condition:
          statement
```

Q-2) Nested if-else conditions.

- In Python, nested if-else means placing one if (or if–else) statement inside another. It's useful when you need to check multiple conditions in a hierarchy.

- **Syntax :**

```
if condition:
          statements
     if condition:
           statements
          else:
      statement(s)
```

(5) Looping (For, While)

Q-1) Introduction to for and while loops

- A loop statement allows us to execute a statement or group of statements multiple times.

- Python programming language provides following types of loops to handle looping requirements.

    - For Loop
    - While Loop

- **For Loop**

    - For loop has the ability to iterate over the items of any sequence, such as a list or a string.

    - **Syntax :**

```
for variable in sequence:
        statements
```

- Python programming language allows to use one loop inside another loop.

    - **Syntax :**

```
for iterating_var in sequence:
        for iterating_var insequence:
                statements
statements
```

- **While Loop**

    - A while loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

- **Syntax :**

```
while expression:
        statements
```

Q-2) How loops work in Python.

- Loops in Python are used to execute a block of code repeatedly, primarily using **for** loops (for iterating over sequences) and **while** loops (for repeating based on a condition).

- **For Loop**

  - The for loop is used to iterate over a sequence (such as a list, tuple, string, or range) and execute a block of code once for each item.

  - **How it Works:**

    - An iteration variable is assigned the value of the current item in the sequence during each pass.
    - The loop body executes for that item.
    - The process repeats until all items in the sequence have been processed, at which point the loop terminates.

- **While Loop**

  - The while loop is used to execute a block of statements repeatedly as long as a given condition remains True.

  - **How it Works:**

    - The loop starts by evaluating a condition.
    - If the condition is True, the code block inside the loop executes.

- After the block is run, the condition is re-evaluated.
- This continues until the condition becomes False, at which point the program exits the loop and continues with the next statement after the loop body.

Q-3) Using loops with collections (lists, tuples, etc.).

- In Python, you can iterate over collection types like lists and tuples primarily using **for loops**, which are designed to process each item in a sequence.

- <u>**Using For Loop**</u>

  - The most straightforward method for looping through collections is using a `for` loop with the `in` keyword.

  - **Syntax :**

```
for item in collection:
    # Code to execute for each item
```

  - <u>**Looping through a List**</u>

    - Lists are mutable collections defined by square brackets `[]`.

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

for fruit in fruits:
        print(f"I like {fruit}")
```

  - <u>**Looping through a Tuple**</u>

    - Tuples are immutable collections defined by round brackets `()`.

```
coordinates = (10, 20, 30)

for coord in coordinates:
    print(f"Coordinate: {coord}")
```

- **Using While Loop**
  - `while` loops can also be used by manually managing an index variable.

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

i = 0


while i < len(my_list):

    print(my_list[i])

    i += 1
```

(6) Generators and Iterators

Q-1) Understanding how generators work in Python.

- Python generators are functions that act as iterators, producing a sequence of values one at a time using the `yield` keyword. Unlike normal functions that return a single value and terminate, generators pause their execution, preserve their local state, and resume from where they left off when the next value is requested.

- **How They Work (Step-by-Step)**

    - You interact with a generator using a for loop (which is the most common way) or by explicitly calling the next() function on the generator object.

1. **Call the function**: When a generator function is called, the function body does not execute; a generator object is created instead.
2. **Request a value**: When you start iterating (e.g., with a for loop or next()), the code inside the function starts running.
3. **Hit yield**: Execution pauses at the first yield statement, and the yielded value is sent back to the caller. The function's state is "frozen".
4. **Next request**: When the next value is requested, the function resumes exactly where it stopped, with all its local variables intact.
5. **Stop**: This cycle continues until the function ends (runs out of code or hits a return statement without a value), at which point a StopIteration exception is raised, signaling the end of the sequence to the caller (a for loop catches this automatically).

Q-2) Difference between yield and return.

Feature	**yield**	**return**
Function Type	Used in a **generator function**.	Used in a **regular function**.
Execution	Pauses execution and saves its state (local variables, position).	Immediately terminates the function's execution.
Value Production	Can produce a sequence of multiple values over time, one at a time, on demand.	Returns a single value or a tuple of values at the end of the process.
Output Type	Returns a **generator object** (an iterator) to the caller.	Returns the actual result value(s) directly to the caller.
Memory Usage	Highly memory-efficient (lazy evaluation) as it only holds the current state and value in memory.	Less memory-efficient for large datasets as it computes and stores all values in memory at once.

Q-3) Understanding iterators and creating custom iterators.

In Python, an iterator is an object that represents a stream of data and produces items one at a time, keeping track of its position. Iterators follow the iterator protocol, which requires implementing the __iter__() and __next__() methods.

- **How They Work**

  - The built-in for loop in Python simplifies the iteration process. Under the hood, a for loop performs the following steps:

    1. Calls iter() on the iterable to get an iterator object.
    2. Repeatedly calls next() on the iterator to get the next item.
    3. Catches the StopIteration exception that the iterator raises when all items are exhausted, at which point the loop terminates.

- **Basic Usage Example**

