Question 1 : Discuss string slicing and provide examples.

Answer : String slicing in Python allows you to extract a specific portion of a string by specifying a range of indices. Slicing is useful when you want to retrieve substrings or reverse strings.

1. start: The index where the slice begins (inclusive). If not specified, the default is the beginning (0).

2. end: The index where the slice ends (exclusive). If not specified, it defaults to the length of the string.

3. step: The interval between characters to include. The default is 1.

Example text = " Amritanshu Mishra" print(text[0:5]) # output : Amrit

Question 2 : Explain the key features of lists in Python.

Answer : In Python, lists are one of the most versatile and commonly used data structures. They allow you to store multiple items in a single variable. Lists are ordered, mutable (modifiable), and can contain elements of different data types.

Key Features of list:

Ordered:

Lists maintain the order of elements as they are inserted. Each element in the list has a specific index starting from 0. Example : my_list = [10, 20, 30] print(my_list[1]) # Output: 20

Mutable:

Lists are mutable, meaning you can change, update, or remove elements after the list is created. Example (modifying a list): my_list = [1, 2, 3] my_list[1] = 20 # Change the second element print(my_list) # Output: [1, 20, 3]

Example of a list in Python.

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

Append an element

my_list.append(6) print(my_list) # Output: [1, 2, 3, 4, 5, 6]

Insert at a specific position

my_list.insert(2, 99) print(my_list) # Output: [1, 2, 99, 3, 4, 5, 6]

Remove an element

my_list.remove(99) print(my_list) # Output: [1, 2, 3, 4, 5, 6]

Accessing elements

print(my_list[0]) # Output: 1

Question 3 : Describe how to access, modify, and delete elements in a list with examples.

Answer : In Python, lists are mutable, which means you can easily access, modify, and delete elements. Let's explore how to do these operations with examples.

1. Accessing Elements in a List You can access individual elements in a list using indexing. Python uses zero-based indexing, meaning the first element is at index 0.

Example: my_list = [10, 20, 30, 40, 50] print(my_list[0]) # Access the first element, Output: 10 print(my_list[3]) # Access the fourth element, Output: 40

2. Modifying Elements in a List You can modify elements in a list by assigning a new value to a specific index.

Example: my_list = [10, 20, 30, 40, 50]

Modify the second element

my_list[1] = 200 print(my_list) # Output: [10, 200, 30, 40, 50]

Modify the last element using negative indexing

my_list[-1] = 500 print(my_list) # Output: [10, 200, 30, 40, 500]

3. Deleting Elements from a List You can remove elements from a list using several methods:

a. del Statement: Removes an element by index. my_list = [10, 20, 30, 40, 50] del my_list[2] # Remove the element at index 2 (30) print(my_list) # Output: [10, 20, 40, 50]

b. remove() Method: Removes the first occurrence of a specified value. my_list = [10, 20, 30, 40, 50] my_list.remove(40) # Remove the element with value 40 print(my_list) # Output: [10, 20, 30, 50]

c. pop() Method: Removes and returns the element at a specific index. If no index is specified, it removes the last item by default. my_list = [10, 20, 30, 40, 50]

Remove and return the element at index 1

popped_element = my_list.pop(1) print(popped_element) # Output: 20 print(my_list) # Output: [10, 30, 40, 50]

Remove and return the last element

popped_element = my_list.pop() print(popped_element) # Output: 50 print(my_list) # Output: [10, 30, 40]

Question 4 : Compare and contrast tuples and lists with examples.

