Python Assignment

Fundamentals Of Python Language

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

**Ans.** Python is one of the most popular and beginner-friendly programming

languages in the world. It was created by Guido van Rossum and first

released in 1991. Python is known for its clean and easy-to-read syntax,

which makes it a great choice for people who are just starting out with

programming.

**Key Features:**

1.Easy To Understand

2. Free And Open Source

3. High Level Language

4. Portable

5.Interpreted Language

**Q.2 History and evolution of Python.**

**Ans.**

**1.Beginning and Development (Late 1980s - 1991):**

Author: Dutch programmer Guido van Rossum.

Origin: Aimed to design a language fusing the best aspects of ABC (an educational language) with the capability of making system calls and utilizing C libraries.

Name: Inspired by the British comedy series “Monty Python’s Flying Circus” — not the snake.

First Released: February 1991 (Python 0.9.0)

Classes, functions, exceptions (or error types), and core data types (e.g., list, dict, str).

**2. Early Growth and Python 1. x (1991 - 2000)**

Python became popular among hobbyist and academics for its simplicity and readability.

Key features added:

Map/Filter/Reduce (Python 1.0), Lambda Functions Module system Improvements in exception handling

**3.Python 2. x Era (2000 - 2010s)**

Python 2.0 released in 2000:

List comprehensions, garbage collection through reference counting + cycle-detecting GC.

Unicode support (a major addition).

**Problem**: Backward compatibility with Python 1.x and even within 2.x made it messy over time.

**Community split**: Between those who wanted to preserve backward compatibility and those who wanted to clean up the language.

**4.Python 3. x Era (2008 - Present)**

python 3.0 (“Python 3000”) was released in December 2008:

Not backwards-compatible with Python 2 x.

Fixed a lot of warts in the language (i.e., print is a function now, support for Unicode)

Early adoption hampered by incompatible libraries.

As a reminder, Python 2 has been end-of-life since January 1, 2020.

**5.Python 3.0 (2020 – Present)**

In particular, Python 3 has dominated education, data science, web development, and automation.

Major modern features:

Type hints and gradual typing (typing module)

Some async programming with AsyncIO

Pattern matching (the match/case syntax, introduced with Python 3.10)

Performance advances (e.g., significant speed boosts in Python 3.11)

**6. Ecosystem and Popularity**

**Uses**: Web development (Django, Flask), Data Science (Pandas, NumPy, TensorFlow), Scripting, Automation, DevOps, and more.

**Community**: One of the most active and welcoming programming communities.

**Ranking**: Consistently among the top programming languages in the world (per TIOBE, Stack Overflow surveys, etc.).

**Q .3 Advantages of using Python over other programming languages.**

Ans. Python is one of the most popular programming languages today, and for good reason. Here are some of the main reasons why developers across the world love using it:

**1. Simple and Easy to Read**

One of Python’s biggest strengths is its clean, readable syntax. It feels almost like writing English, which makes it a great choice for beginners. You don’t need to worry about complex syntax rules like semicolons or curly braces.

**2. Comes with a Powerful Standard Library**

Python follows a "batteries included" philosophy, meaning it comes packed with a standard library full of useful modules. Whether you’re working with files, regular expressions, math operations, or internet protocols, there’s probably already a module for it.

**3. Massive Ecosystem and Community Support**

Python has one of the largest programming communities out there. Whatever you're building—whether it's a website, a machine learning model, or a simple automation script—chances are there's already a library or framework that can help. For example:

* **Web development**: Django, Flask, FastAPI
* **Data science & machine learning**: NumPy, Pandas, TensorFlow, Scikit-learn
* **Automation**: Selenium, PyAutoGUI

Plus, with so many users around the world, you’ll never be short on tutorials, support forums, and open-source tools.

**4. Works on All Major Platforms**

Python is cross-platform, meaning your code can run on Windows, macOS, or Linux without needing big changes. That makes it easier to develop and deploy applications across different environments.

**5. Fast Prototyping and Development**

Python's dynamic typing and flexible data types let you build things quickly. This makes it especially popular in startups, research environments, and anywhere else where speed and experimentation are key.

**Q 4. Installing Python and setting up the development environment (Anaconda, PyCharm, or VS Code).**

Ans. **Installing Python & Setting Up Your Development Environment**

Getting started with Python is straightforward, and you’ve got a few great tools to choose from depending on what you want to do. Here's how to get everything up and running.

