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

Enclosing a list comprehension in square brackets creates a list, while enclosing it in parentheses creates a generator expression.

- Square brackets: When using square brackets around a list comprehension, the result is a list object. It eagerly evaluates the entire list comprehension, generating all the elements of the list immediately.

Example: `[x for x in range(5)]` will return `[0, 1, 2, 3, 4]`

- Parentheses: When using parentheses around a list comprehension, it creates a generator expression. It lazily evaluates the comprehension, producing an iterator. The elements of the resulting sequence are computed on-demand as the iterator is iterated over.

Example: `(x for x in range(5))` returns `<generator object <genexpr> at 0x...>`

2) What is the relationship between generators and iterators?

Generators are a specific type of iterator. They are functions or expressions that can be used to iterate over a sequence of values, **generating each value on-the-fly**. **Generators make use of the `yield` statement** to define the sequence of values to be produced. When a generator function is called or a generator expression is iterated over, it returns an iterator object that can be used to retrieve the generated values one at a time.

In summary, generators are a convenient way to create iterators. They allow you to define a sequence of values without the need to store them all in memory at once, which is especially useful for large or infinite sequences.

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

A function in Python is considered a generator function if it contains the `yield` statement at least once within its body. The presence of **the `yield` statement** differentiates a generator function from a regular function. When a generator function is called, it returns a generator object, which can be iterated over to retrieve the values produced by the `yield` statements.

4) What is the purpose of a yield statement?

The `yield` statement is used in generator functions to define the values that the generator will produce when iterated over. It allows a generator function to generate a series of values, one at a time, instead of computing and returning all the values at once.

When a `yield` statement is encountered during iteration, the generator function's state is saved, and the yielded value is returned to the caller. The next time the generator's `__next__()` method is called, the function's execution resumes from where it left off, allowing the generation of the next value in the sequence.

The `yield` statement essentially pauses the execution of the generator function and retains its internal state, enabling it to produce values incrementally.

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

- Similarities:

  - Both map calls and list comprehensions are used to transform and process elements from an iterable.

  - They allow you to apply a function or expression to each element of the iterable.

  - Both can be used to generate a new iterable sequence as a result.

- Differences:

  - Syntax: List comprehensions have a more concise syntax compared to map calls. They provide a compact way to define a new list by applying an expression to each element of an iterable, including the option for conditional filtering. Map calls require a separate function argument to be passed, which can make the code slightly longer.

  - Eager vs. Lazy: List comprehensions eagerly generate the entire list of transformed values immediately. Map calls, on the other hand, return a map object, which is an iterator. The actual transformation is performed lazily, meaning the values are computed on-demand as

  the map object is iterated over. This can be more memory-efficient when dealing with large datasets or infinite sequences.

  - Flexibility: List comprehensions offer more flexibility in terms of the expressions and conditions that can be used to generate the new list. You can include if-else clauses and perform complex operations within the expression. Map calls are generally more limited in this regard since they require a separate function to be passed.

  - Readability: List comprehensions are often considered more readable and expressive, especially for simple transformations and filtering. The concise syntax allows for a more intuitive

representation of the desired operation. Map calls, while still widely used, might require additional cognitive effort to understand the function being applied.

In summary, list comprehensions provide a concise and powerful syntax for transforming and filtering elements from an iterable, while map calls offer a more traditional functional programming approach by applying a separate function to each element. List comprehensions generate the entire list immediately, while map calls lazily evaluate the transformation as the elements are iterated over.