**Theory**

Q.1 Understanding how to create and access elements in a list.

A list in Python is a collection of ordered, mutable elements. You can create a list using square brackets [] and separate elements with commas.

Ex :

# Creating a list

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

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

mixed = ["hello", 42, 3.14, True] # A list with different data types

**Accessing Elements in a List**

access elements in a list using **indexing** and **slicing**.

**1. Indexing (Accessing Single Elements)**

Python uses **zero-based indexing**, meaning the first element is at index 0.

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

print(fruits[0]) # Output: apple

print(fruits[1]) # Output: banana

also use **negative indexing** to access elements from the end:

print(fruits[-1]) # Output: cherry

**2. Slicing (Accessing Multiple Elements)**

You can extract multiple elements from a list using slicing.

numbers = [10, 20, 30, 40, 50, 60, 70]

print(numbers[1:4]) # Output: [20, 30, 40] (indexes 1 to 3)

print(numbers[:3]) # Output: [10, 20, 30] (from start to index 2)

print(numbers[3:]) # Output: [40, 50, 60, 70] (from index 3 to end)

print(numbers[::2]) # Output: [10, 30, 50, 70] (every second element)

Q.2 Indexing in lists (positive and negative indexing).

Indexing allows you to access specific elements in a list. Python supports **both positive and negative indexing**.

**1. Positive Indexing**

* The first element is at **index 0**.
* The second element is at **index 1**, and so on.

Ex:-

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

print(fruits[0]) # Output: apple

print(fruits[2]) # Output: cherry

**2. Negative Indexing**

* The **last element** is at **index -1**.
* The **second last element** is at **index -2**, and so on.

Ex:-

print(fruits[-1]) # Output: elderberry

print(fruits[-3]) # Output: cherry

Q.3 Slicing a list: accessing a range of elements

list[start:stop:step]

 start: The index where the slice begins (inclusive).

 stop: The index where the slice ends (exclusive).

 step: The interval between elements (default is 1).

**1. Basic Slicing**

Extracts elements from start to stop - 1.

numbers = [10, 20, 30, 40, 50, 60, 70, 80]

print(numbers[1:5]) # Output: [20, 30, 40, 50]

Starts at index 1 (20).

Ends at index 5 - 1 = 4 (50).

print(numbers[:4]) # Output: [10, 20, 30, 40] (Start from index 0)

print(numbers[3:]) # Output: [40, 50, 60, 70, 80] (Up to the end)

3. Using a step

print(numbers[1:6:2]) # Output: [20, 40, 60]

Starts at index 1 (20).

Stops at index 6 - 1 = 5 (60).

Skips every 2 elements.

4. Negative Indexing in Slicing

print(numbers[-5:-2]) # Output: [40, 50, 60]

-5 refers to 40.

-2 refers to 70, but since stop is **exclusive**, it stops at 60.

**5. Reversing a List Using Slicing**

A step of -1 reverses a list.

print(numbers[::-1]) # Output: [80, 70, 60, 50, 40, 30, 20, 10]

Q.4 Common list operations: concatenation, repetition, membership.

**1. List Concatenation (+)**

You can join two or more lists using the + operator

list1 = [1, 2, 3]

list2 = [4, 5, 6]

result = list1 + list2

print(result) # Output: [1, 2, 3, 4, 5, 6]

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

You can repeat a list multiple times using the \* operator.

list1 = ["A", "B", "C"]

result = list1 \* 3

print(result) # Output: ['A', 'B', 'C', 'A', 'B', 'C', 'A', 'B', 'C']

**3. Membership Testing (in, not in)**

You can check if an element exists in a list using in or not in.

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

print("banana" in fruits) # Output: True

print("grape" in fruits) # Output: False

print("grape" not in fruits) # Output: True

Q.4 • Understanding list methods like append(), insert(), remove(), pop().

**1. append() – Add an element to the end**

The append() method adds a single element to the end of the list.

fruits = ["apple", "banana"]

fruits.append("cherry")

print(fruits)

# Output: ['apple', 'banana', 'cherry']

**2. insert() – Add an element at a specific position**

The insert(index, element) method inserts an element at a specific index

numbers = [10, 20, 30, 40]

numbers.insert(2, 25) # Insert 25 at index 2

print(numbers)

# Output: [10, 20, 25, 30, 40]

**3. remove() – Remove a specific element by value**

The remove(value) method removes the **first occurrence** of the specified element

colors = ["red", "blue", "green", "blue"]

colors.remove("blue")

print(colors)

# Output: ['red', 'green', 'blue']

**4. pop() – Remove and return an element by index**

The pop(index) method removes an element at a given index and **returns** it.  
If no index is specified, it removes and returns the **last** element.

animals = ["dog", "cat", "rabbit"]

removed\_item = animals.pop(1)

print(animals) # Output: ['dog', 'rabbit']

print(removed\_item) # Output: cat

Q.5 Iterating over a list using loops.

