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

Creating a List:

- A list is a collection of items enclosed in square brackets [].
- It can store multiple values like numbers, strings, or mixed data types.

Example:
"A list is created by placing items inside square brackets, separated by commas."

Accessing Elements in a List:

- Each item in a list has a position called an index.
- Indexing starts from 0 for the first item.
- You can use positive indexes (from start) or negative indexes (from end) to access elements.

Q2 . Understanding how to create and access elements in a list.]

Positive Indexin

- The index starts at 0 for the first item and increases by 1 as you move through the list.
- Example: For the list ["apple", "banana", "cherry"], the indices are:
    - apple → index 0
    - banana → index 1
    - cherry → index 2

Negative Indexing:

- Negative indexing allows you to access elements from the end of the list.
- The last item has index -1, the second-to-last has index -2, and so on.
- Example: For the list ["apple", "banana", "cherry"], the negative indices are:
- cherry → index -1
- banana → index -2
- apple → index -3’

Q3. Slicing a list: accessing a range of elements.

Slicing in Lists

- Slicing means getting a part of the list.
- We use this format:
  list[start : end]
- It gives items from the start index to one before the end index.
- If you skip start or end, it takes from the beginning or to the end.

Example Idea (no code):

List: [A, B, C, D]
Slice [1:3] → gives [B, C]

Q4. Common list operations: concatenation, repetition, membership.

1. Concatenation

- Joining two lists using +.
- Creates a new list with all items combined.

---

2. Repetition

- Repeating a list using *.
- Useful to duplicate the contents.

---

3. Membership

- Using in or not in to check if an item exists in a list.

Q5. Understanding list methods like append(), insert(), remove(), pop().

1. append()

- Adds an item at the **end** of the list.

---

2. insert()

- Adds an item at a **specific index**.

---

3. remove()

- Removes the **first occurrence** of a value.

---

4. pop()

- Removes and **returns** the item at a given index.
- If no index is given, removes the **last item**.

Q6. Iterating over a list using loops.

Definition:

- Iterating means going through each item in a list one by one using a loop.

Commonly Used Loops:

1. **for loop** – Used to access each item directly.
2. **while loop** – Used with an index to go through the list.

Why Use It?

- To read, print, or perform actions on each item in a list.

Q7. Sorting and reversing a list using sort(), sorted(), and reverse().

1. sort()

- Sorts the list in **ascending order**.
- Changes the **original list**.

---

2. sorted()

- Returns a **new sorted list**.
- **Does not** change the original list.

---

3. reverse()

- Reverses the order of the list.
- Affects the **original list**.

Q8. Basic list manipulations: addition, deletion, updating, and slicing.

1. Addition

You can add elements using append() to add at the end of the list.
Use insert(index, value) to add at a specific position.

---

2. Deletion

Use remove(value) to delete by value, or pop(index) to remove by position.
del list[index] also deletes an item at a given index.

---

3. Updatin

Change an existing item by assigning a new value: list[index] = new_value.
This replaces the old value at that position.

---

4. Slicing

Slicing gives a sub-part of the list: list[start:end].
It includes the start index but excludes the end index.

Q9. Introduction to tuples, immutability.

Tuples:

- A **tuple** is an ordered collection of elements, written using **parentheses ()**.
- It can store multiple items of different data types, just like a list.
- Tuples are commonly used when the data should not be changed.
- They are faster than lists and use less memory.

Immutability:

- **Immutability** means once a tuple is created, its elements **cannot be modified**.
- You **cannot add, delete, or update** values in a tuple.
- This feature makes tuples more secure and reliable for fixed data.
- Immutability also helps improve performance in large programs.

Q10. Creating and accessing elements in a tuple.

Creating a Tuple:

- A tuple is created by placing items inside **parentheses ()**, separated by commas.
- Example: my_tuple = (10, 20, 30)
- You can also create a tuple without parentheses: my_tuple = 10, 20, 30
  For a single-item tuple, use a comma: single = (5,)

Accessing Elements:

- Tuple elements are accessed using **indexing**, just like lists.
  Index starts from **0** for the first item.
- You can also use **negative indexing** to access items from the end.

