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

**Square Brackets [ ]:**

When we enclose a list comprehension in square brackets, it creates a new list containing the elements generated by the comprehension. List comprehensions are used to create lists by applying an expression to each item in an iterable and optionally filtering the items based on a condition.

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

squared\_numbers = [x \*\* 2 for x in numbers if x % 2 == 0]

# Output : [4, 16]

In this example, the list comprehension generates a new list squared\_numbers containing the squares of even numbers from the original numbers list.

**Parentheses ( ):**

When you enclose a list comprehension in parentheses, it creates a generator expression. A generator expression is similar to a list comprehension but doesn't immediately create a list in memory. This can be more memory-efficient when dealing with large data sets.

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

squared\_numbers\_generator = (x \*\* 2 for x in numbers if x % 2 == 0)

for num in squared\_numbers\_generator:

print(num)

# This will print 4 and 16, one at a time

In this example, the generator expression squared\_numbers\_generator generates squared numbers for even values in the numbers list. The for loop then iterates through the generator and prints the squared values one by one.

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

**a) Iterators:**

An iterator is an object that implements the Python iterator protocol, which consists of two methods: \_\_iter\_\_() and \_\_next\_\_(). These methods allow an object to be iterated over, producing values one at a time. The \_\_iter\_\_() method returns the iterator object itself, and the \_\_next\_\_() method returns the next value from the iterator or raises a StopIteration exception if there are no more values.

**b) Generators:**

Generators are a special type of iterator. They are defined using functions, but instead of using the return statement to produce a value and exit the function, generators use the yield statement to produce a value and suspend the function's execution state. When the generator's \_\_next\_\_() method is called, the function resumes from where it was suspended and continues executing until the next yield statement or until the function exits.

Every generator is an iterator, but not every iterator is a generator. In other words, all generators are iterators, but not all iterators are created using generators.

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

A generator function is a special type of function in Python that uses the yield keyword to produce values one at a time.

The most definitive sign of a generator function is the use of the yield keyword to yield values. This is different from regular functions which use the return keyword to return a value and terminate the function's execution.

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

* The yield statement in Python is used within a function to create a generator.
* When a function encounters a yield statement the function's execution is paused at that point, and the value specified after yield is returned to the caller. However, unlike a regular return statement, the function's internal state is preserved, allowing it to resume execution from where it left off later.
* Each time the generator's \_\_next\_\_() method is called the function resumes execution from the point of the last yield statement and proceeds until it encounters the next yield statement. The value following the yield is returned, and the function's state is paused again.
* The main benefits of using the yield statement and generator functions are memory efficiency (as values are generated on-the-fly) and the ability to work with sequences that may be too large to store entirely in memory.

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

Both map() calls and list comprehensions are used in Python for applying a transformation to elements in an iterable (like a list) and generating a new iterable.

**Map Function:**

The map() function applies a given function to each item in an iterable (e.g., a list) and returns a map object, which is an iterator that produces the transformed values one by one.

**List Comprehension:**

A list comprehension is a concise way to create a new list by applying an expression to each item in an iterable and optionally filtering items based on a condition.

**Comparison:**

* List comprehensions are often more concise and readable, especially for simple transformations.
* map() with a lambda function can become less readable, especially for complex transformations.
* With map(), we can use any callable (function, lambda, etc.) as the transformation.
* List comprehensions use expressions, so they are suitable for simple operations.
* map() produces values lazily, which means the transformation is applied as values are requested, potentially saving memory.
* List comprehensions are evaluated eagerly, generating the entire list at once.
* map() can transform items from any iterable to any other type (e.g., list to tuple).
* List comprehensions generate a list as output.