**Step 1: Install Python (If you’re not using Anaconda)**

1. **Go to the official Python website**:  
   [https://www.python.org/downloads](https://www.python.org/downloads" \t "_new)
2. **Download the latest version** for your OS (Windows, macOS, Linux).
3. During installation on Windows:
   * Check **“Add Python to PATH”** before clicking “Install Now”.
   * It helps your system recognize Python commands from the terminal.
4. Verify installation:
   * Open a terminal (Command Prompt or Terminal) and run:

python --version

* + You should see the version number printed.

**Option 1: Use Anaconda (Recommended for Data Science / ML)**

Anaconda is a Python distribution that includes Python + hundreds of useful packages + a great environment manager.

**Steps:**

1. Download Anaconda from:  
   https://www.anaconda.com/products/distribution
2. Run the installer and follow the steps (default options are usually fine).
3. After installation, launch the **Anaconda Navigator** GUI or use the **Anaconda Prompt**.
4. To create a new environment:

conda create -n myenv python=3.11

conda activate myenv

1. You can install packages using:

conda install numpy pandas

**Option 2: Use PyCharm (Great for full applications and general Python dev)**

PyCharm is a full-featured Python IDE made by JetBrains.

**Steps:**

1. Download from:  
   [https://www.jetbrains.com/pycharm/](https://www.jetbrains.com/pycharm/" \t "_new)
   * Choose **Community Edition** (free) or **Professional** (paid, more features).
2. Install and launch PyCharm.
3. When creating a new project:
   * PyCharm will ask you to specify a Python interpreter.
   * It can detect installed Python versions or let you configure a new virtual environment.
4. Start coding with features like:
   * Auto-complete
   * Code navigation
   * Built-in terminal
   * Git integration

**Option 3: Use VS Code (Lightweight, fast, and very flexible)**

VS Code is a powerful code editor that works great with Python after adding a few extensions.

**Steps:**

1. Download from:  
   [https://code.visualstudio.com/](https://code.visualstudio.com/" \t "_new)
2. After installing, open VS Code and go to the **Extensions** tab.
3. Search for and install:
   * **Python** (by Microsoft)
   * Optionally: Jupyter, Pylance, and Code Runner
4. Open your Python file or project, and VS Code will prompt you to select a Python interpreter.
5. You can run Python code directly using the **Run** button or using the terminal.

Q 5. **Writing and executing your first Python program**.

**Ans.**

**Step 1**: Write Your First Python Program in Idle

print ("Hello, Python!")

**Step 2**: Save the File

Save the file with .py extension

**Step 3**: Run the Program

With use of f5 key you can run your program.

**Q 6. Understanding Python’s PEP 8 guidelines.**

Ans. **PEP 8** stands for **Python Enhancement Proposal 8** — it’s the **style guide** for writing Python code. Think of it as a rulebook that helps developers write code that looks clean and consistent.

PEP 8 doesn’t affect how the code runs — it’s all about **readability and maintainability**.

**Why Follow PEP 8?**

* Makes your code easier to read and understand.
* Keeps your project consistent (especially important in teams).
* Helps avoid bugs and confusion.
* Looks professional — most open-source Python projects follow PEP 8.

**Key PEP 8 Guidelines (Made Simple)**

**1. Use 4 Spaces per Indent**

No tabs — always use 4 spaces for indentation.

**2. Limit Lines to 79 Characters**

Break long lines using \ or parentheses.

**3. Blank Lines for Separation**

Use blank lines to separate functions, classes, or sections of logic.

**4. Use Spaces Wisely**

No space right inside parentheses, brackets, or before commas.

Q 7**. Indentation, comments, and naming conventions in Python.**

**Ans.**

**1. Indentation:**

Python uses **indentation** to define blocks of code. Unlike some other languages, indentation is **not optional** in Python.

**Standard practice:**

* Use **4 spaces** per indentation level.
* Avoid mixing tabs and spaces.

Code:

Name=input (“Enter Your Name:”)

if name:

print (f “hello, {name}”)

2.**Multi-line comments:**

Use multiple # lines or docstrings (for functions, classes, modules).

Code:

# This is a longer comment

# explaining the next block of code

def multiply(x, y):

return x \* y

3.**Docstrings:**

Use triple quotes (""") right after function/class definitions.

