1) . What is the difference between enclosing a list comprehension in square brackets and parentheses?

Ans. Parentheses (): When using parentheses in a comprehension, it creates a generator expression instead of a list. A generator expression produces a generator object that generates elements on-the-fly as needed. It does not create the entire list in memory, making it memory-efficient when dealing with large datasets.

Square brackets []: When using list comprehensions with square brackets, it creates a new list by iterating over an iterable and applying the specified expression to each element. The result is a list containing the evaluated expressions.

2) What is the relationship between generators and iterators?

ANs. Generators are a type of iterator in Python. An iterator is any object that implements two methods: \_\_iter\_\_() and \_\_next\_\_(). Generators provide an easy way to create iterators using the yield statement.

Generators: They are functions that contain one or more yield statements. When a generator function is called, it returns a generator object that can be iterated over. The yield statement pauses the function's execution and returns the yielded value to the caller. When the generator is iterated, the function resumes execution from where it was paused until the next yield statement.

Iterators: They are objects that allow iteration over a sequence of elements. Iterators have an \_\_iter\_\_() method that returns the iterator object itself and a \_\_next\_\_() method that returns the next element from the sequence. When there are no more elements to return, the \_\_next\_\_() method raises the StopIteration exception to signal the end of iteration.

3) What are the signs that a function is a generator function?

Ans. The presence of one or more yield statements in the function.

The function typically uses the yield statement to produce values one at a time when called.

4) What is the purpose of a yield statement?

The primary purpose of yield is to generate values lazily on-the-fly, saving memory and processing time when dealing with large datasets or infinite sequences.

5) What is the relationship between map calls and list comprehensions? Make a comparison and contrast between the two.

Ans. Both map() calls and list comprehensions are used to apply a transformation to elements of an iterable. However, they have some differences:

map(function, iterable): map() is a built-in Python function that takes a function and an iterable as input. It applies the function to each element of the iterable and returns an iterator with the results. map() does not create a new list in memory, which makes it memory-efficient, especially when working with large datasets.

Example :

numbers = [1, 2, 3, 4]

squared\_list = [x\*\*2 for x in numbers]

# Result: [1, 4, 9, 16]

Similarity: Both map() and list comprehensions apply a transformation to elements of an iterable and return results based on that transformation.

Difference: map() returns an iterator object, whereas list comprehensions return a new list. List comprehensions may consume more memory as they create the entire list at once, while map() processes elements lazily, one at a time.