```
#Question 1 Discuss string slicing and provide examples

# Strict Silicing examples

a = "Anubella"
a[0]
```

```
'A'
```

```
a[3]
```

```
'b'
```

```
a[6]
```

```
'l'
```

```
a[9]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[9], line 1
----> 1 a[9]

IndexError: string index out of range
```

```
string1 = "Bella soul is a pure soul"
string1
```

```
'Bella soul is a pure soul'
```

```
string1[0:4]
```

```
'Bell'
```

```
string1[7:12]
```

```
'oul i'
```

```
string2= "Bella is a good human being"
string2
```

```
'Bella is a good human being'
```

```
string2[-4]
```

```
'e'
```

```
string2[:-8]
```

```
'Bella is a good hum'
```

```
string2[-4:]
```

```
'eing'
```

```
#question 2 explain the key features of lists in Python?

#features of python

# Description : ordered , mutuable collections of eements. Think of
shopping list or task list. list can hold items of various data
types(numbers, strings, even other list0

# Operations : you can add , remove or modify elements with in a list
using indexing & silicing . list are versatile for storing & managing
collections that might change .

#Examples: python grocery_list = [" bread","mirch", "namkeen", "salt",
"Turmeric"]
        # mixed data type grocery_list.append["coffee"]
      # Adding an item del grocery_list[1]
        # Removing the second item

grocery_list =[" bread","mirch", "namkeen", "salt", "Turmeric"] #mixed
data types
grocery_list.append["coffee"] # Adding an item
del grocery_list [1] # Removing grocery_list

# Analogy : Imagine a shopping cart where you can add new items ,
remove unwanted or change quantities .list offer this flexibility in
data storage

# Question 3 :  Describe how to access, modify, and delete elements in
a list with examples

# Accessing , modifying , and deleting elements in a list are
fundamental operations in python. here an detailed explanation with an
example.

# 1 Accessing element : you can acess elements in a list using
indexing and slicing.

    #Indexing : Access a specific elemnent by its position in a list
(index starts at 0)


list1 = ["tomato", "potato", "onion","brinjal","carrot"]
list1
```

```
['tomato', 'potato', 'onion', 'brinjal', 'carrot']
```

```
first_element= list1[0]
print(first_element)
```

```
tomato
```

```
third_element = list1[2]
print(third_element)
```

```
onion
```

```
#silicing : Access a subset of elements using silicing notation list [start:end]

# Accessing a slice of elements

subset= list1[1:3]
print(subset)
```

```
['potato', 'onion']
```

```
subset= list1[1:]  # accessing from a certain index to the end
print(subset)
```

```
['potato', 'onion', 'brinjal', 'carrot']
```

```
#2 modifying elements: you can modify elements in a list by directly assigning new vlues to specific indices or slices.

list1 = ["tomato", "potato", "onion","brinjal","carrot"]

#modifying third elemet
list1= ["tomato", "potato", "onion","brinjal","carrot"]
list1.append ("bittergod")
print(list1)
```

```
['tomato', 'potato', 'onion', 'brinjal', 'carrot', 'bittergod']
```

```
list1= ["tomato", "potato", "onion","brinjal","carrot"]
list1[2:4]
```

```
['onion', 'brinjal']
```

```
# 3 Deleting elemets : you can delete elements form a list using delete statement

list1= ["tomato", "potato", "onion","brinjal","carrot"]
del list1[3]
print(list1)
```

```
['tomato', 'potato', 'onion', 'carrot']
```

```
# pop statement

list1= ["tomato", "potato", "onion","brinjal","carrot"]
list1.pop()
```

```
'carrot'
```

```
list1
```

```
['tomato', 'potato', 'onion', 'brinjal']
```

```
# Remove statement

list1= ["tomato", "potato", "onion","brinjal","carrot"]
list1.remove("onion")
print(list1)
```

```
['tomato', 'potato', 'brinjal', 'carrot']
```

```
list1= ["tomato", "potato", "onion","brinjal","carrot","onion"]
list1.remove("onion")
print(list1)
```

```
['tomato', 'potato', 'brinjal', 'carrot', 'onion']
```

```
 # Question 4 Compare and contrast tuples and lists with examples

# Tuples and lists are both sequence data types in python, but they
have distinct characteristics that make them suitable for  different
purposes. Here's a comparison and contrast between tuples and lists:

# similarities :
          # 1 Sequence Types: both tuples and list are sequences ,
meaning they maintain the order of elements.
          # 2  Indexing and Slicing: elements in both tuples and list
can be accessed using indexing and slicing.
          # 3  Iteration : you can iterate over both tuples and lists
using loops or comprehensions.

        # Difference

        #1. Mutability:

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

            #Tuples : are immutable , meaning once they are created,
their elements cannot be changed or modified.


            #example


list1= [1,2,3,4,5] # modifying te eement
list1[3] = 23
print(list1)
```

```
[1, 2, 3, 23, 5]
```

```
tuple=(1,2,3,4) # this will raise error tuple are immutable
tuple[2]= 23
print(tuple)
```

```
---------------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
last)
Cell In[73], line 2
      1 tuple=(1,2,3,4)
----> 2 tuple[2]= 23
      3 print(tuple)

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