Code:

def divide(x, y):

"""

Returns the result of dividing x by y.

Raises ZeroDivisionError if y is 0.

"""

return x / y

4.**Naming Conventions**:

* Variables/functions: lower\_case\_with\_underscores
* Classes: CapitalizedWords
* Constants: ALL\_CAPS
* Private: \_prefix

Q 8. **Writing readable and maintainable code.**

**Ans:**

Use clear, descriptive names for variables, functions, and classes.

Write small functions that do one thing.

Add comments and docstrings to explain *why*, not just *what*.

Follow PEP 8 for formatting (indentation, spacing, naming).

Avoid magic numbers — use named constants.

Keep your code DRY (Don’t Repeat Yourself).

Group related code into modules or classes.

Use version control (like Git) to track changes.

Q 9**. Understanding data types: integers, floats, strings, lists, tuples, dictionaries, sets.**

**Ans .**

**int**: whole number

**float**: decimal

**str**: text

**list**: ordered, changeable

**tuple**: ordered, unchangeable

**dict**: key-value pairs

**set**: unordered, unique items

**Code:**

data = [42, 3.14, "hello", (1, 2), {"key": "value"}, {1, 2, 3}]

print(data)

Q 10. **Python variables and memory allocation.**

**Ans.**

In Python, a **variable** is a name that refers to a **value stored in memory**. When you assign a value, Python creates the object in memory and the variable points to it.

**Code:**

x = 10 # 'x' points to an integer object with value 10 in memory

Variables are **references**, not containers.

Memory is managed automatically using **reference counting** and **garbage collection**.

Q 11. **Python operators: arithmetic, comparison, logical, bitwise.**

Ans.

**Arithmetic**: +, -, \*, /, //, %, \*\*

**Comparison**: ==, !=, >, <, >=, <=

**Logical**: and, or, not

**Bitwise**: &, |, ^, ~, <<, >>

**Code:**

a, b = 5, 3

result = (a + b > 7) and (a & b == 1)

Q 12**. Introduction to conditional statements: if , else , elif .**

**Ans.**

if: runs if the condition is true

elif: checks another condition if previous is false

else: runs if all above conditions are false

**Code:**

x = 10

if x > 0:

print("Positive")

elif x == 0:

print("Zero")

else:

print("Negative")

**Q 13. Nested if- else conditions.**

**Ans.**

Nested if-else statements are used when you need to check multiple conditions inside other conditions.

**Code**:

x = 10

y = 5

if x > 0:

if y > 0:

print ("Both x and y are positive")

else:

print ("x is positive, but y is not")

else:

print ("x is not positive")

Q 14. **Introduction to for and while loops.**

**Ans.**

**for loop**: Iterates over a sequence (like a list, string, or range).

**Code**:

for i in range (3):

print(i) # Prints 0, 1, 2

**while loop**: Repeats as long as the condition is True.

**Code**:

count = 0

while count < 3:

print(count) # Prints 0, 1, 2

count += 1

Both loops allow repeating code but in different ways: for works with sequences, and while runs based on a condition.

**Q 15.** **How loops work in Python.**

**Ans.**

**1. for loop:**

It iterates over a sequence (list, tuple, string, etc.) or a range of numbers.  
The loop continues until it has iterated through the entire sequence.

**Code:**

for item in [1, 2, 3]:

print(item) # Prints 1, 2, 3

**2. while loop:**

It continues to run as long as a specified condition is True.  
It’s useful when you don’t know how many times you need to loop in advance.

**Code:**

count = 0

while count < 3:

print(count) # Prints 0, 1, 2

count += 1

**Loop flow:**

* **For loop:** iterates over items in the sequence.
* **While loop:** keeps checking the condition; if True, the loop runs.

Remember to update variables in while loops to avoid infinite loops!

**Q 16.** **Using loops with collections (lists, tuples, etc.).**

**Ans.**

1. **Using a for loop with a list:**

**Code:**

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

for fruit in fruits:

print(fruit) # Prints each fruit in the list

1. **Using a for loop with a tuple:**

**Code:**

coordinates = (10, 20, 30)

for coordinate in coordinates:

print(coordinate) # Prints each value in the tuple

**3. Using a for loop with a dictionary:**

You can loop over keys, values, or key-value pairs.

