Q.1. Discuss string slicing and provide examples ?

Ans. String slicing in Python is a technique used to extract a portion of a string by specifying a range of indices. The syntax for string slicing is:

```
string_slicing = [start,stop,syep]
```

* start is the index where the slice starts (inclusive).

* stop is the index where the slice ends (exclusive).

* step is the amount by which the index increases.

* If start is omitted, it is assumed to be the beginning of the string (index 0).

* If stop is omitted, it is assumed to be the end of the string (the length of the string).

* If step is omitted, it is assumed to be 1 (each character is included in the slice).

Here are some examples

```
string = "hello Pwskills"
slicing = string[:6]
slicing
```

```
'hello '
```

By default it starts from 0

```
slicing1 = string[6:]
slicing1
```

```
'Pwskills'
```

By default it stops at the length of the string.

```
slicing2 = string[::]
slicing2
```

```
'hello Pwskills'
```

It takes step one (1) by default

These examples show how Python's default values for slicing parameters work, making the slicing operation more flexible and easier to use.

```

```

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

Lists are a fundamental data structure in Python and offer a range of features that make them versatile and powerful. Here are the key features of lists in Python:

(a).Ordered

Lists maintain the order of elements. The order in which elements are added is preserved, and elements can be accessed by their index.

```
my_list = [11, 'pwskills','Data Science', 'Data Analytics',200, 100]
print(my_list[2])  # Output: Data Science
```

```
Data Science
```

(b).Mutable

Lists can be modified after their creation. We can change, add, or remove elements.

```
my_list[2] = 'Machine Learning'
my_list    # here output is [11, 'pwskills', 'Machine Learning', 'Data Analytics', 200, 100]
```

```
[11, 'pwskills', 'Machine Learning', 'Data Analytics', 200, 100]
```

(c).Dynamic Size

Lists can grow and shrink in size as needed. We can append elements to increase the size and remove elements to decrease it.

```
my_list.append(50)
print(my_list)  # Output: [11, 'pwskills', 'Machine Learning', 'Data Analytics', 200, 100, 50]

my_list.pop()
print(my_list)  # Output: [11, 'pwskills', 'Machine Learning', 'Data Analytics', 200, 100]
```

```
[11, 'pwskills', 'Machine Learning', 'Data Analytics', 200, 100, 50]
[11, 'pwskills', 'Machine Learning', 'Data Analytics', 200, 100]
```

(d). Slicing

Lists support slicing to access a subset of elements.

```
print(my_list[1:3])  # Output: ['pwskills', 'Machine Learning']
```

```
['pwskills', 'Machine Learning']
```

(e). Built-in Functions and Methods

Python provides several built-in functions and methods for lists, such as len(), min(), max(), sum(), append(), remove(), sort(), and more.

```
my_list1 = [100,200,250,300,450,60]
print(len(my_list1))   # Output: 6
print(min(my_list1))   # Output: 60
print(max(my_list1))   # Output: 450

my_list1.sort()
print(my_list1)   # to write the list in  ascending order
```

```
6
60
450
[60, 100, 200, 250, 300, 450]
```

(f).Iteration

We can iterate over the elements of a list using loops, making it easy to process each element.

```
for i in my_list1:
    print(i)
```

```
60
100
200
250
300
450
```

(g). List Comprehensions

List comprehensions provide a concise way to create lists. It consists of brackets containing an expression followed by a for clause.

```
my_list2 = [23,43,44,55,66,78,99,100]
even_no = [i for i in my_list2 if i%2==0]
even_no
```

```
[44, 66, 78, 100]
```

These features make lists a versatile and essential data structure in Python, suitable for a wide range of applications from simple data storage to complex data manipulation and processing.

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

Ans. In Python, lists are mutable, meaning their elements can be changed. Here's how you can access, modify, and delete elements in a list with examples:

Accessing Elements

We can access elements in a list using indexing.

```
fruits = ['apple', 'banana', 'cherry', 'date', 'elderberry']
first_fruit = fruits[0]
fruits = ['apple', 'banana', 'cherry', 'date', 'elderberry']
second_fruit = fruits[1]
fruits = ['apple', 'banana', 'cherry', 'date', 'elderberry']
last_fruit = fruits[-1]
print(first_fruit)
print(second_fruit)
print(last_fruit)

apple
banana
elderberry
```

Modifying Elements

We can modify elements in a list by directly assigning a new value to a specific index.

```
fruits[1] = 'blueberry'
print(fruits)
fruits[-1] = 'fig'
print(fruits)

['apple', 'blueberry', 'cherry', 'date', 'elderberry']
['apple', 'blueberry', 'cherry', 'date', 'fig']
```

Deleting Elements

We can delete elements from a list using the del statement, the remove() method, or the pop() method.