```
# 2 syntax :

 # lits: lists are defined using square brackets[].

 # Tuple: Tuple are defined using parentheses{}, although they can
also be defined witout parentheses for clarity in some texts.

list= [1,2,3]
tuple= (1,2,3) #busing parentheses

another tuple =4,5,6  # tuple can be defined without parenthesis
```

```
  Cell In[75], line 1
    another tuple
            ^
SyntaxError: invalid syntax
```

```
# 3 Usage


  #1 lists: lists are typically used for collections of homogenous
items where the order and mutability of elemets are important, such as
mamaging a collection of user inputs or a dynamic list of data.

  #2 Tuple:  are used  when the elements are related and often
represent a single entity. They are also useful for returning multiple
values from a function and for immutability guarantees.



#4 Performance :

  # lists : due to their mutability, lists may consume more memory and
have sligtly slower performance compared to tuples for operations like
indexing and iteration.
```

```
  # tuple : are more memory efficent and have faster access times for reading data, especially when the size is fixed  and known at creation time .

list= [1,2,3,4,5]
list[2]= 34
print(list)
```

```
[1, 2, 34, 4, 5]
```

```
tuple= (1,2,3,4,5)
tuple(2)=34
print(tuple)
```

```
  Cell In[5], line 2
    tuple(2)=34
    ^
SyntaxError: cannot assign to function call here. Maybe you meant '==' instead of '='?
```

```
#5  Question  Describe the key features of sets and provide examples of their use

# In python you can work with sets using built in data structures and operations provided by the language. let's describe the key creatures of sets and provide examles of their use in python:



 #1 creating set : sets in python are defined using curly braces {} or the set () function.



a = {32,14,15,18,98,56}
b = {56,67,98,34,23,66}

a.add(58)
print(a)
```

```
{32, 98, 18, 56, 58, 14, 15}
```

```
b.remove(98)
print(b)
```

```
{67, 34, 66, 23, 56}
```

```
# 3 set operation: Python supports operations like union(\ or .union ()), intersection (& or .intersection ()), difference (- or . symmetric_difference())
```

```
a = {1,2,3,4,5}
b = {4,5,6,7,8}
union_set = a | b
print(union_set)
```

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

```
a = {1,2,3,4,5}
b = {4,5,6,7,8}
intersection_set = a & b
print(intersection_set)
```

```
{4, 5}
```

```
a = {1,2,3,4,5}
b = {4,5,6,7,8}
difference_set = a - b
print(difference_set)
```

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

```
a = {1,2,3,4,5}
b = {4,5,6,7,8}
symmetric_difference_set = a ^ b
print(symmetric_difference_set)
```

```
{1, 2, 3, 6, 7, 8}
```

```
# 4 subset and superset checking: you can check if one set is a subset
or superset of another using <=(subset) or  >= (superset) operators.
```