**Code:**

person = {"name": "Alice", "age": 30, "city": "New York"}

for key, value in person.items():

print(f"{key}: {value}") # Prints each key-value pair

1. **Using a for loop with a set:**

**Code:**

colors = {"red", "green", "blue"}

for color in colors:

print(color) # Prints each color in the set

Loops are a powerful tool for iterating over items in any collection type!

Q **17.** **Understanding how generators work in Python.**

**Ans.**

A generator in Python is a special type of iterator that yields values one at a time, rather than returning all values at once. This allows for lazy evaluation, meaning values are generated only when needed, which is memory efficient.

**How Generators Work:**

* They use the yield keyword instead of return.
* They maintain their state between calls, so you can iterate over them multiple times without recomputing all the values.

**Example of a Generator:**

def count\_up\_to(max):

count = 1

while count <= max:

yield count # Pauses here and returns the value

count += 1

gen = count\_up\_to(5)

for number in gen:

print(number) # Prints 1, 2, 3, 4, 5

**Key Points:**

* Memory efficiency: Generators do not store all values in memory.
* State persistence: The generator remembers where it left off after each yield.
* Lazy evaluation: Values are only generated when the loop requests them.

**Generators are especially useful for large datasets or infinite sequences!**

**Q 18.** **Difference between yield and return .**

**Ans.**

* **return:** Ends the function and sends a value back to the caller. The function is complete once it hits a return.
* **yield:** Pauses the function and returns a value to the caller. The function's state is saved, and it can resume from where it left off when the generator is called again.

**Key Differences:**

* **return:** Returns a value once and exits the function.
* **yield:** Returns a value each time it is called and allows the function to resume from the last yield state.

**Example :**

def get\_numbers():

return [1, 2, 3]

result = get\_numbers()

print(result) # Prints the entire list [1, 2, 3]

Example with yield:

def generate\_numbers():

yield 1

yield 2

yield 3

gen = generate\_numbers()

for num in gen:

print(num) # Prints 1, 2, 3 (each number on separate calls)

Q 19. **Understanding iterators and creating custom iterators.**

**Ans.**

An iterator in Python is an object that implements the \_\_iter\_\_() and \_\_next\_\_() methods. Iterators allow you to traverse through all the elements of a collection (like a list or tuple) one at a time.

**Key Concepts:**

* **\_\_iter\_\_()**: Returns the iterator object itself (used to initialize the iteration).
* **\_\_next\_\_()**: Returns the next item from the collection. When there are no more items, it raises StopIteration.

**Creating a Custom Iterator:**

You can create your own iterator by defining a class with these two methods.

**Example of a Custom Iterator:**

class Countdown:

def \_\_init\_\_(self, start):

self.start = start

def \_\_iter\_\_(self):

return self # Returns the iterator object itself

def \_\_next\_\_(self):

if self.start <= 0:

raise StopIteration # No more items to iterate

self.start -= 1

return self.start + 1

# Creating and using the custom iterator

countdown = Countdown(5)

for number in countdown:

print(number) # Prints 5, 4, 3, 2, 1

**Key Points:**

* **\_\_iter\_\_()**: Prepares the object for iteration.
* **\_\_next\_\_()**: Returns the next item in the sequence.
* When the iterator is exhausted (no more items), **StopIteration** is raised to stop the iteration.

Q 20. **Defining and calling functions in Python.**

**Ans.**

In Python, a function is a block of reusable code that performs a specific task. You define a function using the def keyword and call it by its name.

**Defining a Function:**

Use the def keyword, followed by the function name, parentheses, and a colon. Then, write the indented code block inside the function.

**Example:**

def greet(name):

print(f"Hello, {name}!")

greet("Alice") # Calling the function with argument "Alice"

**Key Points:**

* **Function definition**: def function\_name(parameters):
* **Function call**: function\_name(arguments)
* Functions can take parameters (input) and return values (output).

**Functions with Return Values:**

You can use the return statement to send a result back from the function.

def add(a, b):

return a + b

result = add(3, 4) # Calling the function and storing the result

print(result) # Prints 7

By defining functions, you can write more modular and reusable code!

Q 21. Function arguments (positional, keyword, default).

**Ans.**

In Python, function arguments can be passed in different ways: positional, keyword, and default.

**1. Positional Arguments:**

Arguments are assigned based on their position in the function call. The first argument corresponds to the first parameter, the second to the second, and so on.

