**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 know for its simplicity and readability. Create 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 intelligence, scientific computing, automation and more.

**Key Features of Python**

1. **Easy to Learn:** Python is considered easy to learn and use.
2. **No Need to Compile:** Python code runs directly without needing to be compiled first, making development faster.
3. **Flexible data types:** You don’t need to declare the type of a variable beforehand, Python figures it out as you use it.
4. **Wide range of applications:** Python can be used for web development, data analysis, automation, machine learning, and more.
5. **Large community support:** Because it’s so popular, there are many online resources and forums to help you to learn and troubleshoot.

**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, was created by **Guido van Rossum** in the late 1980s as a hobby project, aiming to make coding more readable and accessible; the language was named after the British comedy group “Monty Python’s Flying Circus” and was first released in 1991, gaining popularity due to its simple syntax and wide range of applications like web development, data analysis, and scientific computing.

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

**Ans:** Python offers several advantages over other programming languages, including its simplicity readability, extensive libraries, versatility across various domains like data science and web development, a large supportive community, and the ability to rapidly develop application with less code, making it easier to learn and use for both beginners and experienced developers alike.

1. **Easy to Learn and Readable Syntax**
2. **Large Standard Library**
3. **Cross-Platform Compatibility**
4. **Rapid Development**
5. **Versatility**
6. **Extensive Data Science Ecosystem**
7. **Active Community and Support**
8. **Open-Source Nature**
9. **Object-Oriented Programming**

**(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 configured popular IDEs like **Anaconda, PyCharm, and VS Code.**

Step1: Install Python

Download and Install Python

1. Visit the official website – [Python.org](https://www.python.org/)
2. Download the latest version for windows, macOS, or Linux.
3. Run the installer and ensure your check **“Add Python to Path”** before installing.
4. Verify installation by opening a terminal (Command Prompt/ PowerShell on windows of Terminal on macOS/Linux) and typing:

Python –version

or

Python3 –version

Step2: 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](https://www.anaconda.com/)
2. Install Anaconda by following the setup instructions.
3. Open Anaconda Navigator and launch Jupyter Notebook, Spyder, of 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](https://www.jetbrains.com/pycharm/download/)
   1. Choose Community Edition (free) of 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](https://code.visualstudio.com/)
2. Install the Python Extension:
   1. Open VS Code
   2. Go to Extensions (Ctrl+Shift+X)
   3. Search for “Python” and install it.
3. Select Python Interpreter:
   1. 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 hellow.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 message.

print(“Hello, world!”)

This program tells Python to print “Hello, world!” to screen.

Step 2: Running Your First Python Program

Method 1: Using the command line (Recommended for Beginners)

1. Open Terminal/Command Prompt
   1. Windows: Press win + r, type cmd, and hit Enter.
   2. macOS/Linux: Open Terminal from Application or use ctlr + alt + t.
2. Navigate to the folder where your script is saved

If your script is in Documents, use:

1. 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, 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, 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 User)

1. Open Anaconda Navigator and launch Jupyter Notebook.
2. Click New -> Python3 Notebook.
3. In a new cell, type:

print(“Hello, world!”)

1. Click Run (shift + enter).

Step 3: Understanding the code

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

**Programming Style**

**(Q1) Write a Python program that demonstrates the correct use of indentation, comments, and variables following 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, world!”) # 4 space indentation

Bad Example ❌ (Incorrect Indentation)

def greet():

print(“Hello, world!) # Only 2 space (wrong)

1. 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, world!’.”””

Print(“Hello, world!”)

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

1. Naming Conventions in Python

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

✅ Recommended Naming Styles:

|  |  |  |
| --- | --- | --- |
| Type | Naming Style | Example |
| Variable | snake\_case | user\_name = “Alice” |
| Functions | snake\_case | def calculate\_sum(): |
| Classes | PascalCase | Class EmployeeDetails: |
| Constants | UPPER\_CASE | PI = 3.14 |
| Modules | Lowercase\_with\_underscores | Import my\_module |

Bad Examples ❌

def CalculateSum(): # ❌ Use lowercase with underscore 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 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):

“””Return 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)

**Core Python Concepts**

**(Q1) Understanding data types: Integers, floats, lists, tuples, dictionaries, sets.**

**Ans:** Python has many data types, some of which are built-in, such as numeric, classes, and exception, and others specialized, such as dates and times, double-ended queues, and fixed-type arrays. Data types distinguish data items according to their characteristics and structure. Classifying data this way can make it easier for computer systems to process information.

Python has three numeric data types: float, integer, and complex. Float stores decimal numbers, integers can store whole numbers, and complex is for storing complex numbers.

* **Integers (int)**

The integer class represents negative and positive whole numbers. You will not find fractions or decimals in an integer class. Also, the length of an integer value has no limits, and in Python, you can separate long numbers by using underscores, whereas commas might usually go in regular integers.

