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

**Answer:**  
Yes, you can definitely use both **positive** and **negative indexing** in a program. In fact, Python allows you to access elements from both the beginning (positive index) and the end (negative index) of a list simultaneously, providing a lot of flexibility when working with data.

Here's an example that demonstrates both indexing methods:

def mixed\_indexing(lst):

# Positive indexing

print("First element:", lst[0]) # First element using positive indexing

print("Last element:", lst[-1]) # Last element using negative indexing

# Using both positive and negative indexes

print("Second element from the beginning:", lst[1]) # Positive index

print("Second element from the end:", lst[-2]) # Negative index

# Example list

lst = [10, 20, 30, 40, 50]

mixed\_indexing(lst)

**Repercussions:**

* There are no direct repercussions from using both positive and negative indexing, as long as the indices are within the valid range of the list.
* However, if you attempt to access an index that is out of bounds (for example, using lst[10] on a list with only 5 elements), you will get an **IndexError**.

**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.**

**Answer:**  
The most efficient way to create a list with 1,000 elements, all initialized to the same value, is by using the **multiplication operator** with a list. This method is both fast and concise.

lst = [0] \* 1000 # Creates a list with 1,000 elements, all set to 0

This is more efficient than using a loop or list comprehension, as it directly initializes the list in memory.

**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.)**

**Answer:**  
You can use **list slicing with a step** to achieve this. The step parameter in slicing allows you to pick elements at regular intervals.

Here’s an example to get every other element (i.e., the 1st, 3rd, 5th, 7th, etc.) from a list:

lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

sliced\_lst = lst[::2] # This will take every second element starting from index 0

print(sliced\_lst) # Output: [1, 3, 5, 7, 9]

In the slicing syntax lst[start:stop:step], setting the step to 2 picks every second element.

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

**Answer:**

* **Indexing:**
  + Accesses **a single element** in a sequence (like a list, string, etc.) by specifying its position (index).
  + Uses a **single integer** index.
  + Example: lst[2] accesses the element at index 2 in lst.
* **Slicing:**
  + Extracts a **subsequence** (subset of elements) from the original sequence.
  + Uses a **start index**, **end index**, and optionally, a **step**.
  + Example: lst[1:4] returns a new list with the elements from index 1 to 3 (but not 4).

**Key Distinction:**

* **Indexing** provides a **single element**.
* **Slicing** provides a **sublist or substring**.

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

**Answer:**  
If one of the slicing indexes is out of range, Python **does not raise an error**. Instead, it will adjust the range to be within the bounds of the list, returning a valid sublist (which could be empty if the indices are invalid).

**Example:**

lst = [1, 2, 3, 4, 5]

sliced\_lst = lst[10:15] # Out of range, but no error

print(sliced\_lst) # Output: [] (empty list)

In this case, even though the slice is out of range, Python simply returns an empty list rather than throwing an error.

**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?**

**Answer:**  
You should **avoid reassigning** the list to a new list inside the function if you want the changes to persist after the function returns.

Here's an example that **won't work**:

def modify\_list(lst):

lst = [10, 20, 30] # Reassigning the list, won't affect the original list

original\_list = [1, 2, 3]

modify\_list(original\_list)

print(original\_list) # Output: [1, 2, 3], no change

To modify the original list, you should directly manipulate the elements of the list without reassigning it.

Here's how it should be done:

def modify\_list(lst):

lst[0] = 10 # Modify an element of the list

original\_list = [1, 2, 3]

modify\_list(original\_list)

print(original\_list) # Output: [10, 2, 3], list has been modified

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

**Answer:**  
An **unbalanced matrix** is a matrix in which the number of elements in each row (or column) is not consistent across all rows (or columns). In other words, it is a **non-rectangular matrix**, where some rows may have more elements than others. This is unlike a **balanced (rectangular) matrix**, where each row has the same number of elements.

**Example of an unbalanced matrix:**

matrix = [

[1, 2, 3],

[4, 5],

[6, 7, 8, 9]

]

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

**Answer:**  
To create arbitrarily large matrices, you need a way to **dynamically allocate** the elements and structure. **List comprehension** and **loops** provide flexible and efficient ways to populate matrices, especially when dealing with varying sizes.

For example, with list comprehension, you can create a matrix with a specified number of rows and columns:

# Creating a 3x3 matrix with list comprehension

matrix = [[0 for \_ in range(3)] for \_ in range(3)]

Similarly, loops allow you to iterate and fill in elements row by row.

matrix = []

for i in range(3):

row = [0] \* 3 # Create a row with 3 elements

matrix.append(row)

These methods are essential because they enable dynamic construction, customization, and manipulation of large matrices, which cannot be done by simply hardcoding the elements.