**1. Using a for Loop (Most Common)**

A for loop lets you iterate through each element in a list directly.

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

for fruit in fruits:

print(fruit)

**2. Using for Loop with range() (Index-Based Iteration)**

If you need both the **index** and the **element**, use range(len(list)).

numbers = [10, 20, 30, 40]

for i in range(len(numbers)):

print(f"Index {i}: {numbers[i]}")

Q.6 Sorting and reversing a list using sort(), sorted(), and reverse().

**1. Sorting a List**

**1.1 Using sort() (Modifies the List in Place)**

* The sort() method **sorts** the list in **ascending order** by default.
* It **modifies the original list** and does not return a new list.

numbers = [5, 2, 8, 1, 3]

numbers.sort()

print(numbers)

# Output: [1, 2, 3, 5, 8]

**1.2 Using sorted() (Returns a New Sorted List)**

* The sorted() function returns a **new sorted list**, leaving the original list **unchanged**.

numbers = [5, 2, 8, 1, 3]

sorted\_numbers = sorted(numbers)

print(sorted\_numbers) # Output: [1, 2, 3, 5, 8]

print(numbers) # Output: [5, 2, 8, 1, 3] (Original remains unchanged)

**2. Reversing a List**

**2.1 Using reverse() (Modifies the List in Place)**

* The reverse() method **reverses the list order** **without sorting**.
* It **modifies the original list**.

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

words.reverse()

print(words)

# Output: ['cherry', 'banana', 'apple']

Q.7 Basic list manipulations: addition, deletion, updating, and slicing

**1. Adding Elements to a List**

**A) Using append()**

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

**fruits.append("cherry")**

**print(fruits)**

**# Output: ['apple', 'banana', 'cherry']**

**2. Deleting Elements from a List**

**Using remove()**

**fruits.remove("banana")**

**print(fruits)**

**# Output: ['apple', 'orange', 'cherry', 'grape', 'mango']**

**3. Updating Elements in a List**

**You can update list elements by directly assigning new values using their index.**

**numbers = [10, 20, 30, 40]**

**numbers[2] = 99 # Update value at index 2**

**print(numbers)**

**# Output: [10, 20, 99, 40]**

**4. Slicing a List (Accessing a Range of Elements)**

**numbers = [10, 20, 30, 40, 50, 60, 70, 80]**

**print(numbers[1:5]) # Output: [20, 30, 40, 50]**

Q.8 Introduction to tuples, immutability.

A **tuple** is an **ordered, immutable** collection of elements in Python. It is similar to a list but **cannot be modified** after creation.

**Immutability in Tuples**

Unlike lists, **tuples cannot be changed** after creation. This means:

* You **cannot add** or **remove** elements.
* You **cannot modify** existing elements.

Q.9 Creating and accessing elements in a tuple

**1. Creating a Tuple**

A tuple is created using **parentheses ()** or simply separating values with commas

**2. Accessing Elements in a Tuple**

You can access elements using **indexing** and **slicing**, just like lists.

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

print(fruits) # Output: ('apple', 'banana', 'cherry')

**A) Indexing (Access by Position)**

Tuple indices start at 0 for the first element.

colors = ("red", "green", "blue")

print(colors[0]) # Output: red

print(colors[1]) # Output: green

**B) Slicing (Access a Range of Elements)**

You can use slicing to extract a **portion** of a tuple.

numbers = (10, 20, 30, 40, 50)

print(numbers[1:4]) # Output: (20, 30, 40) (Index 1 to 3)

print(numbers[:3]) # Output: (10, 20, 30) (From start to index 2)

Q.10 Basic operations with tuples: concatenation, repetition, membership

**1. Tuple Concatenation (+)**

You can concatenate two or more tuples using the + operator.

tuple1 = (1, 2, 3)

tuple2 = (4, 5, 6)

# Concatenate the tuples

result = tuple1 + tuple2

print(result)

# Output: (1, 2, 3, 4, 5, 6)

**2. Tuple Repetition (\*)**

You can repeat the elements of a tuple using the \* operator.

tuple1 = ("apple", "banana")

# Repeat the tuple 3 times

result = tuple1 \* 3

print(result)

# Output: ('apple', 'banana', 'apple', 'banana', 'apple', 'banana')

**3. Tuple Membership (in, not in)**

You can check if an element exists in a tuple using in or not in.

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

# Check if "banana" is in the tuple

print("banana" in fruits) # Output: True

# Check if "grape" is not in the tuple

print("grape" not in fruits) # Output: True

q.11 Accessing tuple elements using positive and negative indexing.

