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

Ans1

Yes, it is possible to create a program or function that employs both positive and negative indexing. In many programming languages, including Python, both positive and negative indexing are supported for sequences such as strings, lists, and tuples.

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

my\_list[0]

my\_list[-1]

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.

Ans2

The most effective way to create a Python list with 1,000 elements that are all set to the same value is to use the list multiplication operator (\*) along with a list containing the value to be repeated.

my\_list = [0] \* 1000

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.)

Ans3

To slice a list and get every other element, you can use the slice notation with a step size of 2.

my\_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

new\_list = my\_list[::2]

Q4. Explain the distinctions between indexing and slicing.

Ans4

Indexing and slicing are two related but distinct concepts when working with sequences like lists, tuples, and strings.

Indexing refers to accessing a single element from a sequence by its position (index) in the sequence. indexing starts at 0 for the first element and goes up to the length of the sequence minus one for the last element.

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

third\_element = my\_list[2]

Slicing, on the other hand, refers to extracting a subsequence of elements from a sequence by specifying a range of indices.

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

first\_three\_elements = my\_list[0:3]

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

Ans5

If one of the slicing expression's indexes is out of range, then will not raise an error. Instead, it will simply return the slice of the sequence up to the index that is in range. If the start index is out of range, the slice will start from the beginning of the sequence. If the stop index is out of range, the slice will go up to the end of the sequence.

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

slice = my\_list[1:10]

slice = my\_list[1:]

[]

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?

Ans6

a function to be able to change the values of a list so that the list is different after the function returns, avoid creating a new list with the same name as the input list inside the function. This is because the new list will shadow (i.e., take precedence over) the input list, and any changes made to the new list will not affect the input list outside of the function.

Instead, you should modify the elements of the input list in-place. This can be done by accessing the elements of the list using their indices and assigning new values to them.

def add\_one\_to\_list(my\_list):

for i in range(len(my\_list)):

my\_list[i] += 1

Q7. What is the concept of an unbalanced matrix?

Ans7

An unbalanced matrix is a matrix where the number of elements in each row or column is not the same. In other words, the matrix has different row or column sizes. This is in contrast to a balanced matrix, where each row and column has the same number of elements.

[[1, 2, 3],

[4, 5],

[6, 7, 8, 9]]

Unbalanced matrices can arise in certain situations where the data being represented does not have a uniform structure. However, working with unbalanced matrices can be more challenging than working with balanced matrices, as the varying sizes of the rows or columns can make certain operations more difficult. Some algorithms and functions may not work with unbalanced matrices, or may require special handling to work correctly.

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

Ans8

It is necessary to use either list comprehension or a loop to create arbitrarily large matrices because a matrix is a two-dimensional data structure that contains rows and columns of elements. Unlike one-dimensional data structures like lists or tuples, creating a two-dimensional data structure requires nested iterations or comprehensions to specify the values of each element.

n\_rows = 3

n\_cols = 3

matrix = []

for i in range(n\_rows):

row = []

for j in range(n\_cols):

value = i \* j

row.append(value)

matrix.append(row)