Answer : Tuples and lists are both data structures in Python used to store collections of items, but they have some key differences in terms of mutability, syntax, and usage. Let's compare and contrast them:

1. Mutability List: Mutable (can be changed after creation). Tuple: Immutable (cannot be changed after creation). Example:

   ## List (Mutable)

   my_list = [1, 2, 3] my_list[1] = 20 # Changing the second element print(my_list) # Output: [1, 20, 3]

Tuple (Immutable)

my_tuple = (1, 2, 3)

my_tuple[1] = 20 # This will raise an error, as tuples cannot be changed

2. Syntax List: Defined using square brackets []. Tuple: Defined using parentheses (). Example: my_list = [1, 2, 3] # List my_tuple = (1, 2, 3) # Tuple

3. Use Cases List: Used when the data is expected to change or when items need to be added/removed frequently. Tuple: Used when the data should not be changed (e.g., for fixed collections of items). Example: List: Storing a list of tasks that might change. Tuple: Storing coordinates or fixed configuration settings.

   ## List example: A mutable list of tasks

   tasks = ["study", "exercise"] tasks.append("sleep") print(tasks) # Output: ['study', 'exercise', 'sleep']

Tuple example: A fixed set of coordinates

coordinates = (10.0, 20.0)

5. Functions and Methods List: Lists have more built-in methods like append(), remove(), pop(), etc., because they are mutable. Tuple: Tuples have fewer methods since they are immutable. Example:

List methods

my_list = [1, 2, 3] my_list.append(4) # Adds 4 to the list print(my_list) # Output: [1, 2, 3, 4]

Tuple methods

my_tuple = (1, 2, 3) print(my_tuple.count(2)) # Count occurrences of an element, Output: 1

Question 5 : Describe the key features of sets and provide examples of their use.

Answer : In Python, sets are a collection data type that is unordered, unindexed, and contains unique elements. Sets are useful for storing non-duplicate elements and for performing common mathematical operations such as unions, intersections, and differences.

Key Features of Sets:

1. Unordered:

The elements in a set do not have a specific order. You cannot access elements by index like in lists or tuples. Example: my_set = {10, 20, 30} print(my_set) # Output: {20, 10, 30} (order may vary)

2. Unique Elements:

A set automatically removes duplicate elements, meaning each element is unique. Example: my_set = {1, 2, 2, 3, 4} print(my_set) # Output: {1, 2, 3, 4} (duplicate 2 is removed)

3. Mutable:

Sets are mutable, meaning you can add and remove elements after the set has been created. Example (adding and removing elements): my_set = {1, 2, 3} my_set.add(4) # Adds element 4 print(my_set) # Output: {1, 2, 3, 4}

my_set.remove(2) # Removes element 2 print(my_set) # Output: {1, 3, 4}

4. Unindexed:

Unlike lists or tuples, sets do not support indexing. You cannot retrieve an element by its position. Example: my_set = {10, 20, 30}

my_set[1] # This would raise an error because sets don't support indexing

5. No Duplicate Elements:

Since sets automatically discard duplicates, they are useful when you need a collection of distinct items. Example: my_list = [1, 2, 3, 1, 2, 4] unique_set = set(my_list) print(unique_set) # Output: {1, 2, 3, 4}

Summary of Key Features: Unordered: The elements have no specific order. Unique: No duplicates are allowed. Mutable: You can add or remove elements. Set Operations: Supports union, intersection, difference, and symmetric difference. Fast Membership Testing: Efficiently checks if an element exists in the set. Sets are ideal when you need to store distinct items, perform fast membership tests, or use mathematical set operations like union or intersection.

Question 6 : Discuss the use cases of tuples and sets in Python programming.

Answer : Tuples are immutable and ordered collections, making them ideal for scenarios where data should remain unchanged and its order matters.

Use cases of tuple:

1. Fixed data: Tuples are useful for storing constant data that should not be modified. For instance, configuration settings, fixed options, or constant values.
2. Multiple return values from functions: Functions often return tuples when you want to return multiple values at once.
3. Storing heterogeneous data: Tuples can hold different data types, which is useful for grouping related but diverse data.
4. Data integrity: If you need to ensure that data remains unchanged throughout the program, tuples provide a safe way to guarantee this immutability.
5. Efficient memory uses: Tuples use less memory than lists, making them a more memory-efficient choice for large, constant data sets that don’t need modification.