```
a = {1,2,3,4,5}
b = {4,5,6,7,8}
subset = a <= b
print(subset)
```

```
False
```

```
a = {1,2,3,4,5}
b = {4,5,6,7,8}
superset = b >= a
print(superset)
```

```
False
```

```
# 5 set comprehension : allow you to create sets based on conditions
in a compact and readable way.
```

```
numbers = [1, 2, 3, 4, 5]
[num**2 for num in numbers]
```

```
[1, 4, 9, 16, 25]
```

```
# 6 set  membership and iteration : you can check if an element is in
a set using the in keyword, and iterate over elements  of a set using
a for loop.

a = [1, 2, 3, 4, 5]
3 in set(a)
print (a)
```

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

```
#6  Question Discuss the use cases of tuples and sets in python
programming.

# Tuples and sets are both fundamental data structures in python , but
they serve different purposes and have distinct characteristics that
make them suitable for different use cases.

# tuple:


    #1 Immutable sequence :  # tuples are immutable , meaning once
they are created, their elements cannot be changed. This immutability
makes tuples usefl for representing fixed collections of items that
sould not be modified.

                            # use case : storing data that should
remain constant throughout the proram's execution , such as
coordinates , configuration settings , or database records.


    # 2 Function return values:

                     # Tuples are often used to return multiple valuues
from a  function in a single return statement. This is convenient when
you need to pass back more than one piece of data.

                   # use case : returning multile values from a
function without  using more  complex data structure.


     # 3 Dictionary keys :

                 # since tuples are immutable and  hashable , they can
be used as keys in dictionaries.

             # use case : creating dictionaries where keys need to be
fixed combinations of values ( eg,. cordinates as key for a grid).


       #4 Performance:
```

```
                    # Tuples are generally faster than lists because
they are immutable. This can be advantageous in situations where you
need to iterate over data or access elements quickly .

                   # use case : storing data that needs to be accessed
frequently or iterated over rapidy.

       # 5 unpacking:

                  # Tuples can be unpacked into multiple variables
easily, which makes them handy when you need to assign values to
multiple variabes in a single statement .


                # use case : Assigning values returned from a function
to separate  variable in one step.


        # set :


         # 1 uniqueness and set Operations :

                   # sets in python are collections of unique elements.
They are useful for tasks that involve checking membership ,
eliminating duplicates, and performing set operations ( union,
intersection , difference , etc,)

                  #  use case : removing duplicates from a list ,
checking for existence of items , or combining data while  ensuring
uniqueness.



         # 2 mathematical set operations :

                # sets support operations like union , intersection ,
difference and symmetric difference , which are essential in various
algorithms and data processing tasks.

               # use case : Finding common elements between two lists ,
determining uniqu items in a dataset , or combining data from
different sources while handling duplicates.


        #3 membership testing and fast lookup:

             # checking if an element is in a set is very fast
```

```
(average 0 (1) time complexity), which makes sets ideal for scenarios were quick membership

        #use case : maintaining a collection of unique items for quick lookup or verificaton.


        # 4 mutable and dynamic :

         # unlike tuples , sets  are mutable , meaning you can add or remove elements after teir creation . This flexibility allows sets to be dynamicaally updated based on program reqirements .


         # use case : Tracking canging data over time , suchh as unique user ids ogged into a system .

        # 5 set comprehensions :

              # similar to list comprehensions, pyton supports set comprehension for creating sets in a concise and readable manner.


            # use case : generating sets based on conditions or transformation from other iterables.

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

 # in python , dictionaries are versatile data structure that store key value pairs. Adding , modifying and deleting items in a dictionary are common operations that are straight forward to perform using built in methods and syntax . Here's how you can do each of these operations :



 # 1 . Adding items to a dictionary : to add a new item (key - value pair ) to a dictonary , you can simply assign a value to a new key , or use the update() method to add multiple items at once .