Ex: - x = 20

* **Float (float)**

The float class represents an actual number with a numerical value that can be expressed with a decimal point, commonly known as its floating-point representation. A decimal point specifies it and is accurate up to 15 decimal places. Float values are positive or negative.

Ex: - x = 20.5

* **Complex numbers (complex)**

The complex class represents a complex number data type. Complex numbers are expressed as (real part) + (imaginary part) j, where j denotes the imaginary unit. These numeric data sets are primarily used in computer graphics and scientific computing.

Ex: - 10 + 30j

* **String**

String in Python are sequence of characters and how Python handles textual data. Strings represent Unicode characters. Python does not have a character data type. A single character is a “string of length one.”

A collection of one or more characters is always put in a single quote, double quote, or triple quote.

Ex: - x = “Python”

* **List**

In Python, lists group together related data. List sequences are mutable, meaning you can change their value without changing their identity. You can construct lists in various ways, often involving square brackets and commas, creating lists that are either the same or in the same order as iterable items.

Ex: - x = [“apple”, “grapes”, “cherry”]

* **Tuple**

Tuples are ordered collections of objects similar to lists. One of the main distinctions between tuples and other data types is that they are immutable, meaning they cannot be modified after they have been created. They are also write-protected. Tuples can hold any number of elements.

Ex: - x = (“apple”, “grapes”, “cherry”)

* **Boolean**

Boolean is a built-in data type of Python that has two constant values: true or false. Boolean objects with a true value are considered truthful, while those that equal false are said to be false.

Ex: - x = True

* **Set**

In Python, a “set” is an unorganized compilation of data elements. Unlike a list or tuple, it doesn’t have any particular order. Although it comprises various elements, their order is undefined. The set is iterable and mutable, and it contains no duplicate elements, although frozen sets are immutable. Also, the type of elements in a set is sometimes different.

Sets can compute mathematical operations like union, difference, and intersection. Other common uses in Python include removing duplicates from a sequence and membership testing.

Ex: - {“apple”, “grapes”, “cherry”}

* **Dictionary**

A dictionary is an unordered set of key-value pairs that is a useful data structure for storing data. It can store multiple values under a single name, allowing easy access and retrieval of information. In its functioning, a dictionary is similar to a map.

Unlike other data types in Python that store only one value per element, a dictionary stores key-value pairs.

Ex: - {“name”: “Rose”, “age”: 16}

**(Q2) Python variable and memory allocation.**

**Ans:**

1. Variable in Python
   1. Variable store data in memory.
   2. No need to declare types explicitly (Python is dynamically typed).
2. Memory Allocation in Python
   1. Immutable types (int, float, str, tuple) share memory if they have the same value.
   2. Mutable types (list, dict, set) store references in memory.

a = 10

b = 10

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

list1 = [1, 2, 3]

list2 = [1, 2, 3]

print(id(list1), id(list2)) #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 – 2 = 2 |
| \* | Multiplication | 5 \* 2 = 10 |
| / | Division | 5 / 3 = 1.666 |
| // | Floor Division | 5 // 3 = 1 |
| % | Modulus | 5 % 3 = 2 |
| \*\* | Exponentiation | 2 \*\* 3 = 8 |

a = 10

b = 3

print(a // b, a % b, a \*\* b) # Output: 3 1 1000

1. Comparison (Relational) Operator

|  |  |  |
| --- | --- | --- |
| 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

1. 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 | (4 > 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

1. Bitwise Operator (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)

**Conditional Statement**

**(Q1) Introduction to conditional statements: if, else, elif.**

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

* If – Executes a block of code only if a condition is True.
* Elif (else if) – Checks multiple condition and executes which condition is true.
* Else – Executes when none of the conditions are True.
* The if statement

The if statement check 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:

Your are eligible to vote.

* 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.

* The elif (else if) Statement

Used when there are multiple conditions to check.

Exemple:

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

**Looping Statement (For, While)**

**(Q1) Introduction to for and while loops.**

**Ans:** In Python, “for” loops are used to iterate through each item in a sequence (like a list, string, or tuple), executing a block of code for every element, while “while” loops repeatedly execute a block of code as long as a specified condition remains true, allowing for more flexible iteration where the number of repetitions might not be known beforehand.

Key points about “for” loops:

Iterating through sequences:

The primary purpose of a “for” loop is to go through each item in a sequence, automatically assigning each element to a variable within the loop body.

Example:

fruits = [“apple”, “banana”, “orange”]

for fruit in fruits:

print(fruit)

Key point about “while” loops:

Conditional repetition:

A “while” loop continues to execute as long as a given condition evaluates to “True”.

Unknown number of iterations:

Unlike “for” loops, the number of times a “while” loop runs can vary depending on how the condition changes within the loop.

