What is Python Generator?

Python Generators are the capabilities that return the crossing object and used to make iterators. It simultaneously traverses all of the items. The generator can also be an expression with syntax similar to that of Python's list comprehension.

There is a lot of complexity in creating iteration in Python; we need to implement **__iter__()** and **__next__()** method to keep track of internal states.

It is a lengthy process to create iterators. That's why the generator plays an essential role in simplifying this process. If there is no value found in iteration, it raises **StopIteration** exception.

How to Create Generator function in Python?

In Python, creating a generator is not difficult at all. It is like the typical capability characterized by the def catchphrase and utilizations a yield watchword rather than return. Or on the other hand we can say that if the body of any capability contains a yield explanation, it naturally turns into a generator capability. considering about the accompanying model:

```
def simple():
for i in range(10):
  if(i%2==0):
      yield i

#Successive Function call using for loop
for i in simple():
  print(i)
```

Output:

```
0
2
4
6
8
```

yield vs. return

The yield articulation is answerable for controlling the progression of the generator capability. By saving all states and yielding to the caller, it puts an end to the function's execution. Later it resumes execution when a progressive capability is called. In the generator function, we can make use of the multiple yield statement.

The return explanation returns a worth and ends the entire capability and just a single return proclamation can be utilized in the capability.

Using multiple yield Statement

We can use the multiple yield statement in the generator function. Consider the following example.

```
def multiple_yield():
    str1 = "First String"
    yield str1

    str2 = "Second string"
    yield str2

    str3 = "Third String"
    yield str3
obj = multiple_yield()
print(next(obj))
print(next(obj))
print(next(obj))
```

Output:

```
First String
Second string
Third String
```

Difference between Generator function and Normal function

<
Typical capability contains just a single Lreturn explanation while generator capability can contain at least one yield proclamation.

The normal function is immediately halted and the caller is given control when the generator functions are called.

The states of the local variables are retained between calls.

StopIteration exception is raised automatically when the function terminates.

Generator Expression

We can undoubtedly make a generator articulation without utilizing client characterized capability. It is equivalent to the lambda capability which makes a mysterious capability; An anonymous generator function is created by the generator's expressions.

The portrayal of generator articulation resembles the Python list perception. The only difference is that round parentheses take the place of square brackets. The generator expression only calculates one item at a time, whereas the list comprehension calculates the entire list.

Consider the following example:

```
list = [1,2,3,4,5,6,7]

# List Comprehension
z = [x**3 for x in list]

# Generator expression
a = (x**3 for x in list)

print(a)
print(z)
```

Output:

```
<generator object <genexpr> at 0x01BA3CD8>
[1, 8, 27, 64, 125, 216, 343]
```

In the above program, list comprehension has returned the list of cube of elements whereas generator expression has returned the reference of calculated value. Instead of applying a **for loop**, we can also call **next()** on the generator object. Let's consider another example:

```
list = [1,2,3,4,5,6]

z = (x**3 for x in list)

print(next(z))

print(next(z))

print(next(z))

print(next(z))
```

Output:

```
1
8
27
64
```

Note:- When we call the next(), Python calls __next__() on the function in which we have passed it as a parameter.

In the above program, we have used the **next()** function, which returned the next item of the list.

Example: Write a program to print the table of the given number using the generator.

```
def table(n):
    for i in range(1,11):
        yield n*i
          i = i+1

for i in table(15):
    print(i)
```

Output:

```
15
30
45
60
75
90
105
120
135
150
```

In the above example, a generator function is iterating using for loop.

Advantages of Generators

There are various advantages of Generators. Few of them are given below:

1. Easy to implement

Generators are easy to implement as compared to the iterator. In iterator, we have to implement **__iter__()** and **__next__()** function.

2. Memory efficient

For many sequences, generators utilize memory efficiently. The generator function, on the other hand, calculates the value and suspends their execution, whereas the normal function returns a sequence from the list, which first creates the entire sequence in memory before returning the result. It resumes for progressive call. A limitless succession generator is an extraordinary illustration of memory streamlining. Let's talk about it using the sys.getsizeof() function in the example below.

```
import sys
# List comprehension
nums_squared_list = [i * 2 for i in range(1000)]
print(sys.getsizeof("Memory in Bytes:"nums_squared_list))
# Generator Expression
nums_squared_gc = (i ** 2 for i in range(1000))
print(sys.getsizeof("Memory in Bytes:", nums_squared_gc))
```

Output:

```
Memory in Bytes: 4508
Memory in Bytes: 56
```

We can observe from the above output that list comprehension is using 4508 bytes of memory, whereas generator expression is using 56 bytes of memory. It means that generator objects are much efficient than the list compression.

3. Pipelining with Generators

Information Pipeline gives the office to handle huge datasets or stream of information without utilizing additional PC memory.

Let's say we have a famous restaurant's log file. The log document has a section (fourth segment) that monitors the quantity of burgers sold consistently and we need to total it to find the complete number of burgers sold in 4 years. The generator can create a pipeline using a series of operations in that scenario. The code for it is as follows:

```
with open('sells.log') as file:
burger_col = (line[3] for line in file)  per_hour = (int(x) for x in burger_col if x != 'N/A' )
print("Total burgers sold = ",sum(per_hour))
```

4. Generate Infinite Sequence

The generator can produce infinite items. Infinite sequences cannot be contained within the memory and since generators produce only one item at a time, consider the following example:

```
def infinite_sequence():
    num = 0
    while True:
        yield num
            num += 1

for i in infinite_sequence():
    print(i)
```

Output:

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
.........
..........
315
316
317
Traceback (most recent call last):
File "C:\Users\DEVANSH SHARMA\Desktop\generator.py", line 33, in <module>
print(i)
KeyboardInterrupt
```