1.Discuss string slicing and provide example

ANSWER-- Slicing means extracting a part from something, and slicing a string means extracting some part from a string. This extracted part is also referred to as a sub-string. In short, string slicing in Python is obtaining a substring from a larger string by slicing it based on specific criteria. String slicing allows us to obtain a specific range of characters from a string based on their respective indices. String slicing in Python allows you to extract a portion of a string by specifying a start and end index. It follows the format string[start:end], where start is the index where slicing begins (inclusive) and end is the index where it ends (exclusive).Why USED Python String Slicing-- Subsetting Data: It makes it simple to access and manipulate substrings inside a bigger text since it enables you to extract specific chunks of a string Data Extraction: It makes the process of sifting through strings to find relevant information and extracting it, which is essential for tasks like data cleaning and analysis, simpler String Manipulation: Slicing makes it easier to manipulate strings in a variety of ways, including reversing strings and extracting words or characters for additional use Efficient Access: Slicing offers direct access to a segment rather than iterating through a string character by character, which may be more effective for some jobs Substring Creation: It permits the creation of substrings for activities like producing distinctive IDs, decoding file paths, or producing outputs with specific formattingMethods of String Slicing1. Slice() Method 2. List Slicing / Array Slicing1. The slice() method returns an object containing the range of string slicing. The parameter includes a set of range of values defined as (start, stop, step).

```
In [1]:#FOR EXAMPLE OF SLICE METHOD
       string = 'Coding Ninjas'
       print(string[slice(6)])
```

```
Coding
```

```
In [3]:string = 'pwskills Mahesh'
       print(string[slice(2,10,3)])
```

```
sl
```

```
In [6]:string1 = "I am a good student"
       string1[0:4]
```

```
Out[6]:'I am'
```

2. List Slicing / Array Slicing The List Slicing method is the most common way of string slicing in Python. The parameter includes a set of range of values defined as [start, stop, step].

```
In [2]:#FOR EXAMPLE OF List Slicing / Array Slicing METHOD
       string = 'Coding Ninjas'
       print(string[7:12])
```

```
Ninja
```

```
In [5]:string = 'pwskil machin'
       print(string[1:12:2])
```

```
wklmci
```

```
In [7]:string2 = "hello world"
       string2[0:5:2]
```

```
Out[7]:'hlo'
```

2. Explain the key features of lists in Python ?

ANSWER-- Features of list -Lists can contain items of different types at the same time (including integers, floating point numbers, strings and boolean values). -Lists are mutable and dynamic; list items can be added, removed or changed after the list is defined. -Lists are ordered; newly added items will be placed at the end of the list. -Lists use zero-based indexing; every list item has an associated index, and the first item's index is 0. -Duplicated list items are allowed. -Lists can be nested within other lists indefinitely.for example

```
In [1]:grocer_list = ["milk","orange",1,2,2.2, True]
       type(grocer_list)
```

```
Out[1]:list
```

```
In [2]:pages = ["titlepage", "chap1", "chap2","conclusion","index"]
       pages[-1]
```

```
Out[2]:'index'
```

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

ANSWER-- -- Accessing Values in List There are various ways through which we can access the items present inside the list in Python. With the help of the index, we can access the elements of the list. Index starts from 0 and the index should always be an Integer. If we use an index other than integer like float, then it will result in TypeError.

```
In [7]:# Define a list
       z = [3, 7, 4, 2]
       # Access the first item of a list at index 0
       print(z[0])
```

```
3
```

```
In [8]:#modify --
       # Define a list
       z = [4, 1, 5, 4, 10, 4]
       print(z.index(4))  # The index method returns the first index at which a value occurs. In the code below, it will return 0
```

```
0
```

```
In [9]:#You can also specify where you want to start your search.
       print(z.index(4, 3))
```

```
3
```

```
In [12]:lis = ["Apple", "orange"]
         lis
```

```
Out[12]:['Apple', 'orange']
```

```
In [13]:lis.append("banana")
         lis
```

```
Out[13]:['Apple', 'orange', 'banana']
```

```
In [14]:lis[1]= "mahesh"
         lis
```

