GENERATORS IN PYTHON

What is generator?

Generator provides a way of creating iterators

A generator in Python is a function with unique abilities. We can either suspend or resume it at run-time.

It returns an iterator object which we can step through and access a single value in each iteration.

First, we require to write a class and implement the __iter__() and __next__() methods.

Secondly, we need to manage the internal states and throw StopIteration exception when there is no element to return.

Why generators?

Create Generator In Python

Python generator gives an alternative and simple approach to return iterators.

1. Generator Function

use the yield statement instead of the return.
It notifies Python interpreter that the function is a generator and returns an iterator.

```
# Generator Function Syntax
#
def gen_func(args):
    ...
    while [cond]:
        ...
        yield [value]
```

The return statement is the last call in a function, whereas the yield temporarily suspends the function, preserves the states, and resumes execution later.

Python generator function to determine the next value of a Fibonacci sequence.

```
# Demonstrate Python Generator Function

def fibonacci(xterms):
    # first two terms
    x1 = 0
    x2 = 1
    count = 0

    if xterms <= 0:
       print("Please provide a +ve integer")
    elif xterms == 1:
       print("Fibonacci seq upto",xterms,":")
       print(x1)
    else:
       while count < xterms:
           xth = x1 + x2
           x1 = x2
           x2 = xth
           count += 1
           yield xth

fib = fibonacci(5)

print(next(fib))
print(next(fib))
print(next(fib))
print(next(fib))
print(next(fib))
print(next(fib))
```

The generator function has a while loop to calculate the next value of a Fibonacci series.

After creating the generator function, we've called it and passed five as the input argument. It will only return the iterator object.

The generator function will not execute execution until we call the next() method over the returned object, i.e., 'fib.' We are doing six such steps to iterate over the 'fib' object.

The first five next() calls have succeeded and return the respective element of the Fibonacci series. However, the last one raised the StopIteration exception as the iterator had no items left.

The code prints the following output after execution.

```
Traceback (most recent call last):
  File "C:/Python/Python35/python_generator.py", line 29, in
    print(next(fib))
StopIteration
```

2**. Generator Expression**

Python allows writing generator expressions to create anonymous generator functions.

This procedure is similar to a lambda function creating an anonymous function.

The syntax of a generator expression is same as of a list comprehension in Python. However, the former uses the round parentheses instead of square brackets.

```
# Generator Expression Syntax
#
gen_expr = (var**(1/2) for var in seq)
```

Another difference between a list comprehension and a generator expression is that the LC gives back the full list whereas the generator expression returns one value at a time.

```
# Demonstrate Python Generator Expression
# Define the list
alist = [4, 16, 64, 256]

# Find square root using the list comprehension
out = [a**(1/2) for a in alist]
print(out)

# Use generator expression to calculate the square root
out = (a**(1/2) for a in alist)
print(out)
print(next(out))
print(next(out))
print(next(out))
print(next(out))
print(next(out))
```

While executing the above example, firstly, the list comprehension returns the list containing the square roots of all elements. So we get the conclusive result here.

Next, the generator expression produces an iterator object which gives one result at a time. The size of the list is four. So we have four successive next() method calls which print the square root of respective list elements.

Since we called the next() method one more time, it caused the StopIteration exception. Please check from the below output.

```
[2.00, 4.0, 8.00, 16.0]
 at 0x000000000359E308>
2.0
4.0
8.0
16.0
Traceback (most recent call last):
  File "C:/Python/Python35/python_generator.py", line 17, in
    print(next(out))
StopIteration
```

How To Use Generator?

1. Using Next() Method

It (next()) is the most common way we can request a value from the generator function. Calling the next() method triggers its execution which in turn gives a result back to the caller.

```
# Generator next() Method Demo
#
alist = ['Python', 'Java', 'C', 'C++', 'CSharp']
def list_items():
    for item in alist:
        yield item

gen = list_items()

iter = 0

while iter < len(alist):
    print(next(gen))
    iter += 1
```

Each next() call on the generator object causes its function to execute until it finds a yield statement. Then, Python sends the yielded value back to the caller and preserves the state of the generator for future use.

2. Using For Loop

we can also use the for loop to iterate over the generator object

```
# Generator For Loop Demo
#
alist = ['Python', 'Java', 'C', 'C++', 'CSharp']
def list_items():
    for item in alist:
        yield item
```

```
gen = list_items()

for item in gen:
    print(item)
```

Return Vs. Yield

The return is a final statement of a function. It provides a way to send some value back.

While returning, its local stack also gets flushed. And any new call will begin execution from the very first statement.

yield preserves the state between subsequent function calls.