```
my_list = [1, 2, 3]

# Get an iterator object from the iterable
my_iterator = iter(my_list)

# Manually access items using next()
print(next(my_iterator))  # Output: 1
print(next(my_iterator))  # Output: 2
print(next(my_iterator))  # Output: 3

# Trying to get another item raises StopIteration
# print(next(my_iterator))
```

(7) Functions and Methods

Q-1) Defining and calling functions in Python.

- A function is a block of organized, reusable code that is used to perform a single, related action.

- Functions provide better modularity for your application and a high degree of code reusing.

- **Defining a function**

  - Python gives us many built-in functions like print(), etc. But we can also create our own functions.

  - A function in Python is defined by a def statement. The general syntax looks like this:

```
def function-name(Parameter list):
    statements
return [expression]
```

- The keyword ”def” introduces a function definition.

- It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

- The ”return” statement returns with a value from a function.”return” without an expression argument returns None.

- Falling off the end of a function also returns None.

- <u>Calling a Functions</u>

  - Once function define, we can call it directly or in any other function also.

  - **Syntax :**

```
functionname()
or
functionname(argument)
```

Q-2) Function arguments (positional, keyword, default).

- It is possible to define functions with a variable number of arguments.

- The function arguments can be

  - Default arguments values
  - Keyword arguments

- **<u>Default arguments values</u>**

  - The most useful form is to specify a default value for one or more arguments.

  - This creates a function that can be called with fewer arguments than it is defined to allow.

  - Eg. def employeeDetails(name,gender='male',age=35)

  - This function can be called in several ways:giving only the mandatory argument :employeeDetails("Ramesh")

  - giving one of the optional arguments: employeeDetails("Ramesh",'Female')

- or even giving all arguments : employeeDetails(“Ramesh”,’Female’,31)

- **Keyword Arguments**
  - Functions can also be called using keyword arguments of the form kwarg=value.
  - For instance, the following function:
  - def parrot(voltage, state='a stiff',action='voom', type='Norwegian Blue'):
  - accepts one required argument (voltage) and three optional arguments (state, action, and type).

Q-3) Scope of variables in Python.

➢ The scope of a variable in Python refers to the region of the program where that variable is recognized and can be accessed.

- **Global & Local variables**
  - Defining a variable on the module level makes it a global variable, you don't need to global keyword.
  - The global keyword is needed only if you want to reassign the global variables in the function/method.
  - Defining a variable on the module level makes it a global variable, you don't need to global keyword.
  - The global keyword is needed only if you want to reassign the global variables in the function/method.
- **Local Variables**
  - If a variable is assigned a value anywhere within the function’s body, it’s assumed to be a local unless explicitly declared as global.
  - Local variables of functions can't be accessed from outside.

Q-4) Built-in methods for strings, lists, etc.

- Python provides a rich set of built-in methods for working with strings and lists, which are among its fundamental data types. These methods allow you to perform common operations efficiently without writing custom code.

- **String Methods**

  - String methods are used for manipulating text data. An important point is that strings in Python are **immutable**, meaning these methods return a new string with the modification, rather than changing the original string.

  - **Common string methods include:**

    - **upper():** Converts all characters in the string to uppercase.

    - **lower():** Converts all characters to lowercase.

    - **strip():** Removes leading and trailing whitespace.

    - **split(delimiter):** Splits the string into a list of substrings using a specified delimiter. If no delimiter is provided, it splits on all whitespace characters.

    - **join(list):** Concatenates elements of an iterable (like a list) into a single string using the string as the separator.

  - **find(substring):** Searches for the first occurrence of a substring and returns its index, or -1 if not found.

  - **replace(old, new):** Replaces all occurrences of an old substring with a new substring.

  - **startswith(prefix) / endswith(suffix):** Checks if the string starts or ends with the given prefix/suffix, returning True or False.

  - **isalpha() / isdigit() / isalnum():** Checks if all characters in the string belong to the specified character class (alphabetic, digit, or alphanumeric, respectively).

- **List Methods**

- List methods modify the list in place or provide information about its contents. Unlike strings, lists are **mutable**, meaning most methods change the list object directly.

- **Common list methods include:**

  - **append(item) (single item at the end):** adding elements

  - **extend(iterable) (all elements from an iterable):** adding elements

  - **insert(index, element):** inserting at a specific position

- **pop(index) (by index, or the last item if no index is given):** removing elements
- **remove(element) (first occurrence of a value):** removing elements
- **clear():** clear all elements
- **sort():** Lists can be sorted in ascending order
- **reverse():** the order of elements reversed
- **count(element):** the number of occurrences of an element
- **index(element):** the index of the first occurrence.

(8) Control Statements (Break, Continue, Pass)

Q-1) Understanding the role of break, continue, and pass in Python Loops.

- In Python, `break`, `continue`, and `pass` are control flow statements used to alter the normal execution of loops based on specific conditions.

- **Break**

  - The **break** statement terminates the loop entirely. It is used to exit a loop prematurely, even if the loop condition is still true or all elements in a sequence have not been iterated through. Control is transferred to the statement immediately following the loop.

  - **Use Case:** Searching for a specific item in a list and stopping once it is found to avoid unnecessary iterations.

  - **Example:**

