```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


# 1. Discuss string slicing and provide examples.

String slicing is a technique in programming languages, especially in
Python, that allows you to extract substrings from a given string. It
involves specifying a start index, an end index, and optionally a step
value.
The syntax for slicing is:-
string[start:end:step]

Here’s what each parameter means:-
- start: The index where the slicing begins. If omitted, it defaults to
the beginning of the string.

- end: The index where the slicing ends. The character at this index is
not included in the result. If omitted, it defaults to the end of the
string.

- step: The step size between characters. If omitted, it defaults to 1,
meaning it includes every character between start and end.

Examples:-
text = "Hello, World!"
slice1 = text[0:5]  # 'Hello'
print(slice1)

text = "Hello, World!"
slice2 = text[:5]   # 'Hello'
slice3 = text[7:]   # 'World!'
print(slice2)
print(slice3)

text = "Hello, World!"
slice4 = text[::2]  # 'Hlo ol!'
print(slice4)

text = "Hello, World!"
slice5 = text[-6:-1]  # 'World'
print(slice5)

text = "Hello, World!"
slice6 = text[::-1]  # '!dlroW ,olleH'
print(slice6)
# In[2]:


# Explain the key features of lists in Python.

Lists are a versatile and widely-used data structure that allows you to
store and manipulate collections of items.
Here are some key features of lists:-
```

1.Lists maintain the order of elements as they are added. Each item in a list has a specific index, starting from 0.
2.Lists are mutable, meaning you can change their content after creation. You can add, remove, or modify elements.
3.Lists can grow or shrink in size as needed. You don’t need to specify the size of the list at the time of creation.
4.Lists can hold items of different data types. You can mix integers, strings, and other objects in a single list.
5.Lists can contain duplicate values. There is no restriction on having multiple occurrences of the same value.
6.You can use negative indices to access elements from the end of the list, e.g., my_list[-1] accesses the last item.
7.You can obtain a sublist using slicing. For example, my_list[1:4] gives you a list containing elements from index 1 to 3.
8.Lists come with various built-in methods for common operations, such as append(), extend(), remove(), pop(), insert(), and sort().
9.You can iterate over a list using loops, such as for loops, to process each element.

```
# In[3]:


# DescrIbe how to access, modify and delete elements in a list with
examples
```

1. Accessing Elements
You can access elements in a list using indexing. List indices start at 0 for the first element and go up to len(list) - 1 for the last element. You can also use negative indexing to access elements from the end of the list.

Examples:-

```
my_list = [10, 20, 30, 40, 50]

# Accessing elements by positive index
print(my_list[0])   # Output: 10 (first element)
print(my_list[2])   # Output: 30 (third element)

# Accessing elements by negative index
print(my_list[-1])  # Output: 50 (last element)
print(my_list[-3])  # Output: 30 (third element from the end)
```

2. Modifying Elements
You can modify elements by accessing them via their index and then assigning a new value to that index.

Examples:-

```
my_list = [10, 20, 30, 40, 50]

# Modifying an element
my_list[1] = 25
print(my_list)  # Output: [10, 25, 30, 40, 50]

# Modifying multiple elements using slicing
my_list[2:4] = [35, 45]
print(my_list)  # Output: [10, 25, 35, 45, 50]
```

3. Deleting Elements

You can delete elements using the del statement, the pop() method, or the remove() method.

Examples:-
```
my_list = [10, 20, 30, 40, 50]

# Deleting an element by index
del my_list[2]
print(my_list)  # Output: [10, 20, 40, 50]

# Deleting a slice of elements
del my_list[1:3]
print(my_list)  # Output: [10, 50]
# In[4]:


# Compare and contrast tuples and lists with examples.
```

Tuples and lists are both built-in data structures in Python used to store collections of items. However, they have different characteristics and use cases.
Here’s a comparison of tuples and lists:-