Example:

count = 1

while count <= 5:

print(count)

count += 1

**(Q2) How loops work in Python**

**Ans:** Python for loops are used for iterating over a sequence like lists, tuples strings, and ranges.

* For loop allows you to apply the same operation to every item within loop.
* Using loop can iterate over any iterable object, such as dictionary, list or any custom iterators.

Example:

s = [“apple”, “banana”, “orange”]

for i in s:

print(i)

Python for loop with string

This code uses a for loop to iterate over a **string**  and print each character on a new line. The loop assigns each character to the variable i and continues until all characters in the string have been processed.

Example:

s = “banana”

for i in s:

print(i)

Using range() with for loop

The range() function is commonly used with for loops to generate a sequence of numbers. It can take one, two, or three arguments:

* range(stop): Generates number from 0 to stop-1.
* Range(start, stop): Generates numbers from start to stop-1.
* Range(start,stop,step): Generates numbers from start to stop-1, incrementing by step.

Example:

for i in range(0, 10, 2):

print(i)

**(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

persons = {“name”: “Neel”, “age”: 24, “city”: “Anand”}

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)

**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 # Returnthe value and exits

def using\_yield():

yield 1 # Suspends execution, can continue later

print(using\_return()) # Output: 1

print(usint\_yield()) # Output: <generator object>

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

**(Q3) Understanding iterators and creatin 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

1. 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 = CountDown(5)

for num in counter:

print(num)

Output:

5

4

3

2

1

**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 support 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

1. 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!

1. 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.

1. Variable-Length Argument(\*args and \*\*kwargs)
   1. \*args allows multiple positional arguments as a tuple.
   2. \*\*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 variable 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

1. 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

1. 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

**(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() # Remove last element

nums.insert(1, 10) # Insert 10 at index 1

nums.sort() # Sort 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\_vaues([‘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}

**Control Statement (Break, Continue, Pass)**

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

**Break Statement**

The break statement in Python is used to exit or “break” out of a loop prematurely, before the loop has iterated through all its items of reached its condition. When the break statement is executed, the program immediately exits the loop, and the control moves to the next line of code after the loop.

Example: -

# Using for loop

for i in range(5):

if i == 3:

break

print(i) # Exit the loop when i is 3

# Using while loop

i = 0

while i < 5:

if i == 3:

break # Exit the loop when i is 3

print(i)

i += 1

**Continue Statement**

Python Continue statement is a loop control statement that forces to execute the next iteration of the loop while skipping the rest of the code inside the loop for the current iteration only, i.e. when the continue statement is executed in the loop, the code inside the loop following the continue statement will be skipped for the current iteration and the next iteration of the loop will being.

Example: -

for i in range(5):

if i == 3:

continue # Skip the rest of the code for i = 3

print(i)

**Pass Statement**

Pass statement in Python is null operation or a placeholder. It is used when a statement is syntactically required but we don’t want to execute any code. It does nothing but allows us to maintain the structure of our program.

Example: -

for i in range(5):

if i == 3:

pass # Placeholder for future code

print(i)

**String Manipulation**

**(Q1) Understanding how to access and manipulate strings.**

**Ans:** Strings in Python are sequence 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 (+)**

When you want to join two string then concatenation operator is used.

first = “Hello”

second = “World”

result = first + “ “ + second

print(result) # Output: Hello World

1. **Repetition (\*)**

Repeating a string multiple times.

Word = “Python ”

Print(word \* 3) # Output: Python Python Python

1. **String Length (len())**

Finding the length of a string.

text = “Python”

print(len(text)) # Output: 6

**String Method**

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

1. **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)

1. **Replacing & Splitting**

text = “I love coding”

print(text.replace(“love”, “like”)) # Output: I like coding

words = “apple,banana,orange”

print(words.split(“,”)) # Output: [‘apple’, ‘banana’, ‘orange’]

1. Checking Substrings

text = “Hello, Python”

print(“Python” in text) # True

print(“Java” not in text) # True

**(Q3) 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 (form index 2 to end)

1. Negative Indexing

text = “Python”

print(text[-3:]) # Output: hon

print(text[:-2]) # Output: Pyth

1. Step Slicing

text = “Python”

print(text[::2]) # Output: Pto (every second letter)

print(text[::-1]) # Output: nohtyP (reverses string)

**Advanced Python (map(), reduce(), filter(), Closures and Decorators)**

**(Q1) How functional programming works in Python.**

Functional programming is a programming paradigm in which code is structured primarily in the from of functions. The origins of this programming style arise from a branch of mathematics known as lambda calculus, which is the study of functions and their mathematical properties. In contrast to the popular object-oriented and procedural approaches, functional programming offers a different way of thinking when solving a problem.

**(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]

1. 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]

1. reduce() – Aggregate Data

reduce() reduces an iteragle 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):

der 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 maintain 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.