Out[14]:['Apple', 'mahesh', 'banana']
In [19]:
```
# delete
book_list = ["data str", "algorithum", "web"]
book_list.clear()
```
In [18]:
```
book_list
```
Out[18]:[]

4. Compare and Contrast tuples and list with example

Tuple comparison is the process of comparing two tuples to determine if they are equal or not. In Python, there are two types of tuple comparison operators: the equality operator (==) and the inequality operator (!=). The equality operator (==) checks whether two tuples have the same elements in the same order. If two tuples have the same elements in the same order, then they are considered equal and the equality operator returns True. If the two tuples have different elements or if their elements are in a different order, then they are considered unequal and the equality operator returns False.
In [21]:
```
tuple1 = (1, 2, 3)
tuple2 = (1, 2, 3)

if tuple1 == tuple2:
    print("The two tuples are equal")
else:
    print("The two tuples are not equal")
```
The two tuples are equal
In [22]:
```
# In this example, tuple1 and tuple2 have different elements, so when we compare them using the inequality operator,
# we get True as output.
tuple1 = (1, 2, 3)
tuple2 = (4, 5)

if tuple1 != tuple2:
    print("The two tuples are not equal")
else:
    print("The two tuples are equal")
```
The two tuples are not equal
#comparison of LIST AND TUPLE LIST Lists are mutable The implication of iterations is Time-consuming The list is better for performing operations, such as insertion and deletion. Lists consume more memory Lists have several built-in methods Unexpected changes and errors are more likely to occur
In [23]:
```
#list for example
# Creating a List with
# the use of Numbers
# code to test that tuples are mutable
List = [1, 2, 4, 4, 3, 3, 3, 6, 5]
print("Original list ", List)

List[3] = 77
print("Example to show mutability ", List)
```
Original list [1, 2, 4, 4, 3, 3, 3, 6, 5]
Example to show mutability [1, 2, 4, 77, 3, 3, 3, 6, 5]
TUPLE Tuples are immutable The implication of iterations is comparatively Faster A Tuple data type is appropriate for accessing the elements Tuple consumes less memory as compared to the list Tuple does not have many built-in methods. Because tuples don't change they are far less error-prone.
In [24]:
```
# code to test that tuples are immutable

tuple1 = (0, 1, 2, 3)
tuple1[0] = 4
print(tuple1)
```
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[24], line 4
      1 # code to test that tuples are immutable
      3 tuple1 = (0, 1, 2, 3)
----> 4 tuple1[0] = 4
      5 print(tuple1)

TypeError: 'tuple' object does not support item assignment
```

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

Answer-- Set-- A set is a collection of unique data, meaning that elements within a set cannot be duplicated. A Set in Python programming is an unordered collection data type that is iterable, mutable and has no duplicate elements. features of Set- - Sets are unordered. - Set elements are unique. Duplicate elements are not allowed. - A set itself may be modified, but the elements contained in the set must be of an immutable type.
In [1]:
```
#for example
s = {1}
type(s)
```
Out[1]:set
In [2]:
```
#use case
list1 = [1, 2, 3, "brijal", "brijal", "apple", "apple"]
s = set(list1)
s
```
Out[2]:{1, 2, 3, 'apple', 'brijal'}
In [3]:
```
s = {1, 1, 2, 2, 3, 4}
#set is mutable
```

```
s.add(100) #not compalasary that it will add t the last of set
s
```

Out[3]:{1, 2, 3, 4, 100}

```
In [4]:s = {1, 1, 2, 2, 3, 4, 100}
s.remove(3) #removes a specific element
s
```

Out[4]:{1, 2, 4, 100}

```
In [5]:#set operations
#union -- combines elements from two sets excluding duplicates
#for example
s1 = {"hiking", "reading", "coding"}
s2 = {"coding", "photo", "traveling"}
#union -combine all the elements
s1 | s2
```

Out[5]:{'coding', 'hiking', 'photo', 'reading', 'traveling'}

```
In [6]:#intersection-- only common elements between sets
#for example
s1 = {"hiking", "reading", "coding"}
s2 = {"coding", "photo", "traveling"}
s1 & s2
```

Out[6]:{'coding'}

