Que 1. Discuss string slicing and provide examples

String slicing is a technique used to extract a specific portion or substring from a given string. It allows you to create new strings based on existing ones, making it a valuable tool for string manipulation.

Syntax:

```
string[start:end:step]
```

- `string`: The original string you want to slice.
- `start`: The index of the first character to include in the substring (default is 0).
- `end`: The index of the first character not to include in the substring (default is the length of the string).
- `step`: An optional parameter specifying the step size (default is 1).

Examples:

1. **Extracting a substring:**

```
string = "Hello, world!"
substring = string[7:12]  # Output: "world"
```

2. **Slicing from the beginning:**

```
string = "Python programming"
substring = string[:6]  # Output: "Python"
```

3. **Slicing to the end:**

```
string = "Hello, world!"
substring = string[7:]  # Output: "world!"
```

4. **Slicing with a step:**

```
string = "Hello, world!"
substring = string[::2]  # Output: "Hlo ol"
```

Key Points:

- If `start` is omitted, it defaults to 0 (the beginning of the string).
- If `end` is omitted, it defaults to the length of the string (the end of the string).
- If `step` is omitted, it defaults to 1 (extract every character).
- A negative `step` value reverses the substring.

Que. 2 Explain the key features of lists in Python

Key Features of Lists in Python:

1. Ordered Sequence:

- Elements in a list are stored in a specific order.
- You can access elements by their index (starting from 0).

2. Mutable:

- You can modify elements of a list after it's created.
- You can add, remove, or change elements.

3. Heterogeneous:

- Lists can contain elements of different data types (e.g., integers, strings, floats).

4. Dynamically Sized:

- Lists can grow or shrink as needed. You don't need to specify the size upfront.

5. Indexing and Slicing:

- You can access individual elements using their index (e.g., `list[0]`).
- You can extract sublists (slices) using slicing syntax (e.g., `list[1:3]`).

6. Common Operations:

- **Append:** Add an element to the end of the list.
- **Insert:** Insert an element at a specific index.
- **Remove:** Remove an element by its value or index.
- **Pop:** Remove an element from a specific index and return its value.
- **Extend:** Add elements from another list to the end of the current list.
- **Sort:** Sort the elements in ascending or descending order.
- **Reverse:** Reverse the order of the elements.
- **Length:** Get the number of elements in the list using `len(list)`.

Example:

```
my_list = [1, "hello", 3.14, True]

# Accessing elements
print(my_list[0])  # Output: 1
print(my_list[2])  # Output: 3.14

# Modifying elements
my_list[1] = "world"
print(my_list)  # Output: [1, 'world', 3.14, True]

# Adding elements
my_list.append(5)
print(my_list)  # Output: [1, 'world', 3.14, True, 5]

# Removing elements
my_list.remove(3.14)
print(my_list)  # Output: [1, 'world', True, 5]
```

Lists are a versatile and fundamental data structure in Python, providing a flexible way to store and manipulate collections of data.

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

Accessing Elements in a List

To access an element in a list, you use its index. The index starts from 0, so the first element has an index of 0.

Example:

```
my_list = [1, 2, 3, 4, 5]
first_element = my_list[0]  # Accesses the first element
third_element = my_list[2]  # Accesses the third element
```

Modifying Elements in a List

To modify an element in a list, you can simply assign a new value to it using its index.

Example:

```
my_list = [1, 2, 3, 4, 5]
my_list[2] = 10
print(my_list)  # Output: [1, 2, 10, 4, 5]
```

Deleting Elements in a List

You can delete elements from a list using the `del` keyword or the `pop()` method.

Using `del`:

```
my_list = [1, 2, 3, 4, 5]
del my_list[2]  # Deletes the third element
print(my_list)  # Output: [1, 2, 4, 5]
```

Using `pop()`:

```
my_list = [1, 2, 3, 4, 5]
removed_element = my_list.pop(2)  # Removes the third element and returns
its value
print(my_list)  # Output: [1, 2, 4, 5]
print(removed_element)  # Output: 3
```

Additional Notes:

- You can use negative indices to access elements from the end of the list (e.g., `my_list[-1]` accesses the last element).

- You can slice lists to extract sublists (e.g., `my_list[1:3]` extracts elements 1 and 2).
- Python lists are mutable, so modifying one list doesn't affect other lists that reference the same elements.