Code:

def greet(name, age):

print(f"Name: {name}, Age: {age}")

greet("Alice", 25) # Positional arguments

**2. Keyword Arguments:**

Arguments are passed by explicitly naming the parameter. The order of arguments doesn't matter.

Code:

def greet(name, age):

print(f"Name: {name}, Age: {age}")

greet(age=25, name="Alice") # Keyword arguments

**3. Default Arguments:**

You can set default values for parameters. If no value is provided during the function call, the default value is used.

**Code:**

def greet(name, age=30): # Default value for age

print(f"Name: {name}, Age: {age}")

greet("Alice") # Uses the default value for age

greet("Bob", 40) # Overrides the default value

Q 22. **Scope of variables in Python.**

**Ans.**

The scope of a variable in Python refers to where that variable can be accessed or modified within the program. Python has different levels of variable scope:

**1. Local Scope:**

A variable defined inside a function is **local** to that function. It can only be accessed within the function.

Code:

def my\_function():

x = 10 # Local variable

print(x)

my\_function() # Prints 10

# print(x) # Error: x is not accessible outside the function

**2. Global Scope:**

A variable defined outside of all functions is **global**. It can be accessed and modified from anywhere in the code.

x = 20 # Global variable

def my\_function():

print(x) # Accessing global variable

my\_function() # Prints 20

**3. Enclosing Scope:**

This applies to variables in functions that are **nested** inside other functions. The inner function can access variables from its enclosing function but can't modify them.

Code:

def outer():

x = 30 # Enclosing variable

def inner():

print(x) # Accessing enclosing variable

inner()

outer() # Prints 30

**4. Built-in Scope:**

This refers to variables that are built into Python, like print(), len(), etc. These are always accessible.

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

**LEGB Rule:**

Python looks for variables in the following order of scope:

1. **Local (L)**: Inside the current function.
2. **Enclosing (E)**: In the nearest enclosing function.
3. **Global (G)**: In the top-level of the script.
4. **Built-in (B)**: In Python's built-in namespace.

**Modifying Global Variables:**

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

Code:

x = 50 # Global variable

def modify():

global x

x = 100 # Modifying the global variable

modify()

print(x) # Prints 100

Q 23**. Built-In methods for strings, lists, etc.**

**Ans.**

**1. String Methods:**

* **upper()**: Converts to uppercase.

text = "hello"

print(text.upper()) # "HELLO"

* **lower()**: Converts to lowercase.

text = "HELLO"

print(text.lower()) # "hello"

* **strip()**: Removes leading and trailing spaces.

text = " hello "

print(text.strip()) # "hello"

* **replace()**: Replaces a substring with another.

text = "hello world"

print(text.replace("world", "Python")) # "hello Python"

* **split()**: Splits a string into a list based on a delimiter.

text = "apple,banana,cherry"

print(text.split(",")) # ["apple", "banana", "cherry"]

* **find()**: Returns the index of the first occurrence of a substring.

text = "hello world"

print(text.find("world")) # 6

**2. List Methods:**

* **append()**: Adds an item to the end of the list.

numbers = [1, 2, 3]

numbers.append(4)

print(numbers) # [1, 2, 3, 4]

* **extend()**: Adds multiple items from another list.

numbers = [1, 2, 3]

numbers.extend([4, 5])

print(numbers) # [1, 2, 3, 4, 5]

* **pop()**: Removes and returns the last item (or specified index) from the list.

numbers = [1, 2, 3]

last\_item = numbers.pop()

print(last\_item) # 3

print(numbers) # [1, 2]

* **remove()**: Removes the first occurrence of a specified value.

numbers = [1, 2, 3, 2]

numbers.remove(2)

print(numbers) # [1, 3, 2]

* **sort()**: Sorts the list in ascending order.

numbers = [3, 1, 2]

numbers.sort()

print(numbers) # [1, 2, 3]

* **reverse()**: Reverses the list in place.

numbers = [1, 2, 3]

numbers.reverse()

print(numbers) # [3, 2, 1]

**3. Dictionary Methods (for reference):**

* **keys()**: Returns a view object of the dictionary's keys.

person = {"name": "Alice", "age": 30}

print(person.keys()) # dict\_keys(['name', 'age'])

* **values()**: Returns a view object of the dictionary's values.

print(person.values()) # dict\_values(['Alice', 30])

* **items()**: Returns a view object of the dictionary's key-value pairs.

print(person.items()) # dict\_items([('name', 'Alice'), ('age', 30)])

Q 24. **Understanding the role of break , continue , and pass in Python loops.**

**Ans.**

**1. break:**

The break statement is used to exit the loop prematurely, stopping further iterations. It’s commonly used when a condition is met and you want to stop the loop immediately.