```
In [7]:#difference -- return the elements tat is present in first set and not in second
s1-s2
```

Out[7]:{'hiking', 'reading'}

```
In [8]:#symmetric difference --returns elements that are present in either of sets but not in both
s1 ^ s2
```

Out[8]:{'hiking', 'photo', 'reading', 'traveling'}

```
In [9]:#frozen sets -- immutable version of set, cannot be added or removed any new element
my_fs = frozenset([1, 2, 2, 2, 3, 3, 4, 5])
my_fs
```

Out[9]:frozenset({1, 2, 3, 4, 5})

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

Tuples-- Tuples are immutable. Hence, they are primarily used to store data that doesn't change frequently. Any operation can store data in a tuple when you don't want it to change. Tuples are great to use if you want the data in your collection to be read-only, never to change, and always remain the same and constant.Use Cases of Tuples: --Storing Fixed Data: Tuples are often used to store a collection of related items that should not be changed, such as coordinates, dates, or configuration settings.

```
In [12]:#for example
# Coordinates of a point
point = (4, 5)

# Accessing elements
x = point[0]
y = point[1]

print(f"X: {x}, Y: {y}")
```

X: 4, Y: 5

-Returning Multiple Values from a Function: A function can return multiple values as a tuple, allowing for easy unpacking.

```
In [13]:#for example
def get_user_info():
    name = "Alice"
    age = 30
    country = "USA"
    return name, age, country  # Returning a tuple

# Unpacking the tuple
name, age, country = get_user_info()

print(f"Name: {name}, Age: {age}, Country: {country}")
```

Name: Alice, Age: 30, Country: USA

-Using Tuples as Keys in a Dictionary: Since tuples are immutable, they can be used as keys in dictionaries (whereas lists cannot).

```
In [14]:#for example
# A dictionary with tuple keys
locations = {
    (40.7128, -74.0060): "New York",
    (34.0522, -118.2437): "Los Angeles",
    (51.5074, -0.1278): "London"
}

# Accessing data using a tuple key
city = locations[(40.7128, -74.0060)]
print(f"The city is: {city}")
```

The city is: New York

Data Integrity: When you want to ensure that certain data is protected from being accidentally altered, storing it in a tuple can help maintain data integrity.

```
In [16]:#for example
```

```
# Immutable configuration settings
config = ("user123", "localhost", 8080)
config
```

Out[16]:('user123', 'localhost', 8080)

SET -- A set in Python is an unordered collection of unique elements. Sets are mutable, meaning you can add or remove elements, but they cannot contain duplicate items. They are useful in scenarios where you need to ensure that all elements in a collection are unique, or when you need to perform operations like union, intersection, and difference between collections.Use Cases of Sets: -- Eliminating Duplicates: Sets are ideal for removing duplicate elements from a list or any other iterable.

In [17]:
```
#for example
# A list with duplicate elements
numbers = [1, 2, 3, 4, 4, 5, 6, 7, 7, 8]

# Converting the list to a set to remove duplicates
unique_numbers = set(numbers)

print(f"Unique numbers: {unique_numbers}")
```

Unique numbers: {1, 2, 3, 4, 5, 6, 7, 8}

--Membership Testing: Sets provide an efficient way to test if an element exists within a collection, with average time complexity of O(1).

In [18]:
```
#for example
# A set of users
users = {"Alice", "Bob", "Charlie"}

# Checking if a user exists in the set
if "Alice" in users:
    print("Alice is in the set.")
else:
    print("Alice is not in the set.")
```

Alice is in the set.

-- Set Operations: Sets support mathematical operations like union, intersection, difference, and symmetric difference, which can be very useful in certain algorithms.

In [19]:
```
#for example
# Two sets of letters
set_a = {"a", "b", "c", "d"}
set_b = {"c", "d", "e", "f"}