Using del statement

The del statement removes an element at a specific index.

```
fruits = ['apple', 'banana', 'cherry', 'date', 'elderberry']
del fruits[1]
print(fruits)
fruits.remove('cherry')
print(fruits)
```

```
['apple', 'cherry', 'date', 'elderberry']
['apple', 'date', 'elderberry']
```

We can also use pop() method removes an element at a specific index and returns it. If no index is specified, it removes and returns the last element.

```
fruits = ['apple', 'banana', 'cherry', 'date', 'elderberry']
popped_fruit = fruits.pop(2)
print(popped_fruit)  # Output: cherry
print(fruits)

cherry
['apple', 'banana', 'date', 'elderberry']
```

These are the basic ways to access, modify, and delete elements in a list in Python.

Q.4.Compare and contrast tuples and list with examples .

Ans. Tuples and lists are both used to store collections of items in Python, but they have several key differences in terms of mutability, performance, and usage. Here's a comparison and contrast of tuples and lists with examples:

Mutability

Lists are mutable, meaning you can change, add, or remove elements after the list is created.

Tuples are immutable, meaning once a tuple is created, we cannot change, add, or remove elements.

Example of List Mutability:

```
list = [1, 2, 3, 4, 5]
list[2] = 10    # Modifying an element
print(list)

list.append(6)   # Adding an element
print(list)

list.remove(10)   # Removing an element
print(list)

[1, 2, 10, 4, 5]
[1, 2, 10, 4, 5, 6]
[1, 2, 4, 5, 6]
```

Example of Tuple Immutability:

```
tuple = (1, 2, 3, 4, 5)
my_tuple[2] = 10  # TypeError: 'tuple' object does not support item
assignment
                  # Tuples are immutable, so we cannot add or remove
elements either.
```

Syntax

Lists are defined using square brackets [].

Tuples are defined using parentheses ().

Use Cases

Lists are suitable for collections of items where the data might need to be modified.

Tuples are suitable for collections of items that should not change throughout the program. They are often used to represent fixed collections of items, like coordinates or dates.

By understanding these differences, we can choose the appropriate data structure based on your specific needs in a Python program.

```

```

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

Ans. Sets in Python are a collection type that is both unordered and unindexed, with no duplicate elements allowed. They are particularly useful for membership testing, eliminating duplicates, and performing mathematical operations like union, intersection, difference, and symmetric difference. Here are the key features of sets with examples:

Key Features of Sets

Unordered Collection: Sets do not maintain any order of elements.

No Duplicate Elements: Sets automatically eliminate duplicate elements.

Mutable: Sets can be modified by adding or removing elements, although the elements themselves must be immutable.

Set Operations: Sets support standard mathematical operations such as union, intersection, difference, and symmetric difference.

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

```
# Using the set() constructor
my_set1 = set([1, 2, 2, 3, 4])
print(my_set1)  # Output: {1, 2, 3, 4} (duplicates are removed)

{1, 2, 3, 4, 5}
{1, 2, 3, 4}
```

Adding and Removing Elements

You can add elements using the add() method and remove elements using the remove() or discard() methods.

```
set = {1, 2, 3}
set.add(4)

set.remove(3)
print(set)

# Discarding an element (doesn't raise an error if the element is not present)
my_set.discard(5)
print(my_set)

{1, 2, 4}
{1, 2, 3, 4}
```

Set Operations

Sets support various mathematical operations such as union, intersection, difference, and symmetric difference.

Union

The union of two sets contains all elements from both sets.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

union_set = set1.union(set2)
print(union_set)  # Output: {1, 2, 3, 4, 5}

# Alternatively, we can use the | operator
union_set = set1 | set2
print(union_set)  # Output: {1, 2, 3, 4, 5}

{1, 2, 3, 4, 5}
{1, 2, 3, 4, 5}
```

Intersection

The intersection of two sets contains only the elements that are present in both sets.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

intersection_set = set1.intersection(set2)
print(intersection_set)  # Output: {3}

# Alternatively,we can use the & operator
intersection_set = set1 & set2
print(intersection_set)  # Output: {3}
```

```
{3}
{3}
```

Sets are a powerful data structure for storing unique elements and performing various mathematical operations efficiently. They are particularly useful for membership testing, removing duplicates from a collection, and performing set operations like union, intersection, and difference.

```

```

Q.6. Disscuss the use cases of sets and tuples in Pytgon programming.

Sets

Removing Duplicates: When we need to ensure unique elements in your data, sets shine. They automatically discard duplicates during creation, making them ideal for tasks like finding unique words in a text or collecting unique user IDs.

Set Operations: Sets provide powerful operations like union (combining elements), intersection (finding common elements), and difference (elements in one set but not the other). These are useful for data analysis, filtering items based on specific criteria.

Membership Testing: Checking if an element exists in a set is a very fast operation. This makes sets efficient for tasks like validating user input or checking if a product exists in a shopping cart.

