**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 object, whereas enclosing it in parentheses `()` creates a generator object.

A list is a collection of items that are stored in memory, and it can be accessed multiple times. On the other hand, a generator is an iterable that generates the values on-the-fly, and it can only be iterated over once.

For example, consider the following code snippet:

```

my\_list = [x\*\*2 for x in range(5)]

my\_generator = (x\*\*2 for x in range(5))

print(my\_list) # Output: [0, 1, 4, 9, 16]

print(my\_generator) # Output: <generator object <genexpr> at 0x7f5f0c0a8dd0>

print(list(my\_generator)) # Output: [0, 1, 4, 9, 16]

```

In the above example, `my\_list` is a list of squares of numbers from 0 to 4, whereas `my\_generator` is a generator expression that does the same thing. However, when we print `my\_generator`, we get a generator object. We can convert the generator object into a list object using the `list()` function, which will generate the values on-the-fly and store them in memory.

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

In Python, a generator is a special type of iterator that generates the values on-the-fly instead of storing them in memory. All generators are iterators, but not all iterators are generators.

An iterator is an object that implements the iterator protocol, which requires the `\_\_iter\_\_()` method to return the iterator object itself and the `\_\_next\_\_()` method to return the next value in the sequence. When there are no more values to return, the `\_\_next\_\_()` method should raise the `StopIteration` exception.

A generator is a type of iterator that uses the `yield` keyword to return values one at a time. When a generator is called, it returns an iterator object that can be used to iterate over the sequence of values. When the `yield` keyword is encountered in the generator function, the current value is returned, and the generator function is paused. The next time the generator is called, it resumes execution from where it left off and continues until the next `yield` statement or until the end of the function is reached.

Here is an example of a simple generator function that generates the sequence of squares of integers from 1 to n:

```

def squares(n):

for i in range(1, n+1):

yield i\*\*2

```

We can use this generator function to create an iterator object, which can be used to iterate over the sequence of squares:

```

my\_iterator = squares(5)

print(next(my\_iterator)) # Output: 1

print(next(my\_iterator)) # Output: 4

print(next(my\_iterator)) # Output: 9

print(next(my\_iterator)) # Output: 16

print(next(my\_iterator)) # Output: 25

```

In summary, generators are a type of iterator that generate values on-the-fly using the `yield` keyword. All generators are iterators, but not all iterators are generators.

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

In Python, a generator function is a special type of function that returns an iterator object using the `yield` keyword. Here are some signs that a function is a generator function:

1. It uses the `yield` keyword: The `yield` keyword is used to return values from the generator function one at a time, and to pause the function until the next value is requested.

2. It does not use the `return` keyword to return a value: Unlike regular functions, generator functions do not use the `return` keyword to return a value. Instead, they use `yield` to return a value, and when the function ends, the `StopIteration` exception is raised automatically.

3. It has a function signature similar to a regular function: Generator functions have a function signature similar to regular functions, with the `def` keyword, the function name, and any parameters that the function takes.

4. It is called using the same syntax as a regular function: Generator functions are called using the same syntax as regular functions, with the function name and any arguments passed in parentheses.

Here is an example of a generator function that generates the Fibonacci sequence:

```

def fibonacci():

a, b = 0, 1

while True:

yield a

a, b = b, a + b

```

In this example, the function `fibonacci()` uses the `yield` keyword to return the Fibonacci numbers one at a time, and it does not use the `return` keyword. When the function is called, it returns an iterator object that can be used to iterate over the sequence of Fibonacci numbers.

To summarize, the signs that a function is a generator function are that it uses the `yield` keyword, it does not use the `return` keyword to return a value, it has a function signature similar to a regular function, and it is called using the same syntax as a regular function.

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

In Python, the `yield` statement is used in generator functions to return values one at a time and to pause the function until the next value is requested. When the `yield` statement is encountered in a generator function, the current value is returned, and the function is paused until the next value is requested.

The purpose of the `yield` statement is to allow generator functions to generate a sequence of values on-the-fly, without having to store the entire sequence in memory. This is useful for generating large or infinite sequences of values, or for generating values that depend on some external state.

Here is an example of a generator function that generates a sequence of random numbers between 0 and 1:

```

import random

def random\_numbers(n):

for i in range(n):

yield random.random()

```

In this example, the `random\_numbers()` function uses the `yield` statement to return a random number between 0 and 1, one at a time. When the function is called, it returns an iterator object that can be used to iterate over the sequence of random numbers.

To generate the next value in the sequence, the iterator object is passed to the `next()` function, which resumes the generator function and runs it until the next `yield` statement is encountered. When there are no more values to return, the generator function automatically raises the `StopIteration` exception.

In summary, the purpose of the `yield` statement is to allow generator functions to generate a sequence of values on-the-fly, without having to store the entire sequence in memory, and to pause the function until 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()` and list comprehensions in Python are used to apply a function to each element of a sequence and return a new sequence based on the results of the function. However, they have some differences in terms of syntax and functionality.

Syntax:

- `map(function, iterable)` takes a function and an iterable as arguments and applies the function to each element of the iterable, returning a map object.

- `[expression for item in iterable]` is the syntax for a list comprehension, where `expression` is the function applied to each element of the iterable, and `item` is the current element of the iterable.

Functionality:

- `map()` is useful when we want to apply a function to each element of an iterable and return a new iterable with the results. The output of `map()` is a map object that can be converted to a list or other iterable if needed.

- List comprehensions are more versatile and can be used not only to apply a function to each element of an iterable but also to filter or manipulate the elements in various ways. List comprehensions can be used to create a new list, set, or dictionary based on the results of applying a function or expressions to the elements of an iterable.

Here is an example of using `map()` and a list comprehension to apply a function to a list of numbers:

```

# using map()

def square(x):

return x \*\* 2

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

squared\_numbers = map(square, numbers)

print(list(squared\_numbers)) # Output: [1, 4, 9, 16, 25]

# using a list comprehension

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

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

print(squared\_numbers) # Output: [1, 4, 9, 16, 25]

```

In this example, both `map()` and a list comprehension are used to square each number in a list. The results are the same in both cases, but the syntax and functionality are different.

In summary, `map()` and list comprehensions are both used to apply a function to each element of an iterable, but list comprehensions are more versatile and can be used to filter or manipulate the elements in various ways. `map()` is useful when we only want to apply a function to each element of an iterable and return a new iterable with the results.