```
fruits = ["apple", "banana", "cherry", "date"]
for fruit in fruits:
    if fruit == "cherry":
        print(f"Found {fruit}! Exiting the loop.")
        break
    print(f"Current fruit: {fruit}")
```

- **Output:**

```
Current fruit: apple
Current fruit: banana
Found cherry! Exiting the loop.
```

- **Continue**

  - The **continue** statement is used to skip the remaining code within the current iteration of the loop and jump to the next iteration.

  - **Use Case:** Skipping certain values or invalid inputs within a loop while continuing to process the rest of the items.

  - **Example:**

```python
for num in range(1, 6):
    if num == 3:
        print(f"Skipping {num}...")
        continue
    print(f"Processing number: {num}")
```

  - **Output:**

```
Processing number: 1
Processing number: 2
Skipping 3...
Processing number: 4
Processing number: 5
```

- **<u>Pass</u>**

  - The **pass** statement is a null operation; nothing happens when it executes. It is used as a placeholder where code is syntactically required, but you do not want any action to be performed yet.

  - **Use Case:** Used as a temporary placeholder when defining a function, class, or a loop body during development, preventing indentation errors until the actual code is written.

  - **<u>Example:</u>**

```
for i in range(3):
    if i == 1:
        pass # Placeholder for future logic
    else:
        print(f"Value of i: {i}")
```

  - **<u>Output:</u>**

```
Value of i: 0
Value of i: 2
```

(9) **String Manipulation**

Q-1) Understanding how to access and manipulate strings.

- In Python, strings are **immutable sequences** of Unicode characters that you can access using indexing and slicing, and manipulate using various built-in methods and operators.

- **Accessing Strings**

  - You can access individual characters or portions of a string using square brackets `[]`.

  - **Indexing**: Access individual characters using their zero-based position (index). Negative indices count from the end of the string.

```
s = "Hello World"
print(s[0])   # Output: H (first character)
print(s[6])   # Output: W
print(s[-1])  # Output: d (last character)
```

  - **Slicing**: Extract a substring using the syntax `[start:end:step]`. The character at the `end` index is **excluded** from the result.