# Union (elements in either set)
union = set_a | set_b
print(f"Union: {union}")

# Intersection (elements in both sets)
intersection = set_a & set_b
print(f"Intersection: {intersection}")

# Difference (elements in set_a but not in set_b)
difference = set_a - set_b
print(f"Difference: {difference}")

# Symmetric Difference (elements in either set, but not both)
symmetric_difference = set_a ^ set_b
print(f"Symmetric Difference: {symmetric_difference}")
```

Union: {'e', 'd', 'c', 'b', 'f', 'a'}
Intersection: {'d', 'c'}
Difference: {'a', 'b'}
Symmetric Difference: {'e', 'b', 'f', 'a'}

--Maintaining a Collection of Unique Items: When you need to store a collection of unique items without any specific order, a set is a natural choice.

In [20]:
```
#for example
# A set to store unique email addresses
emails = set()

# Adding emails to the set
emails.add("user1@example.com")
emails.add("user2@example.com")
emails.add("user1@example.com")  # This will not be added again

print(f"Unique emails: {emails}")
```

Unique emails: {'user2@example.com', 'user1@example.com'}

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

ANSWER-- Disctonary-- disctonary is a data stcture that stores data as key value pai keys are unique and imutable1.Adding Items to a Dictionary You can add a new key-value pair to a dictionary by simply assigning a value to a new key.

In [40]:
```
#for example
# Creating an empty dictionary
my_dict = {}

# Adding a key-value pair
my_dict["name"] = "Akshay"
```

```
my_dict["age"] = 30

print(my_dict)
```

```
{'name': 'Akshay', 'age': 30}
```

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

In [31]:
```
#for example
my_dict["name"] = "Alice"
my_dict["age"] = 31
my_dict["city"] = "Los Angeles"
my_dict["profession"] = "Engineer"
print(my_dict)
```

```
{'name': 'Alice', 'age': 31, 'city': 'Los Angeles', 'profession': 'Engineer'}
```

In [33]:
```
my_dict['name'] = "pwskills"
print(my_dict)
```

```
{'name': 'pwskills', 'age': 31, 'city': 'Los Angeles', 'profession': 'Engineer'}
```

Deleting Items from a Dictionary

In [34]:
```
#for example
# Deleting a key-value pair
del my_dict["city"]

print(my_dict)
```

```
{'name': 'pwskills', 'age': 31, 'profession': 'Engineer'}
```

In [35]:
```
# Deleting a key-value pair and getting its value #### another way ####
age = my_dict.pop("age")
print(age)
print(my_dict)
```

```
31
{'name': 'pwskills', 'profession': 'Engineer'}
```

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

Importance of Immutable Dictionary Keys In Python, dictionary keys must be immutable. This means they cannot be changed after they are created. While this might seem restrictive, it's essential for the efficient operation of dictionaries. Why Immutable Keys? Hashing Efficiency: Dictionaries use hashing to store and retrieve values efficiently. Hashing involves converting a key into a numerical value (hash value) to determine its location within the dictionary. Immutable keys guarantee that their hash value remains constant, ensuring consistent and efficient lookups. Data Integrity: If keys were mutable, changing a key's value could inadvertently alter its hash value. This could lead to the key being stored in a different location, making it impossible to retrieve. Immutable keys prevent this kind of data corruption. Consistency: Immutable keys ensure that the dictionary's structure remains consistent. This is crucial for algorithms and operations that rely on the predictable behavior of dictionaries.

In [37]:
```
restaurant_menu = {
    'dish1': {'name': 'pashta', 'price': 450, 'description': "good"},
    'dish2': {'name': 'chiken n', 'price': 550, 'description': "unhealthy"},
    'dish3': {'name': 'pizza ', 'price': 950, 'description': "bad"}
}
type(restaurant_menu)
```

```
Out[37]:dict
```

In [38]:
```
restaurant_menu["dish1"]
```

```
Out[38]:{'name': 'pashta', 'price': 450, 'description': 'good'}
```

In [39]:
```
# Example of immutable keys in a Python dictionary

my_dict = {
    "name": "Alice",
    123: "Bob",
    (1, 2): "Charlie",
    True: "David"
}

# Accessing values using immutable keys
print(my_dict["name"])
print(my_dict[123])
print(my_dict[(1, 2)])
print(my_dict[True])
```

```
Alice
Bob
Charlie
David
```

In [ ]: