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

Yes, you can create a program or function in Python that employs both positive and negative indexing. Positive indexing starts from 0, where the first element is indexed as 0, and negative indexing starts from -1, where the last element is indexed as -1.

For example, if you have a list my\_list = [1, 2, 3, 4, 5], you can access the first element using positive indexing with my\_list[0] and the last element using negative indexing with my\_list[-1].

There is no repercussion to using both positive and negative indexing in Python, but it's essential to be consistent in your code to avoid confusion and make it easier to read and debug.

**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.**

You can create a list with 1,000 elements set to the same value in Python using a list comprehension or the multiplication operator.

Using a list comprehension:

my\_list = [0 for \_ in range(1000)]

This creates a list of 1000 elements, each set to the value of 0. You can replace 0 with any other value you want.

Using the multiplication operator:

my\_list = [0] \* 1000

This creates a list of 1000 elements, each set to the value of 0. You can replace 0 with any other value you want.

Both approaches will generate a list with 1000 elements set to the same value in an efficient manner. However, the multiplication operator is slightly faster than a list comprehension for initializing a list with the same 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.)**

You can slice a list to get every other element starting from the first element by using the step size in the slice notation. To get a new list with the first, third, fifth, seventh, and so on elements of a list, you can use a slice with a step size of 2 starting from index 0, like this:

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

new\_list = my\_list[::2] # [1, 3, 5, 7, 9]

In the above example, my\_list[::2] returns a new list that contains every other element from my\_list, starting from the first element.

The slice notation for this operation is [start:end:step], where start is the index to start the slice, end is the index to end the slice (exclusive), and step is the number of elements to skip between each element in the slice. By omitting start and end, we're telling Python to slice the entire list, and by setting step to 2, we're telling Python to skip every other element.

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

Indexing and slicing are both ways to access elements of a sequence, such as a string or a list, in Python. However, they have some important distinctions.

Indexing refers to accessing a single element of a sequence by its position, or index, in the sequence. In Python, indexing starts at 0 for the first element, and can also be done with negative indices to count from the end of the sequence. For example, to access the first element of a list, you would use my\_list[0], and to access the last element of a list, you would use my\_list[-1].

Slicing, on the other hand, refers to accessing a contiguous sequence of elements from a sequence. A slice is created by specifying the start and end indices of the slice, separated by a colon :. For example, to create a slice of the first three elements of a list, you would use my\_list[0:3]. Slicing also supports a third parameter, which specifies the step size for the slice. For example, to create a slice of every other element in a list, you would use my\_list[::2].

In summary, indexing is used to access a single element of a sequence, while slicing is used to access a contiguous sequence of elements. Indexing uses a single integer index, while slicing uses two integer indices separated by a colon. Slicing can also include an optional step size parameter.

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

If one of the slicing expression's indices is out of range in Python, it will not raise an error, but instead, it will return the slice that is within the bounds of the sequence.

If the start index is out of range, Python will treat it as if it were the first index (0) of the sequence. For example, if my\_list has length 5 and you try to slice it from index 10, it will return an empty list [] because there are no elements in my\_list from index 10 onwards.

If the end index is out of range, Python will treat it as if it were the last index (length of the sequence - 1) of the sequence. For example, if my\_list has length 5 and you try to slice it up to index 10, it will return a slice of the entire list from index 0 to index 4.

If both the start and end indices are out of range, Python will still not raise an error and will simply return an empty list [].

It's important to be careful with the indices when slicing sequences in Python to avoid unexpected behavior and ensure that you're getting the slice that you intended.

**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?**

If you want a function to be able to change the values of a list passed to it, you should avoid reassigning the parameter to a new list.

For example, if you define a function that takes a list as an argument like this:

def my\_func(my\_list):

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

and you pass a list to it like this:

my\_list = [10, 20, 30]

my\_func(my\_list)

the original list my\_list will not be changed by the function, because the parameter my\_list was reassigned to a new list.

To change the values of the list passed to the function, you should modify the list in place, by updating or appending its elements, or by using methods like pop() or remove().

For example, if you define a function like this:

def my\_func(my\_list):

my\_list.append(4)

my\_list[0] = 1

and you call it like this:

my\_list = [10, 20, 30]

my\_func(my\_list)

the original list my\_list will be changed by the function, because the function is modifying the original list in place, rather than reassigning the parameter to a new list.

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

There is no standard concept of an "unbalanced matrix" in Python or in linear algebra more generally. It is possible that this term refers to a matrix with unequal numbers of rows and columns, or to a matrix where the rows or columns have different lengths.

In Python, such a matrix might be represented using a list of lists, where each inner list represents a row of the matrix and may have a different number of elements. For example, the following list of lists might be considered an unbalanced matrix:

my\_matrix = [[1, 2, 3],

[4, 5],

[6, 7, 8, 9]]

This matrix has three rows with lengths 3, 2, and 4, respectively.

However, it's important to note that operations like matrix multiplication or addition typically require matrices to have equal numbers of rows and columns and well-defined dimensions. An unbalanced matrix may not be compatible with such operations, and may require special handling or transformations before it can be used in certain computations.

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

It is necessary to use either list comprehension or a loop to create arbitrarily large matrices in Python because matrices are represented using nested lists, and the size of a list must be specified when it is created.

If you want to create a matrix with a specific size, you can use list comprehension or a loop to create the appropriate number of nested lists with the appropriate number of elements. For example, to create a 3x3 matrix with all elements set to zero, you can use list comprehension like this:

my\_matrix = [[0 for j in range(3)] for i in range(3)]

This creates a list of 3 inner lists, each with 3 elements, all set to zero.

However, if you want to create a matrix with an arbitrary size, you cannot specify the number of nested lists or their lengths directly. Instead, you need to use a loop or list comprehension to generate the nested lists dynamically, based on the desired size. For example, to create a matrix with m rows and n columns, you could use the following list comprehension:

my\_matrix = [[0 for j in range(n)] for i in range(m)]

This creates a list of m inner lists, each with n elements, all set to zero.

In general, any operation that requires generating a large, multi-dimensional data structure like a matrix will require some form of iteration or comprehension to create the appropriate structure. Without iteration or comprehension, it would be impossible to specify the necessary number of nested lists or their lengths for an arbitrary size matrix.