1. **Discuss string slicing and provide examples.**

String slicing is a powerful feature in programming that allows you to extract a subset of characters from a string. It is commonly used in text processing and manipulation tasks.

Syntax of String Slicing ->>

String[start:stop:step]

Example :-

Name = “Himanshu”

Name[0] =’H’

Name[1:4]=’ima’

Name[:-8] = ‘u’

Name[0:8:2] = ‘Hmnh’

2. **Explain the key features of lists in Python.**

Lists are a data structure in Python, used to store collections of items that can be of any data type, including strings, integers, floats, and other lists.

Here are the key features of lists in Python:

(i) Ordered Collection :-- Lists are ordered collections, meaning that the order of the items in the list matters. Each item in the list has an index, which is a numerical value that identifies its position in the list.

(ii)Mutable :-- Lists are mutable, meaning that they can be modified after creation. You can add, remove, or modify items in a list.

(iii) Dynamic Size :-- Lists can grow or shrink dynamically as items are added or removed.

(iv) Indexing and Slicing :-- Lists support indexing and slicing, which allows you to access specific items or ranges of items in the list.

(v) Iterability :-- Lists are iterable, meaning that you can loop over the items in the list using a ‘`for`’ loop or other iteration constructs.

(vi) Methods and Operations :-- Lists have a range of built-in methods and operations that can be used to manipulate and transform the list, such as ‘`append`’,’ `insert`’,’ `remove`’, ‘`sort`’, and ‘`reverse`’.

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

List is a data structure that stores collections of items that can be of any data type include string , integer , float , boolean , complex. In list we can access , modify , and delete elements.

(i) Accessing elements in a list:-- To access an element in a list, you can use its index. The index is the position of the element in the list, starting from 0.

Example:--
```
# Create a list
my_list = [1, 2, 3, 4, 5]

# Access the first element
print(my_list[0])          # Output: 1

# Access the third element
print(my_list[2])          # Output: 3
```

(ii) Modifying Elements in a List :-- To modify an element in a list, you can assign a new value to its index.

Example:--
```
# Create a list
my_list = [1, 2, 3, 4, 5]

# Modify the second element
my_list[1] = 10
print(my_list)             # Output: [1, 10, 3, 4, 5]
```

(iii) Deleting Elements in a List :-- To delete an element in a list, you can use the ' `del'` statement or the '`remove()'` method.

Example :--
```
# Create a list
my_list = [1, 2, 3, 4, 5]

# Delete the second element using del

del my_list[1]
print(my_list)                  # Output: [1, 3, 4, 5]

# Delete the first occurrence of 3 using remove()

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

4. Compare and contrast tuples and lists with examples.

Tuples vs Lists: A Comparison

Tuples and lists are two data structures in Python. While they share some similarities, they have distinct differences in terms of their characteristics, usage, and applications.

Similarities

- Both tuples and lists are ordered collections of elements.
- Both can contain elements of different data types, including strings, integers, floats, and other tuples or lists.
- Both support indexing, slicing, and iteration.

Differences

- **Immutability:** Tuples are immutable, meaning their contents cannot be modified after creation. Lists, on the other hand, are mutable, allowing elements to be added, removed, or modified.
- **Syntax:** Tuples are defined using parentheses '`()`' and elements are separated by commas. Lists are defined using square brackets '`[]`' and elements are also separated by commas.
- **Performance:** Tuples are generally faster and more memory-efficient than lists because they are immutable.

Example of Tuples

```
# Create a tuple

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

# Try to modify a tuple (will raise an error)

Try:

  my_tuple[0] = 10

except TypeError:

   print("Tuples are immutable!")
```

```
# Accessing elements in a tuple

print(my_tuple[0])              # Output: 1

# Slicing a tuple

print(my_tuple[1:3])            # Output: (2, 3)
```

Example of Lists

```
# Create a list

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

# Modify a list

my_list[0] = 10

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

# Accessing elements in a list

print(my_list[0])               # Output: 10

# Slicing a list

print(my_list[1:3])             # Output: [2, 3]
```

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

Sets in Python :-- Sets are an unordered collection of unique elements in Python. They are used to store multiple items in a single variable. Sets are mutable, meaning they can be modified after creation.

Key Features of Sets

- **Unordered:** Sets do not maintain the order of elements.
- **Unique elements:** Sets only store unique elements, duplicates are automatically removed.
- **Mutable:** Sets can be modified after creation.
- **Fast lookups:** Sets provide fast membership testing and lookup operations.

Example :--