It resumes execution from the point where it gave back the control to the caller, i.e., right after the last yield statement.

Generator Vs. Function

We have listed down a few facts to let you understand the difference between a generator and a regular function.

- A generator uses the yield statement to send a value back to the caller whereas a function does it using the return.
- The generator function can have one or more than one yield call.
- The yield call pauses the execution and returns an iterator whereas the return statement is the last one to be executed.
- The next() method call triggers the execution of the generator function.
- Local variables and their states retain between successive calls to the next() method.
- Any additional call to the next() will raise the StopIteration exception if the there is no further item to process.
-

Following is the Generator function having multiple yield statements.

```
# Python Generator Function with Multiple Yield

def testGen():
    x = 2
    print('First yield')
    # Generator function has many yield statements
    yield x

    x *= 1
    print('Second yield')
    yield x

    x *= 1
    print('Last yield')
    yield x

# Call the generator
iter = testGen()

# Invoke the first yield
next(iter)

# Invoke the second yield
next(iter)

# Invoke the last yield
next(iter)
```

After executing the above coding snippet, we get the following output.

```
First yield
Second yield
Last yield
```

When To Use A Generator?

There are many use cases where generators can be useful. We have mentioned some of them here:

- Generators can help to process large amounts of data. They can let us do the calculation when we want, also known as the lazy evaluation. The stream processing uses this approach.
- We can also stack the generators one by one and use them as pipes as we do with the Unix pipes.
- The generators can also let us establish concurrency.
- We can utilize Generators for reading a vast amount of large files. It will help in keeping the code cleaner and leaner by splitting the entire process into smaller entities.
- Generators are super useful for web scraping and help increasing crawl efficiency. They can allow us to fetch the single page, do some operation and move on to the next. This approach is far more efficient and straightforward than retrieving all pages at once and then use another loop to process them.

Why Use Generators?

Generators provide many programming-level benefits and extend many run-time advantages which influence programmers to use them.

1. Programmer-Friendly Feature

It seems like a complicated concept, but the truth is that you can easily incorporate them into programs. They are a perfect alternative for the iterators.

Let's consider the following example to implement the Arithmetic Progression using the Iterator Class.

```
# Generate Arithmetic Progression Using Iterator Class
#
class AP:
    def __init__(self, a1, d, size):
        self.ele = a1
        self.diff = d
        self.len = size
        self.count = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.count >= self.len:
            raise StopIteration
        elif self.count == 0:
            self.count += 1
            return self.ele
        else:
            self.count += 1
            self.ele += self.diff
            return self.ele

for ele in AP(1, 2, 10):
    print(ele)
```

The same logic is much easier to write with the help of a generator. See the below code.

```
# Generate Arithmetic Progression Using Generator Function
#
def AP(a1, d, size):
    count = 1
    while count <= size:
        yield a1
        a1 += d
        count += 1

for ele in AP(1, 2, 10):
    print(ele)
```

2**. Memory Agnostic**

If we use a regular function to return a list, then it will form the full sequence in the memory before sending to the caller. Such an operation would cause high memory usage and become extremely inefficient.

On the contrary, using a generator will consume less memory, and your program will become much more efficient as it will have to process only one item at a time.

3. Capable Of Handling Big Data

Generators can be useful if you have to deal with data of enormous size such as the Big Data. They work as an infinite stream of data.

We can not contain data of such magnitude in memory. But the generator which gives us one value at a time does represent an infinite stream of data.

The following code can produce all the prime numbers theoretically.

```
# Find All Prime Numbers Using Generator
#
def find_prime():
    num = 1
    while True:
        if num > 1:
            for i in range(2, num):
                if (num % i) == 0:
                    break
            else:
                yield num
        num += 1

for ele in find_prime():
    print(ele)
```

4. Generator Pipeline

With the help of generators, we can create a pipeline of different operations. It is a cleaner way to sub-divide responsibilities among various components and then integrates them to achieve the desired result.

In the below example, we’ve chained two functions, the first finds the prime number between 1 to 100, and the latter selects the odd one from them.

```
# Chain Multiple Operations Using Generator Pipeline
#
```

```
def find_prime():
    num = 1
    while num < 100:
        if num > 1:
            for i in range(2, num):
                if (num % i) == 0:
                    break
            else:
                yield num
        num += 1

def find_odd_prime(seq):
    for num in seq:
        if (num % 2) != 0:
            yield num

a_pipeline = find_odd_prime(find_prime())

for a_ele in a_pipeline:
    print(a_ele)
```