**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, it is possible to create a program or function that employs both positive and negative indexing in Python.

Here is an example function that takes a string and returns the first and last characters of the string using both positive and negative indexing:

def first\_and\_last\_chars(s):

first\_char = s[0]

last\_char = s[-1]

return first\_char, last\_char

**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- The most effective way to create a Python list with 1,000 elements set to the same value is to use the list multiplication operator \* along with a list containing the desired value. Using the ‘\*’ operator is more efficient than using a loop to initialize the list, as it avoids the overhead of creating and managing an index variable and checking a loop condition for each element.

**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-To slice a list and get every other element, you can use the slice notation in Python. The syntax for slicing a list is ‘list[start:end:step]’, where ‘start’ is the starting index of the slice, ‘end’ is the ending index (exclusive), and ‘step’ is the step size between elements. To get every other element, you can set ‘step’ to 2.

To get the elements first, third, fifth, seventh, and so on, you can set the start index to 0, and the step index to 2. Here's an example,

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

new\_list = my\_list[::2] # get every other element

print(new\_list) #output: [1, 3, 5, 7, 9]

**Q4. Explain the distinctions between indexing and slicing.**

Ans- Indexing and slicing are two related operations in Python that allow you to access specific elements or sublists within a list.

Indexing refers to the process of accessing a specific element in a list by its position, or index. In Python, list indices are zero-based, which means that the first element of the list has an index of 0, the second element has an index of 1, and so on. To index an element in a list, you can use square brackets ([]) and specify the index of the element you want to access.

Slicing, on the other hand, refers to the process of accessing a sublist, or a portion of a list, by specifying a range of indices. The slice notation in Python is of the form ‘list[start:end:step]’, where ‘start’ is the index of the first element of the slice, ‘end’ is the index of the first element that is not included in the slice, and ‘step’ is the increment between indices. Slicing allows you to extract a portion of a list without modifying the original list.

**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, i.e., it's greater than or equal to the length of the list, or less than the negative length of the list, Python will raise an ‘IndexError’ with a message that indicates the index is out of range.

**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 you want a function to be able to change the values of a list passed as an argument, you should avoid reassigning the list to a new object or creating a new list with the same variable name within the function. Instead, you should modify the elements of the existing list directly.

In Python, lists are mutable, which means that you can change their elements without creating a new list object. This makes it possible to modify a list passed as an argument to a function and have those changes persist outside of the function.

Here's an example of how to modify a list passed as an argument within a function:

def modify\_list(my\_list):

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

my\_list[i] += 2

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

modify\_list(my\_list)

print(my\_list) # output: [3,4,5,6]

**Q7. What is the concept of an unbalanced matrix?**

Ans- In linear algebra, an unbalanced matrix is a matrix where the number of rows and columns are not equal. In other words, the matrix is not square.

Unbalanced matrices can cause problems when we try to perform certain operations, such as matrix addition or multiplication, that require the matrices to have the same dimensions. In the case of matrix addition, for example, we can only add two matrices if they have the same dimensions (i.e., the same number of rows and columns).

So to avoid issues with unbalanced matrices, we may need to reshape or pad the matrices to ensure they have the same dimensions before performing operations on them. Alternatively, we may need to modify our approach or use a different algorithm altogether.

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

Ans- It is necessary to use either list comprehension or a loop to create arbitrarily large matrices because these methods allow us to create a large number of elements efficiently without having to write out each individual element manually.