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

**Ans:** In Python, list comprehensions can be enclosed in either square brackets [] or parentheses (), but the choice of enclosure affects the result:

Square Brackets []: When you use square brackets, you create a list comprehension. It generates a new list by iterating over a given iterable (e.g., a list, tuple, or range), applying an expression to each element, and collecting the results into a new list. The result is a list.

Parentheses (): When you use parentheses, you create a generator expression. It works similarly to a list comprehension but generates values lazily one at a time, rather than creating an entire list in memory. This is more memory-efficient and can be useful for large datasets.

**2) What is the relationship between generators and iterators?**

**Ans:** An iterator is an object that implements the iterator protocol, which includes two methods: \_\_iter\_\_() and \_\_next\_\_(). Iterators can be used to loop over a sequence of elements, and they keep track of their internal state.

A generator is a type of iterator, but it is defined using a special function called a generator function. Generator functions use the yield statement to produce a series of values lazily. When you call a generator function, it returns a generator object. Each time you call the generator's \_\_next\_\_() method or use it in a loop, it executes the function until it encounters a yield statement, yields the value, and suspends its state. When you call \_\_next\_\_() again, it resumes execution from where it left off.

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

**Ans:** To identify a generator function in Python, look for the following characteristics:

It contains the yield keyword: A generator function must use the yield statement to yield values. This is the primary indicator that you are dealing with a generator function.

It has at least one yield statement: A generator function can have one or more yield statements throughout its body. Each yield statement specifies a value to be yielded to the caller.

It typically has a function definition: Like any other function, a generator function is defined using the def keyword, followed by a function name and parentheses for arguments.

**4) What is the purpose of a yield statement?**

**Ans:** The yield statement serves two primary purposes in Python:

Value Yielding: When used in a generator function, the yield statement specifies a value to be yielded to the caller. It temporarily suspends the function's execution, saves its state, and allows the function to be resumed later. Each time the generator's \_\_next\_\_() method is called, the function continues executing from where it left off, yielding the next value.

Memory Efficiency: yield allows for lazy evaluation of values. Unlike list comprehensions or for loops, which create and store all values in memory at once, generators yield values one at a time as needed. This can significantly reduce memory consumption, making generators suitable for handling 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:** map calls and list comprehensions are both used for transforming elements in an iterable, but they have some differences in terms of usage and behavior:

Map Function:

map is a built-in Python function that takes a function and one or more iterables (e.g., lists) as arguments.

It applies the specified function to each element of the iterable(s) in a lazy manner, without creating a new list immediately.

map returns a map object, which is an iterator. To get the results, you often need to convert it to a list or another iterable type.

It can be memory-efficient when dealing with large datasets, as it generates values one at a time.

List Comprehension:

List comprehensions are a concise way to create lists by specifying an expression and an iterable to iterate over.

They create a new list immediately and store all the resulting values in memory.

List comprehensions are generally more readable and expressive for simple transformations.

Comparison:

map is more suitable when you want to apply a function to one or more iterables and produce an iterator, especially for memory efficiency.

List comprehensions are more suitable when you want to create a new list directly and the transformation is simple and easily readable.

List comprehensions are often considered more Pythonic and are preferred for their simplicity and clarity in many cases.

Contrast:

map is a functional programming construct that emphasizes separation of concerns by separating the transformation logic (the function) from the data. List comprehensions combine the logic and data in one expression.

List comprehensions can include conditions to filter elements, while