**1. Positive Indexing**

* The **first element** is at index 0.
* The **last element** is at index len(tuple) - 1.

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

print(fruits[0]) # Output: apple

print(fruits[2]) # Output: cherry

print(fruits[3]) # Output: date

**2. Negative Indexing**

* The **last element** is at index -1.
* The **second last element** is at index -2, and so on.
  + print(fruits[-1]) # Output: date
  + print(fruits[-2]) # Output: cherry
  + print(fruits[-4]) # Output: apple

Q.12 Slicing a tuple to access ranges of elements.

1. Basic Slicing

numbers = (10, 20, 30, 40, 50, 60)

print(numbers[1:4]) # Output: (20, 30, 40) (Includes index 1, excludes index 4)

print(numbers[:3]) # Output: (10, 20, 30) (From start to index 2)

print(numbers[2:]) # Output: (30, 40, 50, 60) (From index 2 to end)

1. Using Step in Slicing

numbers = (10, 20, 30, 40, 50, 60, 70, 80)

print(numbers[::2]) # Output: (10, 30, 50, 70) (Every 2nd element)

print(numbers[1::2]) # Output: (20, 40, 60, 80) (Every 2nd element starting from index 1)

c3. Negative Indexing in Slicing

numbers = (10, 20, 30, 40, 50, 60)

print(numbers[-4:-1]) # Output: (30, 40, 50) (From index -4 to -2)

print(numbers[-3:]) # Output: (40, 50, 60) (Last 3 elements)

1. Reversing a Tuple Using Slicing

numbers = (1, 2, 3, 4, 5)

print(numbers[::-1]) # Output: (5, 4, 3, 2, 1) (Reversed tuple)

Q.13 Introduction to dictionaries: key-value pairs.

A **dictionary** in Python is a **collection of key-value pairs**. It is:  
 **Unordered**   
 **Mutable** (can be changed)  
**Unique keys** (keys must be unique, values can be duplicated)  
 **Efficient** (fast lookups compared to lists)

**1. Creating a Dictionary**

Dictionaries are created using **curly braces {}**

# Creating a dictionary

student = {

"name": "Alice",

"age": 25,

"city": "New York"

}

print(student)

# Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}

**3. Dictionary Key-Value Pair Properties**

* **Keys must be unique**
* **Values can be duplicated**
* **Keys must be immutable (strings, numbers, tuples, etc.)**

bio = {

"name": "Bob",

"age": 30,

"gender": "male",

"age": 35 # Duplicate key; last value is kept

}

print(bio)

# Output: {'name': 'Bob', 'age': 35, 'gender': 'male'}

Q.14 Accessing, adding, updating, and deleting dictionary elements.

**1. Accessing Dictionary Elements**

You can access values using **keys** inside []

student = {"name": "Alice", "age": 25, "city": "New York"}

# Access using key

print(student["name"]) # Output: Alice

# Access using get() (avoids KeyError if key is missing)

print(student.get("age")) # Output: 25

print(student.get("gender", "Not specified")) # Output: Not specified

**2. Adding Elements to a Dictionary**

You can add a new key-value pair by assigning a value to a new key.

student["gender"] = "Female"

print(student)

# Output: {'name': 'Alice', 'age': 25, 'city': 'New York', 'gender': 'Female'}

**3. Updating Dictionary Elements**

To update a value, assign a new value to an existing key.

student["age"] = 26 # Updating age

print(student)

# Output: {'name': 'Alice', 'age': 26, 'city': 'New York', 'gender': 'Female'}

**4. Deleting Elements from a Dictionary**

**Using del to Remove a Key**

del student["city"]

print(student)

# Output: {'name': 'Alice', 'age': 27, 'gender': 'Female'}

Q. 15 Dictionary methods like keys(), values(), and items().

**1. keys() – Get All Keys**

The keys() method returns a **view object** containing all the keys in the dictionary.

student = {"name": "Alice", "age": 25, "city": "New York"}

keys = student.keys()

print(keys)

# Output: dict\_keys(['name', 'age', 'city'])

# Convert to a list

print(list(keys))

# Output: ['name', 'age', 'city']

**2. values() – Get All Values**

The values() method returns a **view object** containing all the values.

values = student.values()

print(values)

# Output: dict\_values(['Alice', 25, 'New York'])

# Convert to a list

print(list(values))

# Output: ['Alice', 25, 'New York']

**3. items() – Get All Key-Value Pairs**

The items() method returns a **view object** of **key-value pairs** as tuples.

items = student.items()

print(items)

# Output: dict\_items([('name', 'Alice'), ('age', 25), ('city', 'New York')])

# Convert to a list

print(list(items))

# Output: [('name', 'Alice'), ('age', 25), ('city', 'New York')]

Q.16 Iterating over a dictionary using loops

scores = {"Alice": 85, "Bob": 92, "Charlie": 78}

# Sort by key (alphabetically)

for key in sorted(scores):

print(f"{key}: {scores[key]}")

# Output:

# Alice: 85

# Bob: 92

# Charlie: 78

Q.17 Merging two lists into a dictionary using loops or zip().