Code:

for i in range(5):

if i == 3:

break # Exit the loop when i is 3

print(i)

**2. continue:**

The **continue** statement skips the current iteration and moves to the next iteration of the loop. It's useful when you want to skip certain conditions without stopping the loop.

Code:

for i in range(5):

if i == 3:

continue # Skip when i is 3

print(i)

**3. pass:**

The **pass** statement is a placeholder that does nothing. It’s often used as a temporary stub in loops or functions where code is expected but not yet written.

**Code:**

for i in range(5):

if i == 3:

pass # Does nothing for i == 3

print(i)

**Q 25.** **Practical Example: 1) Write a Python program to skip 'banana' in a list using the continue statement. List1 = ['apple', 'banana', 'mango']**

**Ans.**

List1 = ['apple', 'banana', 'mango']

for fruit in List1:

if fruit == 'banana':

continue # Skip 'banana'

print(fruit)

Q 26**. Practical Example: 2) Write a Python program to stop the loop once 'banana' is found using the break statement.**

**Ans:**

List1 = ['apple', 'banana', 'mango']

for fruit in List1:

if fruit == 'banana':

break # Stop the loop when 'banana' is found

print(fruit)

Q 27. **Understanding how to access and manipulate strings.**

**Ans.**

**1. Accessing Strings:**

You can access characters in a string using **indexing** and **slicing**.

* **Indexing**: Each character in a string has an index. Indexing starts from 0 for the first character.

**Code**:

text = "Python"

print(text[0]) # P

print(text[3]) # h

* **Negative Indexing**: You can also use negative indices to access characters from the end of the string.

**Code**:

print(text[-1]) # n (last character)

print(text[-2]) # o (second to last character)

* **Slicing**: You can extract a substring by specifying a range of indices.

Code:

print(text[0:3]) # "Pyt" (from index 0 to index 2)

print(text[3:]) # "hon" (from index 3 to the end)

print(text[:4]) # "Pyth" (from the start to index 3)

**2. Manipulating Strings:**

Python offers several string methods to manipulate strings.

* **Concatenation**: Combine strings using the + operator.

**Code**:

greeting = "Hello" + " " + "World!"

print(greeting) # "Hello World!"

* **Repetition**: Repeat a string using the \* operator.

**Code**:

text = "Hi! " \* 3

print(text) # "Hi! Hi! Hi! "

* **Changing Case**:
  + **upper()**: Converts to uppercase.
  + **lower()**: Converts to lowercase.
  + **title()**: Converts to title case (capitalizes the first letter of each word).

**Code**:

text = "hello"

print(text.upper()) # "HELLO"

print(text.lower()) # "hello"

print(text.title()) # "Hello"

* **Finding Substrings**:
  + **find()**: Returns the index of the first occurrence of a substring. Returns -1 if not found.

**Code**:

text = "I love Python"

print(text.find("love")) # 2

print(text.find("Java")) # -1

* **Replacing Substrings**:

**Code**:

text = "I like Python"

new\_text = text.replace("like", "love")

print(new\_text) # "I love Python"

* **Trimming Whitespace**:
  + **strip()**: Removes leading and trailing whitespaces.
  + **lstrip()**: Removes leading whitespaces.
  + **rstrip()**: Removes trailing whitespaces.

**Code**:

text = " hello "

print(text.strip()) # "hello"

* **Splitting a String**:  
  The **split()** method splits a string into a list based on a delimiter (space by default).

**Code**:

text = "apple orange banana"

words = text.split()

print(words) # ["apple", "orange", "banana"]

**3. String Immutability:**

Strings in Python are **immutable**, meaning they cannot be changed after creation. Any modification to a string creates a new string.

Code:

text = "hello"

# text[0] = "H" # This would result in an error because strings are immutable.

text = "H" + text[1:]

print(text) # "Hello"

**Q 28.** **Basic operations: concatenation, repetition, string methods ( upper() , lower() , etc.).**

**Ans.**

**1. Concatenation:**

You can concatenate (combine) two or more strings using the + operator.