Que.4. Compare and contrast tuples and lists with examples

Tuples vs. Lists in Python

Both tuples and lists are used to store collections of elements in Python, but they have key differences in terms of mutability and usage.

Tuples:

- **Immutable:** Once created, tuples cannot be modified. Their elements are fixed.
- **Used for:**
  - Storing related data that should not be changed.
  - Creating immutable key-value pairs in dictionaries.
  - Returning multiple values from functions.
- **Syntax:** Defined using parentheses `()`.

Example:

```
my_tuple = (1, 2, 3, "hello")
```

Lists:

- **Mutable:** Elements can be added, removed, or modified after creation.
- **Used for:**
  - Storing collections of elements that need to be changed dynamically.
  - Implementing data structures like stacks, queues, and linked lists.
- **Syntax:** Defined using square brackets `[]`.

Example:

```
my_list = [1, 2, 3, "hello"]
```

Key Differences:

Feature	Tuples	Lists
Mutability	Immutable	Mutable
Syntax	Parentheses `()`	Square brackets `[]`

Use Cases	Storing unchanging data, key-value pairs, function returns	Dynamic data structures, collections

Comparison:

Operation	Tuples	Lists
Accessing elements	`tuple[index]`	`list[index]`
Adding elements	Not possible	`list.append(element)`
Removing elements	Not possible	`list.remove(element)`
Modifying elements	Not possible	`list[index] = new_value`

In summary:

- **Tuples** are ideal for storing data that should remain constant throughout the program.
- **Lists** are more flexible and suitable for scenarios where you need to modify the collection of elements dynamically.

The choice between tuples and lists depends on the specific requirements of your application.

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

Key Features of Sets in Python:

- **Unordered:** Elements in a set do not have a specific order.
- **Unique:** Each element in a set must be unique. Duplicate elements are not allowed.
- **Mutable:** You can add or remove elements from a set after it's created.
- **Unchangeable:** Individual elements within a set are immutable (cannot be changed).
- **Hashing:** Sets are implemented using hash tables, which allows for efficient membership testing and insertion.

Common Operations:

- **Adding elements:**
  - `set.add(element)`: Adds an element to the set if it doesn't already exist.
- **Removing elements:**
  - `set.remove(element)`: Removes a specific element from the set.
  - `set.discard(element)`: Removes an element if it exists, but doesn't raise an error if it doesn't.
  - `set.pop()`: Removes and returns an arbitrary element from the set.
- **Union:** Creates a new set containing all elements from both sets.
  - `set1 | set2`
- **Intersection:** Creates a new set containing elements that are common to both sets.
  - `set1 & set2`
- **Difference:** Creates a new set containing elements that are in set1 but not in set2.

    - `set1 - set2`
- **Symmetric difference:** Creates a new set containing elements that are in either set1 or set2, but not both.
    - `set1 ^ set2`

Example:

```
my_set = {1, 2, 3, 4}
my_set.add(5)
print(my_set)  # Output: {1, 2, 3, 4, 5}

other_set = {3, 4, 6}

union_set = my_set | other_set
print(union_set)  # Output: {1, 2, 3, 4, 5, 6}

intersection_set = my_set & other_set
print(intersection_set)  # Output: {3, 4}
```

Sets are often used for:

- **Membership testing:** Quickly checking if an element is in a set.
- **Removing duplicates:** Creating a set from a list can remove duplicates.
- **Set operations:** Performing operations like union, intersection, and difference.
- **Implementing algorithms:** Sets are used in various algorithms, such as graph algorithms and data structures.

Que_6. Discuss the use cases of tuples and sets in Python programming

Tuples and Sets in Python: A Comparison

Tuples and sets are both data structures in Python, but they serve different purposes and have distinct characteristics.

Tuples

- **Immutable:** Once created, tuples cannot be modified.
- **Used for:**
    - Storing related data that should remain constant.
    - Creating immutable key-value pairs in dictionaries.
    - Returning multiple values from functions.

Use Cases:

- **Representing fixed data:** When you have a collection of items that should not be changed, such as the days of the week or the months of the year.
- **Creating key-value pairs:** Tuples can be used as keys in dictionaries, providing an immutable and hashable data type.
- **Function returns:** Returning multiple values from a function as a tuple.