**1. Using zip() (Best Method)**

The zip() function pairs elements from two lists and converts them into a dictionary using dict().

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

values = ["Alice", 25, "New York"]

# Using zip() to merge lists into a dictionary

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

print(merged\_dict)

# Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}

**2. Using a Loop (for)**

If zip() is not used, you can manually pair elements using a loop.

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

values = ["Alice", 25, "New York"]

merged\_dict = {}

for i in range(len(keys)):

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

print(merged\_dict)

# Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}

Q.18 Defining functions in Python

functions are defined using the def keyword. Functions allow code reuse, better organization, and improved readability. Here's a basic syntax for defining a function:

def hello():

print("Hello, World!")

hello() # Output: Hello, World!

Q.19 Different types of functions: with/without parameters, with/without return values.

**1. Function Without Parameters & Without Return Value**

* This function neither accepts arguments nor returns a value.

def hello():

print("Hello, World!")

hello() # Output: Hello, World!

Here, hello() does not take any input and does not return any output.

**2. Function Without Parameters & With Return Value**

* This function does not take any parameters but returns a value using the return statement.

def get\_message():

return "Hello from Python!"

msg = get\_message()

print(msg) # Output: Hello from Python!

**3. Function With Parameters & Without Return Value**

* This function takes input parameters but does not return a value.

def hello(name):

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

hello ("Alice") # Output: Hello, Alice!

The function prints the message directly but does not return anything.

**4. Function With Parameters & With Return Value**

* This function takes input parameters and returns a value.

def add(a, b):

return a + b

result = add(5, 3)

print(result) # Output: 8

Q.20 Anonymous functions (lambda functions).

A **lambda function** in Python is a small anonymous function that can have any number of arguments but only one expression. It is commonly used for short, simple operations where defining a full function with def would be unnecessary

Q.21 Introduction to Python modules and importing modules.

A **module** in Python is a file that contains Python code (functions, classes, or variables) that can be reused in other programs. It helps organize code into manageable and reusable components.

A module can be:

* A file containing Python code (.py file)
* A built-in module (pre-installed with Python)
* A third-party module (installed using pip)

**Why Use Modules?**

* **Code Reusability**: Write once, use multiple times.
* **Modularity**: Break large programs into smaller, manageable files.
* **Maintainability**: Easier debugging and updates.

**Importing Modules in Python**

* + **1. Importing a Built-in Module**
  + Python has many built-in modules such as math, random, and datetime.

import math

print(math.sqrt(25)) # Output: 5.0

Q.21 Standard library modules: math, random

Python's **standard library** includes many built-in modules to simplify coding. Two commonly used modules are:

1. **math** – Provides mathematical functions.
2. **random** – Generates random numbers and performs random operations.

**math Module**

The math module provides access to mathematical functions like square root, logarithms, trigonometry, and constants.

import math

print(math.sqrt(16)) # Output: 4.0

print(math.factorial(5)) # Output: 120

print(math.sin(math.pi / 2)) # Output: 1.0

print(math.log(100, 10)) # Output: 2.0

**2️ random Module**

The random module is used for generating random numbers and making random selections.

import random

print(random.random()) # Random float between 0 and 1

print(random.randint(1, 10)) # Random integer between 1 and 10

print(random.choice(["apple", "banana", "cherry"])) # Randomly pick an item

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

random.shuffle(nums)

print(nums) # Shuffled list

print(random.sample(range(1, 50), 5)) # Pick 5 random numbers from 1 to 49

Q.23 Creating custom modules.

A **custom module** in Python is a .py file that contains functions, variables, and classes that can be reused in other programs.

**1️ Creating a Custom Module**

**Step 1: Create a Python File (mymodule.py)**

Write your reusable code inside a new Python file.

# mymodule.py

def greet(name):

return f"Hello, {name}!"

def add(a, b):

return a + b

PI = 3.14159

2 **Importing and Using the Custom Module**

**Step 2: Import the Module in Another Script**

Create a separate Python file and import your custom module.

import mymodule

print(mymodule.greet("Alice")) # Output: Hello, Alice!

print(mymodule.add(5, 3)) # Output: 8

print(mymodule.PI) # Output: 3.14159