Use cases of Sets:

1. Removing Duplicates from a Collection: When you need to remove duplicates from a list or any collection, converting it to a set automatically eliminates the duplicates.
2. Membership Testing: Sets are optimized for checking whether an element exists in the collection, making them ideal for fast lookups.
3. Mathematical Set Operations: Sets support operations such as union, intersection, difference, and symmetric difference. These are useful when comparing datasets or filtering based on multiple criteria.
4. Tracking Unique Items: Sets are great for keeping track of unique items. For example, when counting unique visitors to a website or tracking items sold in a store.
5. Filtering Data: When you need to filter data based on the presence or absence of certain elements, sets make it easy to find unique elements or eliminate specific values.
6. Set Comparisons: Sets can be used to compare datasets in tasks such as determining which items are common between two datasets, or which are exclusive to one set.

Question 7 : Describe how to add, modify, and delete items in a dictionary with examples.

Answer: In Python, dictionaries are collections of key-value pairs that are mutable, meaning you can add, modify, and delete items. Here’s how you can perform these operations with examples.

1. Adding Items to a Dictionary You can add new key-value pairs to a dictionary by simply assigning a value to a new key.

Example:

Create an empty dictionary

my_dict = {}

Add key-value pairs

my_dict['name'] = 'Alice' my_dict['age'] = 25

print(my_dict) # Output: {'name': 'Alice', 'age': 25}

2. Modifying Items in a Dictionary To modify an item in a dictionary, you access the key and assign it a new value.

Example:

Modify the value associated with the 'age' key

my_dict['age'] = 26

print(my_dict) # Output: {'name': 'Alice', 'age': 26, 'city': 'New York', 'email': 'alice@example.com'}

3. Deleting Items from a Dictionary You can remove items from a dictionary in several ways:

Using del Statement: This removes the key and its corresponding value from the dictionary.

Remove the 'city' key

del my_dict['city']

print(my_dict) # Output: {'name': 'Alice', 'age': 27, 'email': 'alice@example.com'}

Question 8 : Discuss the importance of dictionary keys being immutable and provide examples.

Answer : In Python, dictionary keys must be immutable. This is crucial because dictionaries use keys to look up values quickly using a technique called hashing. Immutable objects like strings, numbers, and tuples (when containing only immutable elements) can be hashed, meaning they can serve as dictionary keys. Mutable objects like lists and sets, however, are not hashable and cannot be used as keys.

Dictionary Keys must Be Immutable becouse: Hashing Requirement:

When you insert a key-value pair into a dictionary, the key is hashed (converted into a unique integer). This hash is used to quickly locate the associated value. If the key were mutable, its value could change after insertion, leading to incorrect lookups or loss of data because the hash value would no longer match the key’s original hash. Immutable keys ensure the consistency and integrity of the dictionary. Efficient Lookups:

Dictionary operations like checking if a key exists, adding new key-value pairs, or retrieving values are highly efficient because of hashing. Immutable keys ensure that the hash values remain constant, making lookups fast and reliable.

Examples of Immutable and Mutable Objects: Immutable Objects (can be used as dictionary keys): Integers: 1, 42 Strings: "hello", "python" Tuples (with only immutable elements): ("x", 1) Mutable Objects (cannot be used as dictionary keys): Lists: [1, 2, 3] Sets: {"a", "b", "c"} Dictionaries: {"key": "value"}

Example: Using Immutable Keys in a Dictionary:

Using strings (immutable) as keys

my_dict = { "name": "Alice", "age": 25 } print(my_dict["name"]) # Output: Alice

Example: Using Tuples (Immutable) as Keys:

Using a tuple (immutable) as a dictionary key

coordinates = { (40.7128, -74.0060): "New York", (34.0522, -118.2437): "Los Angeles" } print(coordinates[(40.7128, -74.0060)]) # Output: New York