    # using assignment :

# creating an empty dictionary
    my_dict = {}

  Cell In[19], line 2
    my_dict = {}
    ^
```

```
IndentationError: unexpected indent
```

```
# adding a single item
d = {"name":"bella" , "age":25 , "email id":"bella@gmail.com"}
```

```
d
```

```
{'name': 'bella', 'age': 25, 'email id': 'bella@gmail.com'}
```

```
d ["name"]
```

```
'bella'
```

```
d ["age"]
```

```
25
```

```
d ["email id"]
```

```
'bella@gmail.com'
```

```
# adding multiple items at once
d1 = {"address": "143A modi colony","contact no" : 7654398777}
d.update(d1)
```

```
d1
```

```
{'address': '143A modi colony', 'contact no': 7654398777}
```

```
# 2 modifying items in a dictionary : to modify an existing item in a
dictionary, simply reassign a new value to existing key.
```

```
example :
```

```
d = {"name"- "bella", "conatct no" - 8764289977 , "age"- 43,"city"-
"new york"}
```

```
# modifying age
d["age"] = 67
d
```

```
{'name': 'bella',
 'age': 67,
 'email id': 'bella@gmail.com',
 'address': '143A modi colony',
 'contact no': 7654398777,
 'city': 'ottawa'}
```

```
d["city"] = "ottawa"
d
```

```
{'name': 'bella',
 'age': 67,
 'email id': 'bella@gmail.com',
 'address': '143A modi colony',
 'contact no': 7654398777,
 'city': 'ottawa'}
```

```
# deleting items form a dictionary:
#using
```

```
d = {"name": "Alice", "age":25, "city": "boston"}
d
```

```
{'name': 'Alice', 'age': 25, 'city': 'boston'}
```

```
# del age
d = {"name": "Alice", "age":25, "city": "boston"}
d.pop("age")
print(d)
```

```
{'name': 'Alice', 'city': 'boston'}
```

```
# using pop
d = {"name": "Alice", "age":25, "city": "boston"}
d.pop("city")
print(d)
```

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

```
d
```

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

```
# 8 Question  Discuss the importance of dictionary keys being
immutable and provide examples

 # importance of immutable dictionary keys :


    #1 Hashing and lookup efficiency :

        # dictionary keys must be immutable so that they can be
reliably hashed. hashing is a technique used to convert a key into a
unique numerical value , which allows for efficient storage and
retrieval of key - value pairs.

        # immutable keys ensure that their harsh value remains
constant throughout their lifetime in the dictionary . This
consistency is essential for quick lookups, as python can directly
compute the hash locate the corresponding bucket in constant time on
average (0(1) time complexity.
```

```
    #2 preventing unintended modifications :

      # if dictionary keys were mutable , and if a key's value were
to change after it had been used as a dictionary key, it could lead to
unexpected behavior. for example , changing the value of a mutable
object (such as a list or another dictionary ) used as a key could
potentialy change its hash value or affect te overall structure of the
dictionary.

      # immutabe keys  prevent accidental or intentional
modifications that could compromise the integrity of dictionary
operations.



   # 3 Ensuring Cosistency in Hash tables :

      # dictonries in python are implemented as hash tables , where
each key - value pair is stored based on the key's hash value.
maintaining the consistency required for hash table oerations like
insertion , retrieval and deletion.


       # this consistency is fundamental to the reliable performance
of dictionaries, especially in scenarios where dictionaries are eavily
utlized for data storage , caching or lookup operations .


       examples:




#1 strin as keys : string are immutable in pyton whic makes them
excellent candidates for dicionary keys. once a string is used as a
key in a dictionary, its hash value remains unchanged.
d = {"name":"bella", "age":24}
d
```

```
{'name': 'bella', 'age': 24}
```

```
# 2 tuple as key :tuple are immutable collections and thus can be used
as dictionary keys. The immutability of tuples ensures tat their has
values are consistent.


cordinates = {(0,0):'origin', (1,1): 'diagonal'}
print(cordinates)
```

```
{(0, 0): 'origin', (1, 1): 'diagonal'}

# 3 numbers and immutable objects

{'name':'Alice', 'age' : 30}
{(0,0)
```