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

In Python, enclosing a list comprehension in square brackets [ ] creates a list, whereas enclosing it in parentheses ( ) creates a generator expression.

A list comprehension creates a new list in memory, which stores all the elements generated by the expression. This means that a list comprehension can be used to generate a list that can be iterated over multiple times, and the values can be accessed randomly.

On the other hand, a generator expression generates the values one by one on-the-fly, and doesn't store the entire list in memory. This makes generator expressions more memory-efficient than list comprehensions, especially when the list is very large.

For example:

# list comprehension

squared\_numbers = [x\*\*2 for x in range(1, 11)]

print(squared\_numbers) # [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

# generator expression

squared\_numbers = (x\*\*2 for x in range(1, 11))

print(squared\_numbers) # <generator object <genexpr> at 0x7f5ab5e5f6d0>

In general, list comprehensions are preferred when you need to generate a list and use it multiple times, whereas generator expressions are preferred when you need to generate values one-by-one and save memory.

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

In Python, a generator is a special type of iterator. An iterator is an object that implements the iter method, which returns an object that implements the \_\_next\_\_ method. The \_\_next\_\_ method is used to get the next item from the iterator.

A generator is a concise and memory-efficient way to create an iterator in Python. A generator is defined using a generator function or a generator expression, which yields values one by one. When a generator function or expression is called, it returns a generator object, which can be used in a for loop or with the next function to iterate over the values.

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

A function is a generator function in Python if it contains one or more yield statements. The yield statement is used to return a value from the generator function and pause its execution until the next value is requested.

Here's an example of a generator function:

def count\_up\_to(n):

i = 1

while i <= n:

yield i

i += 1

The key signs that this is a generator function are:

The yield keyword: The presence of the yield keyword is the most obvious sign that a function is a generator function.

The return value: A generator function returns a generator object when called, not a list or any other data structure.

The behavior: When a generator function is called, it doesn't execute the function body immediately. Instead, it returns a generator object, which can be used to execute the function body one step at a time, using the next function or a for loop.

Note that generator functions can also have a return statement, but it is used to signal the end of the generator, not to return a value. When the return statement is encountered in a generator function, it raises a StopIteration exception to signal that the generator has exhausted its values.

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

The yield statement in Python is used to return a value from a generator function. A generator function is a special type of function that generates a sequence of values, one at a time, instead of returning a single value. The yield statement is used to return each value in the sequence one at a time, allowing the generator function to be resumed from where it left off when the next value is requested.

**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 ways to apply a function to each element of a sequence in Python, but there are some differences between the two.

map is a built-in function that takes a function and one or more sequences as arguments, and returns a map object, which is an iterator that applies the function to each element of the sequences.

List comprehensions, on the other hand, are a concise and readable way to apply a function to each element of a sequence and create a new list.

Comparing the two, list comprehensions are often considered to be more readable and easier to understand, as they provide a more direct and concise way to apply a function to each element of a sequence and create a new list. map calls can be less readable, especially when the function being applied is complex or the sequence being mapped over is long. However, map calls have the advantage of being more flexible and efficient, as they can handle more than one sequence at a time and generate an iterator instead of a list, which can save memory when the resulting list is large.