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

**Ans:-** In python. List comprehensions are a concise way to create lists based on existing sequences or iterable objects. When using lists comprehensions, you can enclose them in either square brackets([]) or parentheses. However, the choice between the square brackets and parentheses affects the type of object that is created.

1. **Square brackets :-** When you enclose a list comprehension into a square brackets, it creates a new list object. The resulting object is a list containing the elements generated by the list comprehension. Example :-

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

**print(squares) # Output: [0, 1, 4, 9, 16]**

1. **Parentheses :-** When you enclose a list comprehension in parentheses, it creates a generator object. A generator is an iterable that produces values on-the-fly, as they are requested. Unlike a list, a generator doesn’t store all the generated values in memory at once. Example:-

**squares = (x \*\* 2 for x in range(5))**

**print(squares) # Output: <generator object <genexpr> at 0x00000123456789AB>**

2) What is the relationship between generators and iterators?

**Ans:-** Generators and iterators are closely related concepts in python. As generators are a type of iterator. Both generators and iterators are used to iterate over a sequence of values, but they differ in how they are implemented and how they generate the values.

An iterator is an object that implements the iterator protocol, which consists of two methods:- **‘\_\_iter\_\_()’**  and ‘**\_\_next\_\_()’.** The ‘\_\_iter\_\_()’ method returns the iterator object itself, and the ‘\_\_next\_\_()’ method returns the next value in the sequence or raises a ‘**stop separation’** exception, if there are no more values. Example :-

**class MyIterator:**

**def \_\_init\_\_(self, limit):**

**self.limit = limit**

**self.current = 0**

**def \_\_iter\_\_(self):**

**return self**

**def \_\_next\_\_(self):**

**if self.current < self.limit:**

**value = self.current**

**self.current += 1**

**return value**

**else:**

**raise StopIteration**

**# Using the iterator**

**my\_iterator = MyIterator(5)**

**for num in my\_iterator:**

**print(num)**

Generators on the other hand, are a convenient way to create iterators. They are defined using special kind of function called a generator function. A generator function uses the ‘**yeild’** keyword to yield values one at a time, instead of returning a value like a regular function. When a generator function is called, it returns a generator object, which is an iterator. Example :-

**def my\_generator(limit):**

**current = 0**

**while current < limit:**

**yield current**

**current += 1**

**# Using the generator**

**my\_generator\_obj = my\_generator(5)**

**for num in my\_generator\_obj:**

**print(num)**

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

**Ans :-** In python, you can identify whether a function is a generator function based on a few distinctive sign:-

1. **Use of the ‘yield’ keyword :-**  Generator function makes use of the yield keyword to produce a series of values. This is the key feature that distinguishes them from regular functions. A generator function will typically have one or more ‘**yield’** statements inside its body.
2. **Return value is a generator object :-** When a generator function is called, it returns a generator object. This object is an iterator and can be used to iterate over the generated values. You can check if a function returns a generator object by invoking it and verifying the type of the returned value.
3. **Absence of a return statement :-** Generator function typically does not contain a ‘**return’** statement or may have a bare ‘**return’** statement without any value. If a generator function encounters a ‘**return’**  statement, it will raise a ‘**stopiteration’** exception, indicating the end of iteration. However, the primary mechanism for producing values in generator function is the ‘**yield’** statement.

4) What is the purpose of a yield statement?

**Ans:-** The ‘**yield’** statement in python is a powerful construct used in generator functions to define a point at which the function can pause its execution and yield a value. It serves two main purposes :-

1. **Producing a series of values :-** The primary purpose of the ‘**yield’**  statement is to generate a series of values one at a time, allowing the generator function to produce a sequence of values lazily. Each time the ‘**yield’**  statement is encountered, the function temporarily suspends its execution, yields the specified value and remembers its internal state.

**Maintaining function state between invocations :-** Another important aspect of the ‘**yield’**  statement is that it allows the generator function to retain its internal state between invocations. When a generator function yields a value and then gets called again, its resumes its execution from the point of the last yield statement, using the remembered state. This feature allows the generator function to produce values on-the-fly, as they are requested, and conserve memory. They don’t generate and store all the values in the memory at once but generate values dynamically when needed.

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

**Ans:-** Botha ‘**map’** calls and list comprehensions are constructs in python used to transform or process elements from existing iterables. While they serve a similar purpose, they have some differences in terms of syntax, functionality and readability :-

1. **Syntax :-**
2. **Map call :-** The ‘**map’**  function takes two arguments , a function and an iterable. It applies the function to each element of the iterable and returns a map object, which is an iterator. The syntax is **‘map(function, iterable)’.**
3. **List comprehension :-** List comprehension has a more concise and expressive syntax. They consist of an expression followed by a ‘**for’**  clause and optional ‘**if’** clauses. The syntax is **‘[expression for item in iterable if condition]’.**

**2. Functionality :-**

1. **Map call :-** The ‘**map’**  function applies a specified function to each element of an iterable, and the results are collected in a map object. The function can be a built-in function or a custom defined function. It can also take multiple iterable as arguments.
2. **List comprehension :-** List comprehension allow you to generate a new list by applying an expression to each element of an iterable. The expression can be a simple operation or a more complex computation. You can also use ‘**if’**  clause to filter elements based on conditions.

3.  **Return type :-**

1. **Map call :-** The ‘**map’**  function returns a map object, which is an iterator. To obtain the results as a list, you need to convert the map object into a list, using the ‘**list()’**  function.
2. **List comprehension :-** List comprehension directly return a new list containing the transformed elements. No further conversion is required.

4. **Readability and expressiveness :-**

1. **Map call :-** ‘**map’** calls can be useful when you want to apply a function to each element of an iterable, especially if the transformation is simple and the function is already defined. However, the syntax of ‘**map’**  calls can be less readable and less intuitive, particularly when using lambda functions, or multiple iterables.
2. **List comprehensions :-** List comprehensions offer a more expressive and a readable way to create new lists based on existing iterables. They provide a compact and self contained syntax that is often easier to understand and maintain. List comprehensions are especially advantageous for complex transformations or when multiple conditions need to be applied.