**text1 = "Hello"**

**text2 = "World"**

**result = text1 + " " + text2 # Concatenate with a space in between**

**print(result) # Output: "Hello World"**

**2. Repetition:**

You can repeat a string using the \* operator.

Code:

**text = "Hi! "**

**repeated = text \* 3**

**print(repeated) # Output: "Hi! Hi! Hi! "**

**3. String Methods:**

* **upper():** Converts all characters to uppercase.

**Code:**

**text = "hello"**

**print(text.upper()) # Output: "HELLO"**

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

**Code:**

**text = "HELLO"**

**print(text.lower()) # Output: "hello"**

* **title():** Capitalizes the first letter of each word.

**Code:**

**text = "hello world"**

**print(text.title()) # Output: "Hello World"**

* **capitalize():** Capitalizes the first letter of the string.

**Code:**

**text = "hello"**

**print(text.capitalize()) # Output: "Hello"**

* **replace(): Replaces a specified substring with another substring.**

**Code:**

**text = "I like Python"**

**new\_text = text.replace("like", "love")**

**print(new\_text) # Output: "I love Python"**

* **find**(): Returns the index of the first occurrence of a substring, or -1 if not found.

**Code:**

**text = "I love Python"**

**print(text.find("love")) # Output: 2**

**print(text.find("Java")) # Output: -1**

* **split(): Splits a string into a list of substrings based on a delimiter (default is whitespace).**

**Code:**

**text = "apple orange banana"**

**words = text. Split()**

**print(words) # Output: ['apple', 'orange', 'banana']**

**Q 29.** **String slicing.**

**Ans.**

A String in python can be Sliced for Getting a part of The String is called String Slicing.

* **Slicing With Positive Index:**
  + The index in a String Starts From 0 to (length - 1) in Python. In order to Slice, we Use the Following Syntax:
  + Syntax:

SL=name [Ind\_Start: Ind\_End]

Ind\_Start= First Index of String

Ind\_End= Last Index of String

* **Slicing With Negative Index:**

Negative Index Starts From (-1) to (- Length). (-1) Refers to a Last Character of a String And (-Length) Refers To a First Character Of a String.

* + Syntax:

SL=name[Ind\_Start: Ind\_End]

Ind\_Start = Last Index of String

Ind\_End = First Index of String

* **Slicing With Skip Value:**

We can Provide a Skip Value as a Part of our Slice like This .

* + Syntax:

SL=name [Ind\_Start : Ind\_End : Skip\_Value]

Ind\_Start = First Index of String

Ind\_End = Last Index of String

Skip\_ Value = Value You Want to Skip

* Example:

s=Python Programming

//Positive Index

Print (s [3:13])

Print (s [ :14])

Print (s [2:])

Print (s [: :3])

Print (s [5::5])

//Negative Index

Print (s [-15: -3])

Print (s [: -2])

Print (s [-12:])

Print (s [-14: :2])

Print (s [: :-1])

Q 30**. How functional programming works in Python.**

**Ans.**

**Key Concepts in Functional Programming:**

1. **First-Class Functions**: Functions can be assigned to variables, passed as arguments to other functions, and returned from functions.

**Code**:

def greet(name):

return f"Hello, {name}"

func = greet # Assign function to a variable

print(func("Alice")) # Output: "Hello, Alice"

1. **Higher-Order Functions**: Functions that take other functions as arguments or return functions as results.

**Code**:

def apply\_function(func, value):

return func(value)

def double(x):

return x \* 2

result = apply\_function(double, 5) # Pass the function `double` as an argument

print(result) # Output: 10

1. **Lambda Functions**: Anonymous, one-line functions, often used with higher-order functions like map(), filter(), or reduce().

**Code**:

add = lambda x, y: x + y

print(add(2, 3)) # Output: 5

1. **Map, Filter, Reduce**: These built-in functions are functional programming tools that allow you to work with sequences in a functional way.
   * **map()**: Applies a function to each item in an iterable and returns a map object (which is an iterator).

**Code**:

numbers = [1, 2, 3, 4]

squared = map(lambda x: x\*\*2, numbers)

print(list(squared)) # Output: [1, 4, 9, 16]

* + **filter()**: Filters an iterable based on a condition, returning a filter object (which is also an iterator).