```
# Create a set using the set() function

my_set = set([1, 2, 3, 4, 5])

print(my_set)          # Output: {1, 2, 3, 4, 5}

# Create a set using the {} syntax

my_set = {1, 2, 3, 4, 5}

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

Set Operations

Sets provide various operations for manipulating and combining sets.

- **Union:** Combines two sets into a new set containing all elements from both sets.
- **Intersection:** Returns a new set containing only the elements common to both sets.
- **Difference:** Returns a new set containing only the elements in the first set but not in the second set.
- **Symmetric difference:** Returns a new set containing only the elements in either set but not in both.

```
# Create two sets
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}
```

```
# Union
print(set1 | set2)              # Output: {1, 2, 3, 4, 5, 6}

# Intersection
print(set1 & set2)              # Output: {3, 4}

# Difference
print(set1 - set2)              # Output: {1, 2}

# Symmetric difference
print(set1 ^ set2)              # Output: {1, 2, 5, 6}
```

Adding and Removing Elements :-- Elements can be added to a set using the `add()` method, and removed using the `remove()` or `discard()` methods.

Example :--

```
# Create a set
my_set = {1, 2, 3}

# Add an element
my_set.add(4)
print(my_set)              # Output: {1, 2, 3, 4}

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

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

Use Cases of Tuples and Sets in Python Programming

Tuples and sets are two fundamental data structures in Python, each with their unique characteristics and use cases.

Use Cases of Tuples

1. Immutable Data :-- Tuples are ideal for storing immutable data, such as constants or configuration values, that should not be modified after creation.

2. Function Return Values :-- Tuples are often used as return values from functions, especially when multiple values need to be returned.

3. Dictionary Keys :- Tuples can be used as dictionary keys because they are hashable, whereas lists are not.

4. Data Structures :-- Tuples can be used to create complex data structures, such as trees or graphs, where the structure needs to be immutable.

5. Performance-Critical Code :-- Tuples are generally faster and more memory-efficient than lists, making them suitable for performance-critical code.

6. Named Tuples :-- Named tuples, a subclass of tuples, can be used to create lightweight, immutable objects with named fields.

Use Cases of Sets

1. Removing Duplicates :-- Sets are ideal for removing duplicates from a list or other iterable.

2. Fast Membership Testing :-- Sets provide fast membership testing, making them useful for checking if an element is in a collection.

3. Combining Collections :-- Sets can be used to combine multiple collections into a single collection, removing duplicates in the process.

4. Mathematical Operations :-- Sets can be used to perform mathematical operations, such as union, intersection, and difference, on collections.

5. Data Validation :-- Sets can be used to validate data, such as checking if a value is in a set of allowed values.

6. Caching :-- Sets can be used as a cache to store unique values, reducing the need for duplicate calculations or database queries.

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

Working with Dictionaries in Python :-- Dictionaries are a fundamental data structure in Python, allowing you to store and manipulate key-value pairs. Here's a comprehensive guide on how to add, modify, and delete items in a dictionary.

Creating a Dictionary :-- A dictionary can be created using the '`{}`' syntax or the '`dict()`' constructor.

