**Module 7: Python – Collections, functions and Modules**

**Accessing List**

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

**Ans:**

**Creating a list**

my\_list = ["apple", "banana", "cherry", "date", "elderberry", "fig", "grape"]

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

**Ans:**

**Accessing elements using positive indexing**

print("First element (index 0):", my\_list[0])

print("Third element (index 2):", my\_list[2])

**Accessing elements using negative indexing**

print("Last element (index -1):", my\_list[-1])

print("Second last element (index -2):", my\_list[-2])

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

**Ans:**

print("Elements from index 1 to 4:", my\_list[1:5]) # Index 5 is excluded

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

print("Elements from index 3 to end:", my\_list[3:])

print("Full list using slicing:", my\_list[:])

print("List in reverse order:", my\_list[::-1])

# Accessing every second element

print("Every second element:", my\_list[::2])

# Last 3 elements of the list

print("Last 3 elements:", my\_list[-3:])

### 2. List Operations

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

### Ans:

### Creating two lists

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

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

### 1. List Concatenation (+ operator)

### concatenated\_list = list1 + list2

### print("Concatenated List:", concatenated\_list)

### 2. List Repetition (\* operator)

### repeated\_list = list1 \* 2

### print("Repeated List:", repeated\_list)

### 3. Membership Test (in operator)

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

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

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

### Ans:

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

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

### list1.append("grape")

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

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

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

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

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

list1.remove("cherry")

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

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

popped\_element = list1.pop() # Removes last element

print("Popped element:", popped\_element)

print("List after pop:", list1)

**# Pop at a specific index**

popped\_index\_element = list1.pop(2) # Removes element at index 2

print("Element popped from index 2:", popped\_index\_element)

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

**3. Working with Lists**

**(Q1)Iterating over a list using loops.**

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

#### ****Example: Using a**** for ****Loop****

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

print("Iterating using a for loop:")

for fruit in fruits:

print(fruit)

#### ****Example: Using a**** while ****Loop****

print("Iterating using a while loop:")

i = 0

while i < len(fruits):

print(fruits[i])

i += 1

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

#### Ans:

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

numbers = [5, 2, 9, 1, 5, 6]

numbers.sort()

print("Sorted list (ascending):", numbers)

numbers.sort(reverse=True)

print("Sorted list (descending):", numbers)

#### ****Using**** sorted() ****(Returns a new sorted list)****

numbers = [3, 8, 1, 6, 0, 7]

sorted\_numbers = sorted(numbers)

print("Original list:", numbers)

print("Sorted list:", sorted\_numbers)

sorted\_descending = sorted(numbers, reverse=True)

print("Sorted in descending order:", sorted\_descending)

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

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

fruits.reverse()

print("Reversed list:", fruits)

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

**Ans:**

#### ****1 Adding Elements****

my\_list = [10, 20, 30]

# Using append()

my\_list.append(40)

print("After append:", my\_list)

# Using insert()

my\_list.insert(1, 15)

print("After insert at index 1:", my\_list)

# Using extend()

my\_list.extend([50, 60])

print("After extend:", my\_list)

#### ****2 Deleting Elements****

# Using remove()

my\_list.remove(30)

print("After removing 30:", my\_list)

# Using pop() - Removes last element

popped\_element = my\_list.pop()

print("Popped element:", popped\_element)

print("After pop:", my\_list)

# Using del

del my\_list[1] # Deletes the element at index 1

print("After deleting index 1:", my\_list)

# Using clear() - Empties the list

my\_list.clear()

print("After clear:", my\_list)

#### ****3 Updating Elements****

my\_list = [10, 20, 30, 40]

my\_list[2] = 35 # Updating index 2

print("Updated list:", my\_list)

#### ****4 Slicing a List****

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

print("Slice from index 2 to 5:", numbers[2:6]) # Excludes index 6

print("First 4 elements:", numbers[:4])

print("Elements from index 3 onwards:", numbers[3:])

print("Last 3 elements:", numbers[-3:])

print("Every second element:", numbers[::2])

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

**4. Tuple**

**(Q1)Introduction to tuples, immutability.**

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

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

#### ****Example: Creating a Tuple****

# Creating a tuple

my\_tuple = (1, 2, 3, "apple", "banana")

print("Tuple:", my\_tuple)

