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

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

1. **List comprehension**: When using square brackets, a list comprehension is created. It evaluates the expression inside and generates a list of results. This means that the **entire list** is **created** **in** **memory** **at** **once**. It **allows** for **random access** to elements, **multiple iterations**, and **modifications** to the list.

Example:

```python

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

print(my\_list) # Output: [0, 1, 2, 3, 4]

```

2. **Generator expression**: When using parentheses, a generator expression is created. It **generates** **values** **on-the-fly as they are needed**, **without storing them all in memory at once**. It produces values one at a time, lazily, as they are requested. It is **memory-efficient** but does **not allow random access** or **multiple iterations**.

Example:

```

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

print(my\_generator) # Output: <generator object <genexpr> at 0x000001>

```

In summary, if you need to access the elements multiple times or modify the resulting sequence, use a list comprehension enclosed in square brackets. If you only need to iterate over the elements once and memory efficiency is a concern, use a generator expression enclosed in parentheses.

1. What is the relationship between generators and iterators?

Ans:

1. **Iterator**:

- An iterator is an **object that represents a stream of data** and **implements** the **`\_\_iter\_\_()`** and **`\_\_next\_\_()` methods**.

- The `\_\_iter\_\_()` method returns the iterator object itself, and the `\_\_next\_\_()` method returns the next value from the stream.

- **Iterators maintain the state of iteration**, **allowing you to fetch the next value when needed.**

Example:

```

my\_iter = iter(my\_list)

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

print(next(my\_iter)) # Output: 2

```

2. **Generator**:

- A generator is **a special kind of iterator** that **allows you to define a function that generates a sequence of values lazily**.

- **Instead** of using the **`return` statement**, generators **use** the **`yield` keyword** to yield values one by one.

- Generator functions are defined using the `def` keyword, and they can be iterated over just like any other iterable.

Example:

```

def count\_up\_to(n):

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

yield i

numbers = count\_up\_to(5)

for num in numbers:

print(num)

```

1. What are the signs that a function is a generator function?

The **presence of the `yield` keyword** within a function is a clear sign that it is a generator function. When a function contains a `yield` statement, it indicates that the function will return a generator object when called, rather than a regular value or data structure. The `yield` statement allows the generator function to produce a series of values, one at a time, and pause its execution between yields.

4) What is the purpose of a yield statement?

**Yield**:

- The `yield` keyword is **used inside a generator function** to **yield a value from the sequence being generated**.

- **When** the `yield` statement is **encountered**, the **current state of the function is saved**, and the **yielded value is returned as the next value in the sequence.**

- The **function execution** is then **paused**, and the **control** is **transferred back to the caller**.

- The next time the iterator's `\_\_next\_\_()` method is called, the function resumes execution from where it left off.

Example:

```

def generate\_numbers():

yield 1

yield 2

yield 3

numbers = generate\_numbers()

print(next(numbers)) # Output: 1

print(next(numbers)) # Output: 2

```

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

- Both `map()` and list comprehensions provide ways to transform elements of an iterable, but **list comprehensions offer a more concise and readable syntax compared to `map()`.**

- **List comprehensions generate a new list** containing the transformed elements, while **`map()` returns an iterator**.

Example using `map()`:

```

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

squares = map(lambda x: x\*\*2, numbers)

```

Example using a list comprehension:

```

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

squares = [x\*\*2 for x in numbers]

```