```
s = "Hello World"
print(s[0:5])    # Output: Hello (characters from index 0 up to,
but not including, 5)
print(s[6:])     # Output: World (from index 6 to the end)
print(s[:5])     # Output: Hello (from the beginning to index 5)
print(s[::2])    # Output: HloWrd (every second character)
print(s[::-1])   # Output: dlroW olleH (reverses the string)
```

- **Manipulating Strings**

  - Since strings are immutable, any "manipulation" creates a **new** string.

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

```
greeting = "Hello"
name = "Alice"
message = greeting + ", " + name + "!"
print(message) # Output: Hello, Alice!
```

  - **Repetition**: Repeat a string multiple times using the `*` operator.

```
print("ha" * 3) # Output: hahaha
```

  - **Membership**: Check if a substring exists within a string using the `in` or `not in` keywords.

```
print("Py" in "Python")    # Output: True
print("Java" not in "Python") # Output: True
```

Q-2) Basic operations: concatenation, repetition, string methods (upper(), lower(), etc.).

- Python provides simple and intuitive ways to perform basic string operations like concatenation, repetition, and the use of various built-in methods such as upper() and lower().

- **Basic Operations**

Operation	Symbol	Description	Example	Result
Concatenation	+	Joins two or more strings together.	'Hello' + ' World'	'Hello World'
Repetition	*	Repeats a string a specified number of times.	'Hi' * 3	'HiHiHi'

- **Example**

```
# Concatenation example
first_name = "John"
last_name = "Doe"
full_name = first_name + " " + last_name
print(full_name)

# Repetition example
separator = "-"
print(separator * 10)
```

- **Common String Methods**

    - Python strings come with many built-in methods that allow you to modify or get information about the string. Methods are called using dot notation (.) after the string variable or literal.

Method	Description	Example	Result
upper()	Converts all characters in a string to uppercase.	'Hello'.upper()	'HELLO'
lower()	Converts all characters in a string to lowercase.	'WORLD'.lower()	'world'
capitalize()	Converts the first character to uppercase and the rest to lowercase.	'pYtHoN'.capitalize()	'Python'
title()	Converts the first character of each word to uppercase.	'learn python'.title()	'Learn Python'
strip()	Removes leading and trailing whitespace.	' whitespace '.strip()	'whitespace'
replace(old, new)	Replaces occurrences of a substring with another.	'I like cats'.replace('cats', 'dogs')	'I like dogs'
split(separator)	Splits a string into a list of substrings based on a delimiter.	'apple,banana,cherry'.split(',')	['apple', 'banana', 'cherry']

- **Example**

```
# Example using various string methods
text = "  Welcome to Python programming!  "

# Using strip() to remove whitespace
stripped_text = text.strip()
print(f"Stripped: '{stripped_text}'")

# Using upper() and lower()
print(f"Uppercase: {stripped_text.upper()}")
print(f"Lowercase: {stripped_text.lower()}")

# Using replace()
new_text = stripped_text.replace("Python", "Java")
print(f"Replaced: {new_text}")

# Using split()
words = stripped_text.split() # Splits by whitespace by default
print(f"Words list: {words}")
```

Q-3) String slicing.

- String slicing in Python is a method for extracting a substring from a sequence of characters by specifying a range of indices. It uses a concise syntax within square brackets `[]` that can include optional `start`, `stop`, and `step` parameters.

- **Syntax**

  - The general syntax for string slicing is string[start:stop:step].

Parameter	Description	Default Value (if omitted)
start	The beginning index (inclusive).	0 (start of the string).
stop	The ending index (exclusive); slicing stops *before* this index.	len(string) (end of the string).
step	The interval between characters (stride).	1 (every character).

- **Examples**

    - Let's use the string s = "Hello, World!" to demonstrate various slicing techniques:

Code	Explanation	Result
s[0:5]	Characters from index 0 up to (but not including) 5.	'Hello'
s[7:]	Characters from index 7 to the end of the string.	'World!'
s[:5]	Characters from the beginning up to (but not including) index 5.	'Hello'
s[:]	A copy of the entire string (uses all default values).	'Hello, World!'
s[::2]	Every second character in the string.	'Hlo ol!'
s[::-1]	Reverses the entire string using a negative step.	'!dlroW ,olleH'
s[-6:-1]	Characters from the 6th-to-last index up to the 2nd-to-last (exclusive).	'World'