```
## python assignmnet 3

 Que 1 , discuss string slicing and provide examples
 ans - string slicing helps in extracting specific parts in a string using index ranges
```

```
# format for a  basic string slicing

 string[start:stop:step]

# ex 1
text = "Hello, World!"
print(text[0:5])
```

```
Hello
```

```
# ex.2

print(text[:5])
# omitting to start
```

```
Hello
```

```
# ex. 3
print(text[7:])
# omitting to stop
```

```
World!
```

```
# ex. 4
print(text[::2])
```

```
Hlo ol!
```

```
# ex. 5
print(text[-6:-1])
# negative incdice
```

```
World
```

```
# que 2 . EXplain the key features of lists in Python
# ans
The list class is a fundamental built-in data type in Python. It has an impressive and useful set of features,
allowing you to efficiently organize and manipulate different type of data.

Fetaures of lists are :-

1. Lits are mutable
2. Maintain a order
3. Lists can store different type of data types
4. List can stores other lists in it as elements
5. There is no fixed size for a list; it can hold as many elements as memory allows.
```

```python
# que 3 : Describe how to access , modify , and delete elements in a list with examples .
 # ans = We can use index to access lists in python

my_list = ['apple', 'banana', 'cherry']

# Access first element
print(my_list[0])
# Access last element using negative indexing
print(my_list[-1])

# Access a range of elements using slicing
print(my_list[0:2])
```

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

```python
# ex.2
# we can modify an element by assigning a new value to the index where the element is stored.

my_list = ['apple', 'banana', 'cherry']

# Modify the second element
my_list[1] = 'blueberry'
print(my_list)
# Modify a range of elements
my_list[0:2] = ['grape', 'mango']
print(my_list)
```

```
['apple', 'blueberry', 'cherry']
['grape', 'mango', 'cherry']
```

```python
# ex.3 we can delete elements in a list in several ways one of the way is to use "DEL" Function

my_list = ['apple', 'banana', 'cherry']

# Delete the first element
del my_list[0]
print(my_list)
```

```
['banana', 'cherry']
```

```python
# que 4  Compare and contrast tuples and lists witth examples .
# ans = Tuples and lists are the part of data structure in python but he main difference in between them are as follows :-
#  1. Mutable = Lists are mutable while tuples are not we can add , modify , delete intems in a list but we cant do it in tuples
    #ex. 1

    # List: Can be modified
my_list = [1, 2, 3]
my_list[0] = 100
print(my_list)
# Tuple: Cannot be modified
my_tuple = (1, 2, 3)

#  2.the syntax of a list include [] while tuples include ()
#ex.2
# list
my_list = [1, 2, 3]

# Tuple
my_tuple = (1, 2, 3)

#  3. because of its dynamic nature lists locs performance in compaironce with tuples which are immutables
#  4. Lists: Use more memory due to their dynamic nature.
#      Tuples: More memory-efficient because of immutability.
 # e.t.c
```

```
[100, 2, 3]
```

```
# que 5  Describe the key Features of sets and provide examples of their use.
# ans = 1 , Unique elements = it store unique elements while removing any duplicates , sets use to filter out duplicates in a dataset.
# For example, when tracking unique users visiting a website
{user1, user2, user3} automatically excludes repeated entries.
# 2 , undordered collection = elements in a set are unordered and have no index. The position of items doesn't matter.
    # When we  don't care about the order of items, such as in checking membership or creating sets of attributes for comparison, unordered sets allow fast lookups.
# 3   Set Operations (Union, Intersection, Difference e.t.c)= Sets support mathematical operations like union, intersection, and difference.
  #                                                          Useful in tasks such as finding common customers across two datasets. For example,
#                                                             the intersection of two sets {A, B, C}  and {B, C, D} will give {B, C}, identifying shared customers.

# 4 , Efficient Membership Testing = Sets allow fast membership tests due to their underlying hash table implementation
#                                    In situations like checking whether an element exists in a collection, such as verifying if a word is in a dictionary of banned words,
#                                    sets offer quicker lookups compared to lists.

# 5 ,  Mutable Nature = Sets allow you to add or remove elements after creation (except for frozenset which is immutable).
#                       When we dealing with a dynamic dataset, such as a list of ongoing transactions, we can modify the set
#                        as transactions are added or completed, allowing real-time updates.
```

```python
# que 6 =Discuss the usecase of tuples and sets in python programming
# ans =tuples
#Immutability: Tuples are ordered collections of elements that cannot be modified once created. This makes them suitable for
#representing data that should remain constant, such as coordinates, dimensions, or configuration settings.
#Indexing and Slicing: You can access elements in a tuple using indexing (e.g., tuple[0]) and slicing (e.g., tuple[1:3]).
#Packing and Unpacking: Tuples can be used for packing multiple values into a single variable (e.g., coordinates = (x, y))
#and unpacking them into individual variables (e.g., x, y = coordinates).
#Efficiency: Tuples are generally more efficient than lists in terms of memory usage and performance, especially when dealing with large datasets.

# sets
#Unordered Collections: Sets are collections of unique elements that are not stored in a specific order.
# This makes them ideal for tasks that involve removing duplicates or checking for membership.
#Set Operations: Sets support various operations, including union (|), intersection (&), difference (-),
#and symmetric difference (^). These operations can be used to perform complex data analysis and filtering tasks.
#Membership Testing: Checking if an element exists in a set is a very fast operation,
#making sets efficient for tasks like validating user input or checking if a product exists in a shopping cart.
```

```python
# examples
# Tuples
coordinates = (10, 20)
print(coordinates)

# Sets
fruits = {"apple", "banana", "orange", "apple"}
print(fruits)
# Set operations
fruits1 = {"apple", "banana"}
fruits2 = {"banana", "orange"}
union_fruits = fruits1 | fruits2
print(union_fruits)
```

```
(10, 20)
{'apple', 'banana', 'orange'}
{'apple', 'banana', 'orange'}
```

```
# que 7 = Describe how to add , modify , and delete items in a dictionary with examples .
# ans =

# A dictionary is a collection of key-value pairs. Each key is unique, and it is used to access the corresponding value.
#You can add, modify, and delete items in a dictionary using the following methods:

#Adding items:

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

my_dict = {}
 my_dict['name'] = 'dheeraj'
 print(my_dict)  # Output: {'name': 'Alice'}
 # Add another key-value pair
 my_dict['age'] = 25
 print(my_dict)
 {'name': 'dheeraj'}
 {'name': 'dheeraj', '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}
 my_dict['age'] = 26
    print(my_dict)

    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}
```

```
## que 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
#  (b). .Consistency and Reliability
#  (c). Performance

# (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':'age' 'Krishna', : 30, 'city' : 'New York'}
    print
     (person[
            'name'
 ])
# Output: Alice
 # Using numbers as dictionary keys
    inventory = {
 101:
'Apple',
 102:
'Banana',
 103:
'Cherry'
```

```
'Cherry'
}
      print
    (inventory[
101
])
# Output: Apple
```