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

# Convert the list to a set to remove duplicates
unique_elements = set(my_list)

# Print the unique elements
print(unique_elements)  # Output: {1, 2, 3, 4, 5}
```

Function Arguments: Tuples are often used to return multiple values from a function.

```
# Example: Function returning multiple values
def get_min_max(numbers):
    return (min(numbers), max(numbers))

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

Sets: Best for collections where uniqueness is required and membership testing is frequent. Useful for operations like removing duplicates, performing mathematical set operations, and efficient membership testing.

Tuples: Ideal for fixed collections of items, ensuring data integrity, using as dictionary keys, and returning multiple values from functions. Suitable for representing immutable data and for situations where data should not be modified.

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

Ans.Dictionaries in Python provide a powerful way to store and manage data using key-value pairs. Here's how you can add, modify, and delete items within a dictionary:

Adding Items:

Direct Assignment: We can directly assign a value to a new key within the dictionary curly braces.

```
my_dict = {}
my_dict['name'] = 'Pankaj'
print(my_dict)  # Output: {'name': 'Alice'}

# Add another key-value pair
my_dict['age'] = 25
print(my_dict)
```

```
{'name': 'Pankaj'}
{'name': 'Pankaj', 'age': 25}
```

Modifying Items in a Dictionary

We can modify the value of an existing key by reassigning a new value to that key.

```
my_dict = {'name': 'Alice', 'age': 25}

# Modify the value of an existing key
my_dict['age'] = 26
```

```
print(my_dict)  # Output: {'name': 'Alice', 'age': 26}

# Modify another key-value pair
my_dict['name'] = 'Ajay'
print(my_dict)
```

```
{'name': 'Alice', 'age': 26}
{'name': 'Ajay', 'age': 26}
```

The del statement removes the key-value pair associated with a specified key.

```
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}

# Delete a key-value pair
del my_dict['age']
print(my_dict)
```

```
{'name': 'Alice', 'city': 'New York'}
```

The pop() method removes the key-value pair associated with a specified key and returns the value.

```
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}

# Remove a key-value pair and return the value
age = my_dict.pop('age')
print(age)      # Output: 25
print(my_dict)
```

```
25
{'name': 'Alice', 'city': 'New York'}
```

The popitem() method removes and returns the last key-value pair added to the dictionary

```
my_dict = {'name': 'Africa', 'age': 25, 'city': 'New York'}

# Remove and return the last key-value pair
last_item = my_dict.popitem()
print(last_item)  # Output: ('city', 'New York')
print(my_dict)
```

```
('city', 'New York')
{'name': 'Africa', 'age': 25}
```

These operations allow us to effectively manage the data stored in dictionaries in Python.

```
```

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

Ans. The immutability of dictionary keys in Python is a fundamental requirement that ensures the integrity and efficiency of dictionary operations. This design choice is rooted in how dictionaries function, particularly in their reliance on hash values for quick data retrieval.

Importance of Immutable Keys

(a). Hashing Consistency

Dictionaries in Python are implemented as hash tables, which means that they use a hash function to compute a hash value for each key. This hash value determines where the corresponding value is stored in memory. If a key were mutable, its hash value could change over time. This inconsistency would lead to significant issues:

Lookup Failures: If the key's value changes, the dictionary would be unable to locate the value associated with that key because it would be searching for an outdated hash value. For example, if a list were used as a key and its contents were modified, the dictionary would not find the expected entry, leading to errors during lookups.

(b).Consistency and Reliability:

Immutable keys ensure that the mapping between keys and values remains stable throughout the lifetime of the dictionary. This stability is crucial for maintaining the integrity of the dictionary.

(c).Performance:

Immutable objects are often implemented in a way that makes them faster to hash and compare. This improves the overall performance of dictionary operations, such as insertions, lookups, and deletions.

```
person = {
    'name': 'Krishna',
    'age': 30,
    'city': 'New York'
}
print(person['name'])  # Output: Alice

# Using numbers as dictionary keys
inventory = {
    101: 'Apple',
    102: 'Banana',
    103: 'Cherry'
}
print(inventory[101])  # Output: Apple
```

```python
# Using tuples as dictionary keys
coordinates = {
    (40.7128, -74.0060): 'New York',
    (34.0522, -118.2437): 'Los Angeles'
}
print(coordinates[(40.7128, -74.0060)])  # Output: New York
```

```
Krishna
Apple
New York
```

Mutable Types as Dictionary Keys (Not Allowed)

Using mutable types like lists or dictionaries as dictionary keys would result in a TypeError because they are not hashable.

Summary

Hashing Requirement: Immutable types have consistent hash values, making them suitable for dictionary keys.

Consistency and Reliability: Immutable keys ensure stable mappings.

Performance: Immutable objects often have faster hash and comparison operations.

By enforcing the immutability of dictionary keys, Python ensures that dictionaries remain efficient, reliable, and performant.