```
# Create a dictionary using the {} syntax
my_dict = {'name': 'John', 'age': 30}

# Create a dictionary using the dict() constructor
my_dict = dict(name='John', age=30)
```

Adding Items to a Dictionary :-- You can add a new item to a dictionary by assigning a value to a new key.

Example :--

```
my_dict = {'name': 'John', 'age': 30}

# Add a new item
my_dict[' occupation'] = 'Developer'

print(my_dict)        # Output: {'name': 'John', 'age': 30, 'occupation': 'Developer'}
```

Modifying Items in a Dictionary :-- You can modify an existing item in a dictionary by assigning a new value to an existing key.

Example :--

```
my_dict = {'name': 'John', 'age': 30}

# Modify an existing item
my_dict['age'] = 31

print(my_dict)              # Output: {'name': 'John', 'age': 31}
```

Deleting Items from a Dictionary :-- You can delete an item from a dictionary using the ‘`del`’ statement or the ’ `pop()`’ method.

Example :--

```
my_dict = {'name': 'John', 'age': 30, 'occupation': 'Developer'}

# Delete an item using del
del my_dict['occupation']

print(my_dict)               # Output: {'name': 'John', 'age': 30}

# Delete an item using pop()
my_dict.pop('age')

print(my_dict)              # Output: {'name': 'John'}
```

Updating a Dictionary :-- You can update a dictionary using the ‘`update()`’ method, which merges the key-value pairs from another dictionary.

Example :--

```
my_dict = {'name': 'John', 'age': 30}

# Update the dictionary
my_dict.update({' occupation': 'Developer', 'country': 'USA'})

print(my_dict)          # Output: {'name': 'John', 'age': 30, 'occupation': 'Developer', 'country': 'USA'}
```

Accessing Dictionary Items :-- You can access dictionary items using their keys.

Example :--

```
my_dict = {'name': 'John', 'age': 30}

# Access an item
print(my_dict['name'])          # Output: John

# Access an item with a default value
print(my_dict.get(' occupation', 'Unknown'))        # Output: Unknown
```

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

The Importance of Dictionary Keys Being Immutable

In Python, dictionaries are a fundamental data structure that stores key-value pairs. The keys in a dictionary play a crucial role in accessing and manipulating the associated values. One of the essential characteristics of dictionary keys is that they must be immutable.

Why Dictionary Keys Must Be Immutable

Dictionary keys must be immutable because they are used to hash and store the key-value pairs in the dictionary. Hashing is a process that converts an object into a fixed-size integer, which is used to store and retrieve the object from a hash table.

If a dictionary key were mutable, it could change its hash value after being stored in the dictionary. This would lead to unpredictable behavior , as the dictionary would not be able to find the associated value when the key is modified.

Examples of Immutable Dictionary Keys

Here are some examples of immutable dictionary keys:

1. Strings :-- Strings are immutable in Python, making them an ideal choice for dictionary keys.

Example :-

```
my_dict = {'hello': 'world'}
print(my_dict['hello'])        # Output: world
```

2. Integers :-- Integers are immutable in Python, making them suitable for dictionary keys.

Example :-

```
my_dict = {1: 'one', 2: 'two'}
print(my_dict[1])          # Output: one
```

3. Tuples :-- Tuples are immutable in Python, making them a good choice for dictionary keys.

Example :-

```
my_dict = {(1, 2): 'pair', (3, 4): 'another pair'}
print(my_dict[(1, 2)])        # Output: pair
```

Examples of Mutable Objects That Cannot Be Dictionary Keys

Here are some examples of mutable objects that cannot be used as dictionary keys:

1. Lists :-- Lists are mutable in Python, making them unsuitable for dictionary keys.

Examples :--

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

my_dict = {my_list: 'list'}              # TypeError: unhashable type: 'list'
```

2. Dictionaries :-- Dictionaries are mutable in Python, making them unsuitable for dictionary keys.

Examples :--

```
my_dict1 = {'a': 1, 'b': 2}

my_dict2 = {my_dict1: 'dict'}          # TypeError: unhashable type: 'dict'
```

3. Sets :-- Sets are mutable in Python, making them unsuitable for dictionary keys.

Examples :--

```
my_set = {1, 2, 3}

my_dict = {my_set: 'set'}              # TypeError: unhashable type: 'set'
```

Link of this page :--
https://docs.google.com/document/d/1o5LGlKjKsvJZmg0WbXaN3Rtztte6xaQiJN_TKMvWbQY/edit