Code:

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

even\_numbers = filter(lambda x: x % 2 == 0, numbers)

print(list(even\_numbers)) # Output: [2, 4]

* + **reduce()** (from functools module): Accumulates a result by applying a function to the items of an iterable.

**Code:**

from functools import reduce

numbers = [1, 2, 3, 4]

product = reduce(lambda x, y: x \* y, numbers)

print(product) # Output: 24 (1\*2\*3\*4)

1. **Recursion**: In functional programming, recursion is often used instead of loops to repeat actions.

**Code**:

def factorial(n):

if n == 1:

return 1

else:

return n \* factorial(n-1)

print(factorial(5)) # Output: 120

Q 31. **Using map() , reduce() , and filter() functions for processing data.**

**Ans.**

**1. map():**

The **map()** function applies a given function to each item in an iterable (e.g., a list) and returns a **map object**, which can be converted to a list or another iterable type.

**Syntax:**

map(function, iterable)

**Example:**

Doubling each number in a list using **map()**:

numbers = [1, 2, 3, 4]

doubled = map(lambda x: x \* 2, numbers)

print(list(doubled)) # Output: [2, 4, 6, 8]

In this example, the lambda function doubles each element in the list.

**2. filter():**

The **filter()** function is used to **filter** elements of an iterable based on a given condition (i.e., a function that returns True or False). It returns a **filter object**, which can be converted to a list.

**Syntax:**

filter(function, iterable)

**Example:**

Filtering out even numbers from a list using **filter()**:

numbers = [1, 2, 3, 4, 5, 6]

even\_numbers = filter(lambda x: x % 2 == 0, numbers)

print(list(even\_numbers)) # Output: [2, 4, 6]

Here, the lambda function checks if a number is even (x % 2 == 0), and filter() returns a list of those numbers.

**3. reduce():**

The **reduce()** function (from the functools module) applies a rolling computation to sequential pairs of items in an iterable. It accumulates the result of applying a function to the items of the iterable.

**Syntax:**

from functools import reduce

reduce(function, iterable)

**Example:**

Calculating the product of all numbers in a list using **reduce()**:

from functools import reduce

numbers = [1, 2, 3, 4]

product = reduce(lambda x, y: x \* y, numbers)

print(product) # Output: 24 (1 \* 2 \* 3 \* 4)

In this example, reduce() applies the lambda function to the first two elements (1 \* 2), then to the result (2) and the next element (2 \* 3), and so on, ultimately computing the product.

Q 32. **Introduction to closures and decorators.**

**Ans.**

**1. Closures:**

A **closure** is a function that retains access to the variables of its enclosing scope, even after the outer function has finished executing. In other words, a closure "remembers" the environment in which it was created.

**How Closures Work:**

* A closure occurs when a nested function references variables from its enclosing function.
* The outer function must return the nested function, and the inner function has access to the outer function's variables, even after the outer function has finished executing.

**Example of Closure:**

def outer\_function(outer\_variable):

def inner\_function(inner\_variable):

return outer\_variable + inner\_variable # Access outer\_variable

return inner\_function

closure = outer\_function(10) # outer\_variable is set to 10

print(closure(5)) # Output: 15 (10 + 5)

In this example:

* inner\_function is a closure because it retains access to the outer\_variable even after outer\_function has finished executing.

**2. Decorators:**

A **decorator** is a function that takes another function as input and extends or modifies its behavior without permanently modifying the original function. Decorators are commonly used in Python to enhance the functionality of functions or methods, such as adding logging, authentication, or performance monitoring.

Decorators are implemented using **closures**. A decorator wraps another function and can modify its behavior.

**Syntax of a Decorator:**

def decorator\_function(original\_function):

def wrapper\_function():

print("Wrapper executed before {}".format(original\_function.\_\_name\_\_))

return original\_function()

return wrapper\_function

**Example of a Decorator:**

# A simple decorator

def my\_decorator(func):

def wrapper():

print("Before function call")

func()

print("After function call")

return wrapper

**Code:**

# Applying the decorator to a function

@my\_decorator

def say\_hello():

print("Hello!")

say\_hello()

**Output**:

Before function call

Hello!

After function call

In this example:

* The @my\_decorator syntax is used to **apply the decorator** to the say\_hello function.
* The wrapper function is executed before and after the original say\_hello function, allowing you to extend its behavior.