**FSDS MAY BATCH 2022(Python Basics 25)**

**Submitted by: Shubham Tiwari**

Q1)What is the difference between enclosing a list comprehension in square brackets and parentheses?

Ans: Enclosing a list comprehension in square brackets (**[...]**) will create a new list object containing the elements generated by the list comprehension. Enclosing the list comprehension in parentheses (**(...)**), on the other hand, will create a generator object, which is a special type of iterable that can be used to iterate over the elements of the sequence generated by the list comprehension.

# Using square brackets to create a list

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

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

# Using parentheses to create a generator

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

print(my\_generator) # Output: <generator object <genexpr> at 0x7f9a3a5d1c50>

# Iterating over the generator to generate the sequence

for num in my\_generator:

print(num)

# Output:

# 0

# 1

# 4

# 9

# 16

Q2) What is the relationship between generators and iterators?

Ans: 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 item from the sequence, or raises the **StopIteration** exception if there are no more items.

A generator is a special type of iterator that is defined using a function that contains one or more **yield** statements. When the function is called, it returns a generator object that can be used to iterate over the sequence of values generated by the **yield** statements. Each time the **yield** statement is encountered, the function's state is saved, and the value is returned to the caller. The next time the generator's **\_\_next\_\_** method is called, the function's state is restored, and execution continues from where it left off, until the next **yield** statement is encountered.

Q3) What are the signs that a function is a generator function?

Ans: Here are some signs that a function is a generator function:

1. The function contains one or more **yield** statements: The **yield** statement is used to produce a value from the generator function. When the function is called, it returns a generator object that can be used to iterate over the sequence of values generated by the **yield** statements.
2. The function has a **yield** statement instead of a **return** statement: When a **yield** statement is encountered, the function's state is saved, and the value is returned to the caller. The next time the generator's **\_\_next\_\_** method is called, the function's state is restored, and execution continues from where it left off, until the next **yield** statement is encountered. This is different from a regular function, which terminates when a **return** statement is encountered.
3. The function can be used in a **for** loop: Since a generator function returns a generator object, it can be used to generate values in a **for** loop or other iterable contexts, like list comprehensions or the **sum** function.

Q4) What is the purpose of a yield statement?

Ans: The **yield** statement is used in a generator function to produce a sequence of values on-the-fly, without generating them all at once and storing them in memory. When a generator function is called, it returns a generator object, which can be used to iterate over the sequence of values generated by the **yield** statements.

The purpose of the **yield** statement is to produce a value from the generator function and temporarily suspend its execution, saving its state for later resumption. When the **yield** statement is encountered, the function's state is saved, and the value is returned to the caller. The next time the generator's **\_\_next\_\_** method is called, the function's state is restored, and execution continues from where it left off, until the next **yield** statement is encountered.

This allows generator functions to generate large sequences of values, one at a time, without using up a lot of memory. Instead of generating all the values at once and storing them in a list or other data structure, a generator function can generate each value as it is needed, only keeping one value in memory at a time. This can be especially useful when dealing with large or infinite sequences, such as reading lines from a file or generating an infinite stream of random numbers.

Q5) What is the relationship between map calls and list comprehensions? Make a comparison and contrast between the two.

Ans: In Python, both **map()** calls and list comprehensions can be used to apply a function to every element of an iterable and generate a new iterable with the transformed values. However, there are some differences between the two.

Comparison:

* Both **map()** calls and list comprehensions apply a function to every element of an iterable and generate a new iterable with the transformed values.
* Both **map()** calls and list comprehensions can be used to write concise and readable code.

Contrast:

* **map()** returns an iterator object that generates the transformed values lazily, while list comprehensions generate a list with all the transformed values at once.
* **map()** only applies a single function to the iterable, while list comprehensions can include multiple transformations and conditions in a single expression.
* **map()** can only be used to apply functions that take one argument, while list comprehensions can be used with any expression that can be applied to the elements of the iterable .
* List comprehensions can be more flexible and expressive than **map()** calls, but they can also be more verbose and harder to read for complex transformations.

# Using map()

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

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

print(list(squares)) # Output: [1, 4, 9, 16, 25]

# Using a list comprehension

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

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

print(squares) # Output: [1, 4, 9, 16, 25]

In this example, both **map()** and a list comprehension are used to generate a new list with the squares of the numbers in the original list. The **map()** call uses a lambda function to apply the **\*\*2** operation to each element of the list, while the list comprehension uses a **for** loop to iterate over the elements of the list and apply the **\*\*2** operation to each element.

In general, the choice between using **map()** and a list comprehension depends on the specific task and the readability and maintainability of the resulting code. For simple transformations, **map()** can be more concise and readable, while for more complex transformations, list comprehensions can be more expressive and flexible.