Q1. Can you create a programme or function that employs both positive and negative indexing? Is there any repercussion if you do so?

Ans: Yes, we can create a program or function that employs both positive and negative indexing in Python. In fact, many programs use both types of indexing to access different parts of a string.

Here's an example function that uses both positive and negative indexing to extract a substring from a string:

def get\_substring(s, start, end):

return s[start:end]

s = "hello world"

print(extract\_substring(s, 1, 5)) # Output: "ello"

print(extract\_substring(s, -5, -1)) # Output: "worl"

There are no major repercussions to using both positive and negative indexing in a program or function. However, it's important to be consistent in your use of indexing throughout the code, to avoid confusion and errors.

Q2. What is the most effective way of starting with 1,000 elements in a Python list? Assume that all elements should be set to the same value.

Ans: If we want to create a Python list with 1,000 elements that are all set to the same value, the most effective way is to use the \* operator to create a list of the desired length, and then use a list comprehension to set each element to the desired value.

Here's an example:

n = 1000

my\_list = [0] \* n

In this example, the \* operator is used to create a list of n zeros. we can replace 0 with any other value that we want to initialize the list with.

If we want to set the elements to a value other than 0, we can use a list comprehension like this:

n = 1000

my\_value = "hello"

my\_list = [my\_value for i in range(n)]

In this example, the list comprehension [my\_value for i in range(n)] creates a list of n copies of my\_value.

Q3. How do you slice a list to get any other part while missing the rest? (For example, suppose you want to make a new list with the elements first, third, fifth, seventh, and so on.)

Ans: l = [1,2,3,4,5,6,7,8]

print(l[::2])

**O/P:**

[1, 3, 5, 7]

Q4. Explain the distinctions between indexing and slicing.

Ans: Here are some key differences between indexing and slicing:

a. Indexing selects a single element, while slicing selects a range of elements.

b. Indexing uses a single index, while slicing uses two indices separated by a colon.

c. Indexing returns a single element, while slicing returns a new list or string containing the selected elements.

d. Indexing raises an IndexError if the index is out of bounds, while slicing does not (it returns an empty list or string if the slice is out of bounds).

Q5. What happens if one of the slicing expression's indexes is out of range?

Ans: If one of the slicing expression's indexes is out of range, Python will not raise an error, but will simply adjust the index to the nearest valid value.

If the starting index is out of range, Python will set it to 0. For example, if you try to slice a list with the expression my\_list[10:20] but the list only has 8 elements, Python will adjust the starting index to 0 and slice the entire list.

If the ending index is out of range, Python will set it to the length of the list or string. For example, if you try to slice a list with the expression my\_list[2:10] but the list only has 6 elements, Python will adjust the ending index to 6 and slice from the third element to the end of the list.

If both indexes are out of range, Python will slice an empty list or string. For example, if you try to slice a list with the expression my\_list[10:20] but the list only has 8 elements, Python will adjust both indexes and return an empty list.

It's worth noting that in all of these cases, Python will not raise an error, so you need to be careful when slicing to ensure that you get the expected results.

Q6. If you pass a list to a function, and if you want the function to be able to change the values of the list—so that the list is different after the function returns—what action should you avoid?

Ans: If we create a new list with the same name as the original list inside the function, we're replacing the reference to the original list with a reference to a new list object. This means that any changes that we make to the new list will not be reflected in the original list outside the function.

Ex:

def replace\_list(my\_list):

my\_list = [1, 2, 3, 4, 5]

my\_list = [10, 20, 30, 40, 50]

replace\_list(my\_list)

print(my\_list) # Output: [10, 20, 30, 40, 50]

In this example, the replace\_list function creates a new list with the same name as the original list passed to the function. This replaces the reference to the original list with a reference to the new list, so any changes made to the new list inside the function are not reflected in the original list outside the function.

To avoid this, we should modify the original list directly inside the function. For example:

def replace\_list(my\_list):

my\_list[0] = 1

my\_list[1] = 2

my\_list[2] = 3

my\_list[3] = 4

my\_list[4] = 5

my\_list = [10, 20, 30, 40, 50]

replace\_list(my\_list)

print(my\_list) # Output: [1, 2, 3, 4, 5]

In this example, the replace\_list function modifies the original list directly by assigning new values to its elements. This changes the original list in place, so the changes are visible outside the function.

Q7. What is the concept of an unbalanced matrix?

Ans: An unbalanced matrix is a matrix or a two-dimensional array where the number of elements in each row is not the same. In other words, the matrix has different lengths for different rows. This is in contrast to a balanced matrix, where each row has the same number of elements.

Ex:

[[1, 2, 3],

[4, 5],

[6, 7, 8, 9]]

Unbalanced matrices can be more challenging to work with than balanced matrices because we cannot assume that all rows have the same number of elements. This can make certain operations, such as matrix multiplication or transposition, more complicated or even impossible.

To work with an unbalanced matrix, we may need to use techniques such as padding or truncation to ensure that all rows have the same number of elements. Alternatively, we may need to use specialized data structures or libraries that can handle unbalanced matrices efficiently.

Q8. Why is it necessary to use either list comprehension or a loop to create arbitrarily large matrices?

Ans: In Python, a matrix can be represented as a list of lists, where each inner list represents a row of the matrix. To create a matrix with an arbitrary number of rows and columns, we need to create a list of the appropriate length containing inner lists of the appropriate length.

we can create a matrix using a loop or list comprehension because they allow we to generate the necessary list structure dynamically. For example, a loop can be used to generate the rows of the matrix one at a time, while list comprehension can be used to create a list of lists in a single statement.

If we try to create a matrix using only literal syntax, such as [[0]\*n]\*m, we may encounter unexpected behavior. This is because the inner lists in the resulting matrix are actually references to the same list object, so modifying one row will affect all other rows. This can lead to subtle bugs and errors in wer code.

Therefore, to create a matrix with arbitrary size and content, we need to use either a loop or list comprehension to create each row and assemble them into a list of lists representing the matrix.