- Example: my_tuple[0] gives the first element, my_tuple[-1] gives the last.

Q11. Basic operations with tuples: concatenation, repetition, membership.

1. Concatenation:

- You can **join two tuples** using the + operator.
- It creates a new tuple with elements of both tuples.
- Example: (1, 2) + (3, 4) gives (1, 2, 3, 4).
- Tuples are immutable, so original tuples stay unchanged.

---

2. Repetition:

- Use the * operator to **repeat a tuple** multiple times.
- Example: (1, 2) * 3 gives (1, 2, 1, 2, 1, 2).
- It's useful for creating repeated patterns or placeholders.

---

3. Membership:

- The in and not in operators check if an item **exists** in a tuple.
- Example: 3 in (1, 2, 3) returns True.
- It helps in searching values in a tuple quickly.

Q12. Accessing tuple elements using positive and negative indexing.

Positive Indexing:

1. Indexing starts from **0** for the first element.

2. Access elements from **left to right**.
3. Example: my_tuple[1] accesses the **second** element.
4. Used when you know the **exact position** from the start.
5. Helps in **reading or using** specific elements in the tuple.

Negative Indexing:

1. Indexing starts from **-1** for the **last element**.
2. Access elements from **right to left**.
3. Example: my_tuple[-2] accesses the **second-last** element.
4. Useful for accessing the end of the tuple **without knowing its length**.
5. Works the same way as in lists and supports slicing too.

Q13. Slicing a tuple to access ranges of elements.

1. **Slicing** is used to access a **subset of elements** from a tuple.
2. The basic syntax is: tuple[start : end].
3. It includes the **start index** but **excludes the end index**.
4. **Negative indexes** can be used to slice from the end of the tuple.
5. Omitting start starts from the **beginning**, and omitting end goes to the **end**.
6. You can add a **step value**: tuple[start:end:step] to skip items.
7. Slicing **does not change** the original tuple — it returns a **new tuple**.

Q14. Introduction to dictionaries: key-value pair.

Dictionaries:

Key-Value Pairs

A dictionary is a type of data structure in Python used to store data in pairs.

- Each item has a key and a value.
- The key is like a label.
- The value is the data or information.

Features:

- Data is stored as key: value
- Keys are unique
- Values can be of any type
- Dictionary is changeable (mutable)

- Example :

Think of a dictionary like a real dictionary:

· Word (key): "Apple"

· Meaning (value): "A fruit"

Q15. Accessing, adding, updating, and deleting dictionary elements.

1. Accessing Elements

To use or read data from a dictionary, we use the key.

Each key helps us find its matching value.

If the key is not found, it may show an error.

---

2. Adding Elements

We can add new data to a dictionary by creating a new key and giving it a value.

This helps us store more information when needed.

---

3. Updating Elements

We can change or update the value of an existing key.

This is useful when the information has changed.

---

4. Deleting Elements

We can remove unwanted or incorrect data from the dictionary.

Deleting a key will also remove its value.

Q16. Dictionary methods like keys(), values(), and items().

1. keys() Method

This method returns a list of all the keys present in the dictionary.

It helps us know what data labels are used.

---

2. values() Method

This method returns a list of all the values stored in the dictionary.

It helps us see the actual data.

---

3. items() Method

This method returns a list of all key-value pairs in the dictionary.

Each pair is shown together, which helps in looping and displaying full data.

Q17. Iterating over a dictionary using loops.

In Python, we can use loops (mostly for loops) to go through a dictionary and work with its data.

This process is called iteration.

Why We Use Iteration in Dictionaries:

- To display all keys
- To display all values
- To display both keys and values together
- To perform operations on each item in the dictionary

Types of Iteration:

1. Iterating through Keys

We can loop through the dictionary to get all the keys one by one.

This helps us know what labels or categories are used in the dictionary.

---

2. Iterating through Values

We can also loop through the dictionary to get only the values.