# Attempting to modify a tuple (this will raise an error)

# my\_tuple[1] = 100 # TypeError: 'tuple' object does not support item assignment

🔹 **Use Cases of Tuples:**

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

**(Q2)Creating and accessing elements in a tuple.**

**Ans:**

#### ****Creating a Tuple****

# Tuple with different data types

tuple1 = (10, 20, 30)

tuple2 = ("apple", "banana", "cherry")

tuple3 = (True, False, "Hello", 5.5)

# Single-element tuple (must include a comma!)

single\_element\_tuple = (5,) # Without the comma, it's just an integer!

print("Single-element tuple:", single\_element\_tuple)

#### ****Accessing Tuple Elements****

# Using positive indexing

print("First element:", tuple1[0])

print("Second element:", tuple2[1])

# Using negative indexing

print("Last element:", tuple3[-1])

print("Second last element:", tuple3[-2])

#### ****Tuple Slicing****

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

print("Slice from index 2 to 5:", numbers[2:6]) # Elements from index 2 to 5

print("Last three elements:", numbers[-3:])

print("Every second element:", numbers[::2])

print("Tuple in reverse:", numbers[::-1])

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

**Ans:**

#### ****1 Concatenation (****+****)****

tuple1 = (1, 2, 3)

tuple2 = (4, 5, 6)

# Joining two tuples

concatenated\_tuple = tuple1 + tuple2

print("Concatenated tuple:", concatenated\_tuple)

#### ****2 Repetition (****\*****)****

my\_tuple = ("Hello",) \* 3 # Repeat tuple 3 times

print("Repeated tuple:", my\_tuple)

#### ****3 Membership (****in****)****

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

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

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

5. Accessing Tuples

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

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

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

#### ****Example: Accessing Elements Using Indexing****

# Creating a tuple

my\_tuple = ("apple", "banana", "cherry", "date", "elderberry")

# Accessing using positive indexing

print("First element:", my\_tuple[0]) # apple

print("Third element:", my\_tuple[2]) # cherry

# Accessing using negative indexing

print("Last element:", my\_tuple[-1]) # elderberry

print("Second last element:", my\_tuple[-2]) # date

🔹 **Output:**

First element: apple

Third element: cherry

Last element: elderberry

Second last element: date

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

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

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

#### ****Example: Tuple Slicing****

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

print("Elements from index 2 to 5:", numbers[2:6]) # (2, 3, 4, 5)

print("First four elements:", numbers[:4]) # (0, 1, 2, 3)

print("Elements from index 3 to end:", numbers[3:]) # (3, 4, 5, 6, 7, 8, 9)

print("Last three elements:", numbers[-3:]) # (7, 8, 9)

print("Every second element:", numbers[::2]) # (0, 2, 4, 6, 8)

print("Tuple in reverse:", numbers[::-1]) # (9, 8, 7, 6, 5, 4, 3, 2, 1, 0)

🔹 **Output:**

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

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

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

Last three elements: (7, 8, 9)

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

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

**6. Dictionaries**

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

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

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

#### ****Example: Creating a Dictionary****

# Creating a dictionary

student = {

"name": "Alice",

"age": 20,

"grade": "A",

"subjects": ["Math", "Science"]

}

print("Student Dictionary:", student)

🔹 **Output:**

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

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

#### Ans: ****1 Accessing Elements****

# Using keys

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

# Using get() (avoids KeyError if key is missing)

print("Age:", student.get("age")) # 20

print("GPA (non-existent key):", student.get("GPA", "Not Available")) # Default value

#### ****2 Adding Elements****

# Adding a new key-value pair

student["GPA"] = 3.8

print("After Adding GPA:", student)

#### ****3 Updating Elements****

# Updating an existing key

student["grade"] = "A+"

print("After Updating Grade:", student)

#### ****4 Deleting Elements****

# Using del

del student["age"]

print("After Deleting Age:", student)

# Using pop() - Removes & returns the value

removed\_value = student.pop("GPA")

print("Removed GPA:", removed\_value)

print("After pop:", student)

# Using popitem() - Removes last inserted key-value pair (Python 3.7+)

student.popitem()

print("After popitem():", student)

# Using clear() - Removes all elements

student.clear()

print("After clear:", student)