Sets

- **Unordered:** Elements in a set do not have a specific order.
- **Unique:** Each element in a set must be unique. Duplicates are not allowed.
- **Mutable:** Elements can be added or removed from a set.
- **Used for:**
    - Membership testing: Quickly checking if an element is in a set.
    - Removing duplicates from a list.
    - Set operations like union, intersection, difference, and symmetric difference.

Use Cases:

- **Membership testing:** Efficiently checking if an element exists in a collection.
- **Data cleaning:** Removing duplicate values from a dataset.
- **Mathematical operations:** Performing set operations like union, intersection, and difference.
- **Graph algorithms:** Implementing graph algorithms that involve sets of vertices or edges.

In Summary:

- **Tuples** are best suited for scenarios where you need a fixed collection of elements that should not be modified.
- **Sets** are ideal for situations where you need to efficiently check for membership, remove duplicates, or perform set operations.

The choice between tuples and sets depends on the specific requirements of your application and the nature of the data you're working with.

Que_8. Describe how to add, modify, and delete items in a dictionary with examples

Adding Items to a Dictionary

To add a new item to a dictionary, you can use the following methods:

1. Using square brackets:

```
my_dict = {'key1': 'value1', 'key2': 'value2'}
my_dict['key3'] = 'value3'  # Adds a new key-value pair
print(my_dict)  # Output: {'key1': 'value1', 'key2': 'value2', 'key3':
'value3'}
```

2. Using the `update()` method:

```
my_dict = {'key1': 'value1', 'key2': 'value2'}
my_dict.update({'key3': 'value3', 'key4': 'value4'})
print(my_dict)  # Output: {'key1': 'value1', 'key2': 'value2', 'key3':
'value3', 'key4': 'value4'}
```

Modifying Items in a Dictionary

To modify the value associated with an existing key, you can simply assign a new value to it.

```
my_dict = {'key1': 'value1', 'key2': 'value2'}
my_dict['key1'] = 'new value'
print(my_dict)  # Output: {'key1': 'new value', 'key2': 'value2'}
```

Deleting Items from a Dictionary

To delete an item from a dictionary, you can use the `del` keyword or the `pop()` method.

Using `del`:

```
my_dict = {'key1': 'value1', 'key2': 'value2'}
del my_dict['key1']
print(my_dict)  # Output: {'key2': 'value2'}
```

Using `pop()`:

```
my_dict = {'key1': 'value1', 'key2': 'value2'}
removed_value = my_dict.pop('key1')
print(my_dict)  # Output: {'key2': 'value2'}
print(removed_value)  # Output: 'value1'
```

Remember:

- Dictionaries are unordered, so the order of elements may not be preserved.
- Keys in a dictionary must be unique. Trying to add a key that already exists will overwrite the existing value.

Que_9. Discuss the importance of dictionary keys being immutable and provide examples

Importance of Immutable Dictionary Keys

In Python, dictionary keys must be immutable objects. This means that they cannot be changed after they are created. This is a fundamental requirement for dictionaries to function efficiently and correctly.

Reasons for Immutability:

1. **Hashing:** Dictionaries use hashing to store and retrieve key-value pairs. The hash value of a key is used to determine its location in the dictionary. If keys were mutable, their hash values could change, leading to unpredictable behavior and potential errors.

2. **Consistency:** Immutable keys ensure that the dictionary remains consistent and predictable. If keys could be changed, it would be difficult to find the corresponding value, as the key's hash value would no longer match the stored location.
3. **Efficiency:** Immutable objects are often more efficient to store and compare than mutable objects. This can improve the performance of dictionary operations.

Examples of Immutable Objects:

- **Numbers:** Integers, floats, and complex numbers are immutable in Python.
- **Strings:** Strings are also immutable, meaning you cannot modify individual characters within a string.
- **Tuples:** Tuples are immutable collections of elements.

Examples of Mutable Objects (not allowed as dictionary keys):

- **Lists:** Lists can be modified by adding, removing, or changing elements.
- **Dictionaries:** Dictionaries themselves are mutable.
- **Sets:** Sets can be modified by adding or removing elements.

In summary, the immutability of dictionary keys is essential for the efficient and reliable operation of dictionaries in Python. By using immutable objects as keys, you can ensure that the dictionary remains consistent and that key-value pairs can be retrieved efficiently.