1.a-Lists: Mutable. You can modify, add, or remove elements after the list is created.

```
my_list = [1, 2, 3]
my_list[1] = 20       # Modify an element
my_list.append(4)    # Add an element
my_list.remove(2)    # Remove an element
print(my_list)       # Output: [1, 20, 4]
```

1.b-Tuples: Immutable. Once a tuple is created, you cannot change its elements, add new ones, or remove existing ones.

```
my_tuple = (1, 2, 3)
# my_tuple[1] = 20    # This will raise a TypeError
# my_tuple.append(4) # This will raise an AttributeError
# my_tuple.remove(2) # This will raise an AttributeError
print(my_tuple)     # Output: (1, 2, 3)
```

2.a-Lists: Defined with square brackets [].

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

2.b-Tuples: Defined with parentheses ().

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

3.a-Lists: Have several built-in methods such as append(), extend(), remove(), pop(), sort(), and reverse().

```
my_list = [1, 2, 3]
my_list.append(4)
my_list.sort()
print(my_list)  # Output: [1, 2, 3, 4]
```

3.b-Tuples: Have fewer built-in methods, primarily count() and index().

```
my_tuple = (1, 2, 3, 1)
print(my_tuple.count(1))  # Output: 2 (counts occurrences of 1)
print(my_tuple.index(2))  # Output: 1 (finds index of first occurrence of
2)
```

4.a-Lists: Generally have more overhead due to their mutability. This means that operations on lists can be slightly slower compared to tuples.

```
import timeit

list_time = timeit.timeit('my_list.append(5)', setup='my_list = [1, 2,
3]', number=10000)
print(f'List time: {list_time}')
```

4.b-Tuples: Typically have less overhead and can be more efficient in terms of performance, especially when used as keys in dictionaries or as elements in sets.

```
tuple_time = timeit.timeit('my_tuple + (5,)', setup='my_tuple = (1, 2,
3)', number=10000)
print(f'Tuple time: {tuple_time}')
```

5.a-Lists: Ideal when you need a collection that can be modified. Suitable for tasks where elements will change or need to be reordered.

```
shopping_list = ['milk', 'bread', 'eggs']
shopping_list.append('butter')  # Adding a new item
shopping_list[1] = 'whole grain bread'  # Changing an existing item
```

5.b-Tuples: Best used when you have a collection of items that should not be modified. Commonly used to represent fixed collections of items, like coordinates or records.

```
point = (10, 20)  # Coordinates
coordinates = (1, 2, 3)  # Immutable collection of values
# In[5]:
```

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

Sets in Python are a built-in data structure used to store collections of unique elements.

Here are the key features of sets along with examples of their use:

1.Sets do not maintain the order of elements. The order in which elements are added is not necessarily the order in which they are stored or iterated.
2.Sets automatically eliminate duplicate elements. Each element in a set must be unique.
3.Sets are mutable, meaning you can add and remove elements after the set is created. However, the elements themselves must be immutable (e.g., numbers, strings, tuples).
4.Unlike lists and tuples, sets do not support indexing, slicing, or other sequence-like behaviors.

5.Sets provide efficient membership testing due to their underlying hash-based implementation. Checking if an item is in a set is generally faster compared to lists.
6.Sets support mathematical operations like union, intersection, difference, and symmetric difference, which are useful for set theory operations.

Examples-

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

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

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

# Removing an element that does not exist (will raise KeyError)
# my_set.remove(10)  # Uncommenting this line will raise an error

# Using discard() method (no error if element is not found)
my_set.discard(10)
print(my_set)  # Output: {1, 3, 4}

# In[6]:


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

Tuples and sets are versatile data structures in Python, each with unique properties that make them suitable for different use cases.

Use Cases for Tuples:-

1.Tuples are ideal when you need a collection of items that should not change. This immutability ensures that the data remains constant throughout the program.
2.Because tuples are immutable, they are suitable for data that should not be accidentally modified, which can help maintain data integrity.
3.Tuples can be used to return multiple values from a function. They allow you to group related pieces of data together.
4.Tuples support unpacking, allowing you to assign multiple variables in a single statement. This is useful for handling multiple values returned by functions or iterating over items.
5.Tuples can be used as keys in dictionaries or elements in sets, whereas lists cannot be used in this way because they are mutable.