Useful when we want to focus only on the data part.

---

3. Iterating through Key-Value Pairs

We can loop through both keys and their matching values together.

This is helpful when we need full information (label + data) for each item.

Q18. Merging two lists into a dictionary using loops or zip().

Merging Two Lists into a Dictionary Using Loops or zip()

We can create a dictionary by combining two separate lists — one for keys and one for values.

Each item in the first list becomes a key, and the item at the same position in the second list becomes its value.

1. Using Loops

- We use a loop to go through the index of both lists.
- At each step, we take one item from each list and make a key-value pair.
- This continues until all elements are added to the dictionary.

---

2. Using zip() Function

- zip() joins two lists together by pairing elements with the same index.
- It is a short and simple way to create key-value pairs directly.
- Then we convert the zipped result into a dictionary.

Benefits:

- Saves time when creating dictionaries.
- Useful when data is stored separately in two lists.

Q19. Counting occurrences of characters in a string using dictionaries.

Counting Occurrences of Characters in a String Using Dictionaries

Dictionaries can be used to count how many times each character appears in a string.

Each character becomes a key, and its count becomes the value.

How It Works:

1. Go through each character in the string one by one.
2. If the character is already in the dictionary, increase its count by 1.
3. If it is not in the dictionary, add it with a count of 1.
4. Repeat this process until the end of the string.

Why Use Dictionaries:

- It stores each character only once.
- It keeps track of how often each character appears.
- It gives a clear summary of the character frequency.

Q20. Defining functions in Python.

A function is a block of code that performs a specific task.

We can use a function whenever we want to repeat the same task multiple times.

Why Use Functions:

- Makes code clean and reusable
- Saves time by avoiding repetition
- Helps in organizing the program better

How to Define a Function:

1. Use the keyword def (which stands for define).
2. Write the function name.
3. Use parentheses ( ) — can have inputs called parameters.
4. Write a colon (:) at the end.
5. Add the indented block of code under it — this is the function's task.

Key Terms:

- Function Name: The name you give to the task.
- Parameters: Optional inputs the function can accept.
- Return: The result a function can give back to the program.

Q21. Different types of functions: with/without parameters, with/without return values.

Functions can be of four types based on input (parameters) and output (return value):

1. No Parameters, No Return Value

- No input, no output
- Just does a task (like printing a message)

2. With Parameters, No Return Value

- Takes input
- Does a task, but doesn't return anything

3. No Parameters, With Return Value

- No input
- Returns a result

4 . With Parameters, With Return Value

- Takes input
- Returns a result

Q22. Anonymous functions (lambda functions).

An anonymous function is a function without a name.

In Python, it is created using the lambda keyword, so it is also called a lambda function.

Features:

- Small and simple function
- Written in one line
- Can take any number of inputs but only one expression

Why Use Lambda Functions?

- Useful for short tasks
- Often used with functions like map(), filter(), and sorted().

Q23. Introduction to Python modules and importing modules.

What is a Module?

A module is a file that contains Python code (functions, variables, or classes).

It helps to organize code and reuse it in other programs.

Types of Modules:

1. Built-in modules – already available in Python (e.g., math, random)
2. User-defined modules – created by the user

Importing Modules:

To use a module in your program, you need to import it using the import keyword.

Q24. Standard library modules: math, random.

math Module

- Used for mathematical functions
- Includes functions like:

- sqrt() – square root
- pow() – power
- floor() – round down
- ceil() – round up
- pi – value of π

random Module

- Used to generate random numbers
- Common functions:
- random() – random float between 0 and 1
- randint(a, b) – random integer between a and b
- choice() – randomly picks an item from a list

Q25. Creating custom modules.

Creating Custom Modules in Python

A custom module is a Python file made by the user that contains functions, variables, or classes.

It allows us to reuse code in other programs.

How to Create a Custom Module:

1. Write your Python code (functions, variables, etc.) in a file.
2. Save the file with a .py extension.

   Example: my_module.py
3. In another Python file, import your module using the import keyword.

PYTHON MODULE - 2