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

**Ans:**

#### keys() ****- Get all keys****

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

print("Keys:", student.keys()) # dict\_keys(['name', 'age', 'grade'])

#### ****2**** values() ****- Get all values****

print("Values:", student.values()) # dict\_values(['Alice', 20, 'A'])

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

print("Items:", student.items())

# dict\_items([('name', 'Alice'), ('age', 20), ('grade', 'A')])

**7. Working with Dictionaries**

**(Q1) Iterating over a dictionary using loops.**

**Ans:**

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

#### ****1 Iterating Over Keys****

student = {

"name": "Alice",

"age": 20,

"grade": "A"

}

print("Iterating over keys:")

for key in student:

print(key)

#### ****2 Iterating Over Values****

print("\nIterating over values:")

for value in student.values():

print(value)

#### ****3 Iterating Over Key-Value Pairs****

print("\nIterating over key-value pairs:")

for key, value in student.items():

print(f"{key}: {value}")

🔹 **Output:**

Iterating over keys:

name

age

grade

Iterating over values:

Alice

20

A

Iterating over key-value pairs:

name: Alice

age: 20

grade: A

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

**Ans:**

#### ****1 Using a Loop****

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

keys = ["name", "age", "grade"]

values = ["Alice", 20, "A"]

# Using a loop to merge the lists into a dictionary

merged\_dict = {}

for i in range(len(keys)):

merged\_dict[keys[i]] = values[i]

print("Merged Dictionary using loop:", merged\_dict)

#### ****2 Using**** zip()

# Using zip() to combine lists into key-value pairs

merged\_dict\_zip = dict(zip(keys, values))

print("Merged Dictionary using zip():", merged\_dict\_zip)

🔹 **Output:**

Merged Dictionary using loop: {'name': 'Alice', 'age': 20, 'grade': 'A'}

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

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

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

# String to count characters

text = "hello world"

# Initialize an empty dictionary

char\_count = {}

# Iterating through the string

for char in text:

if char in char\_count:

char\_count[char] += 1

else:

char\_count[char] = 1

print("Character occurrences:", char\_count)

#### ****Output:****

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

**8. Functions**

**(Q1)Defining functions in Python.**

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

#### ****Syntax:****

def function\_name(parameters):

# Function body

# Code to execute

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

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

def greet():

print("Hello, World!")

# Calling the function

greet()

🔹 **Output:**

Hello, World!

**(Q3)Anonymous functions (lambda functions).**

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

def greet\_person(name):

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

# Calling the function with an argument

greet\_person("Alice")

greet\_person("Bob")

🔹 **Output:**

Hello, Alice!

Hello, Bob!

**9. Modules**

**(Q1) Introduction to Python modules and importing modules.**

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

#### ****What is a Module?****

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

#### ****Importing Modules****

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

import module\_name # Importing the whole module

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

from module\_name import function\_name # Import specific function

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

import module\_name as alias # Import with alias

**(Q2) Standard library modules: math, random.**

#### Ans:

#### ****1**** math ****Module****

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

import math

# Using some functions from the math module

print("Square root of 16:", math.sqrt(16)) # 4.0

print("Pi:", math.pi) # 3.141592653589793

print("Factorial of 5:", math.factorial(5)) # 120

#### ****2**** random ****Module****

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

import random

# Random number generation

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

# Random float between 0 and 1

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

# Randomly select an element from a list

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

print("Random fruit:", random.choice(fruits)) # e.g., 'banana'

# Shuffle the list randomly

random.shuffle(fruits)

print("Shuffled list:", fruits) # e.g., ['banana', 'cherry', 'apple', 'dat

**(Q3) Creating custom modules.**

#### Ans:

#### ****1 How to Create a Custom Module****

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

**Contents of my\_module.py:**

# Defining a function in the custom module

def greet(name):

return f"Hello, {name}!"

# Defining a variable

pi = 3.14159

#### ****2 Importing Your Custom Module****

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

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

import my\_module # Importing the custom module

# Using the function from the custom module

print(my\_module.greet("Alice"))

# Using the variable from the custom module

print("Value of Pi:", my\_module.pi)

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

from my\_module import greet, pi

print(greet("Bob"))

print("Value of Pi:", pi)