Use Cases for Sets:-
1.Sets automatically remove duplicate elements, making them ideal for filtering out duplicates from a list.
2.Sets provide fast membership testing due to their hash-based implementation. This makes them useful for checking if an element exists in a collection efficiently.
3.Sets support mathematical operations such as union, intersection, difference, and symmetric difference, which are useful in various analytical tasks.

4.When you need to ensure that a collection of items does not contain duplicates and maintain uniqueness, sets are the ideal choice.
5.For large datasets where operations like membership testing or deduplication are frequent, sets offer better performance compared to lists due to their underlying hash-based structure.

```
# In[7]:


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

Dictionaries in Python are collections of key-value pairs where each key is unique. They are mutable, meaning you can add, modify, and delete items after the dictionary has been created.

Here’s how you can perform these operations:-
1. Adding Items
You can add items to a dictionary by assigning a value to a new key. If the key does not already exist in the dictionary, it will be created.

```
# Creating a dictionary
my_dict = {'a': 1, 'b': 2}

# Adding a new key-value pair
my_dict['c'] = 3
print(my_dict)  # Output: {'a': 1, 'b': 2, 'c': 3}

# Adding multiple key-value pairs using the `update()` method
my_dict.update({'d': 4, 'e': 5})
print(my_dict)  # Output: {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
```

2. Modifying Items
You can modify the value associated with an existing key by reassigning it.

```
# Creating a dictionary
my_dict = {'a': 1, 'b': 2, 'c': 3}

# Modifying an existing key-value pair
my_dict['b'] = 20
print(my_dict)  # Output: {'a': 1, 'b': 20, 'c': 3}

# Modifying multiple key-value pairs using the `update()` method
my_dict.update({'a': 10, 'c': 30})
print(my_dict)  # Output: {'a': 10, 'b': 20, 'c': 30}
```

3. Deleting Items
You can delete items from a dictionary using the del statement or the pop() method. The pop() method also returns the value of the removed key, while del does not.

```
# Creating a dictionary
my_dict = {'a': 1, 'b': 2, 'c': 3}

# Deleting a key-value pair
del my_dict['b']
print(my_dict)  # Output: {'a': 1, 'c': 3}
```

```
# Deleting a key-value pair that does not exist will raise a KeyError
# del my_dict['d']  # Uncommenting this line will raise KeyError
# In[8]:
```

```
# Discuss the importance of dictionary keys being immutable and provide
example.
```

Dictionary keys must be immutable. This is a crucial characteristic because it ensures the integrity and efficiency of the dictionary's data structure.
Here's a detailed discussion of why dictionary keys need to be immutable, along with examples:

1.Hashing Requirement:

Dictionaries in Python are implemented using a hash table. Hash tables use a hash function to compute an index where the key-value pair should be stored. This index is used to quickly locate the value associated with a given key.
For the hash table to function correctly, the hash value of a key must remain constant throughout its lifetime. If keys were mutable and could change, their hash values could change, leading to inconsistent or incorrect behavior when looking up values.

2.Data Integrity:

Immutable keys ensure that once a key is added to the dictionary, it cannot be altered. This helps maintain the integrity of the dictionary’s internal structure and ensures that the key-value pair relationships remain consistent.

3.Efficiency:

Immutable keys enable dictionaries to perform efficiently in terms of lookups and insertions. Since the hash value of a key does not change, the dictionary can quickly locate the key without worrying about its value being modified.

Examples:-

```
# Using a string as a key
my_dict = {'name': 'Alice', 'age': 30}
print(my_dict['name'])  # Output: Alice

# Using a tuple as a key
my_dict2 = {(1, 2): 'point A', (3, 4): 'point B'}
print(my_dict2[(1, 2)])  # Output: point A
# In[ ]:
```