Python Generators

There was a lot of overhead in building an iterator in Python; we had to implement a class with `__iter__()` and `__next__()` method, keep track of internal states, raise `StopIteration` when there was no values to be returned etc. This is both lengthy and counter intuitive. Generator comes into rescue in such situations.

Python generators are a simple way of creating iterators. All the overhead that we mentioned above are automatically handled by generators in Python. Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

Creating a Generator in Python

It is fairly simple to create a generator in Python. It is as easy as defining a normal function with `yield` statement instead of a `return` statement. If a function contains at least one `yield` statement (it may contain other `yield` or `return` statements), it becomes a generator function. Both `yield` and `return` will return some value from a function. The difference is that, while a `return` statement terminates a function entirely, `yield` statement pauses the function saving all its states and later continues from there on successive calls. Here is how a generator function differs from a normal function.

- Generator function contains one or more `yield` statement.
- When called, it returns an object (iterator) but does not start execution immediately.
- Methods like `__iter__()` and `__next__()` are implemented automatically. So we can iterate through the items using `next()`.
- Once the function yields, the function is paused and the control is transferred to the caller.
- Local variables and theirs states are remembered between successive calls.
- Finally, when the function terminates, `StopIteration` is raised automatically on further calls.

Here is an example to illustrate all of the points stated above. We have a generator function named `my_gen()` with several `yield` statements.

```
def my_gen():
    """a simple generator function"""
    n = 1
    print("This is printed first")
    # Generator function contains yield statements
    yield n

    n += 1
    print("This is printed second")
    yield n
```

```
    n += 1
    print("This is printed at last")
    yield n
```

An interactive run in the interpreter is given below.

```
>>> # It returns an object but does not start execution immediately.
>>> a = my_gen()

>>> # We can iterate through the items using next().
>>> next(a)
This is printed first
1
>>> # Once the function yields, the function is paused and the control is transferred to
the caller.

>>> # Local variables and theirs states are remembered between successive calls.
>>> next(a)
This is printed second
2
>>> next(a)
This is printed at last
3

>>> # Finally, when the function terminates, StopIteration is raised automatically on
further calls.
>>> next(a)
Traceback (most recent call last):
...
StopIteration
>>> next(a)
Traceback (most recent call last):
...
StopIteration
```

One interesting thing to note in the above example is that, the value of variable *n* is remembered between each call. Unlike normal functions, the local variables are not destroyed when the function yields. Furthermore, the generator object can be iterated only once. To restart the process we need to create another generator object using something like `a = my_gen()`.

One final thing to note is that we can use generators with `for` loops directly. This is because, a `for` loop takes an iterator and iterates over it using `next()` function. It automatically ends when `StopIteration` is raised. Check here to know how a for loop is actually implemented in Python.

```
>>> for item in my_gen():
...     print(item)
...
This is printed first
1
This is printed second
2
This is printed at last
3
```

Python Generators with a Loop

The above example is of less use and we studied it just to get an idea of what was happening in the background. Normally, generator functions are implemented with a loop having a suitable terminating condition. Let's take an example of a generator that reverses a string.

```
def rev_str(my_str):
    length = len(my_str)
    for i in range(length - 1,-1,-1):
        yield my_str[i]
```

In this example, we use `range()` function to get the index in reverse order. Here is a call to this function.

```
>>> for char in rev_str("hello"):
...     print(char)
...
o
l
l
e
h
```

It turns out that this generator function not only works with string, but also with other kind of iterables like list, tuple etc.

Python Generator Expression

Simple generators can be easily created on the fly using generator expressions. It makes building generators easy. Same as lambda function creates an anonymous function, generator expression creates an anonymous

generator function. The syntax for generator expression is similar to that of a list comprehension in Python. But the square brackets are replaced with round parentheses.

The major difference between a list comprehension and a generator expression is that while list comprehension produces the entire list, generator expression produces one item at a time. They are kind of lazy, producing items only when asked for. For this reason, a generator expression is much more memory efficient than an equivalent list comprehension.

```
>>> my_list = [1, 3, 6, 10]

>>> # square each term using list comprehension
>>> [x**2 for x in my_list]
[1, 9, 36, 100]

>>> # same thing can be done using generator expression
>>> (x**2 for x in my_list)
<generator object <genexpr> at 0x0000000002EBDAF8>
```

We can see above that the generator expression did not produce the required result immediately. Instead, it returned a generator object with produces items on demand.

```
>>> a = (x**2 for x in my_list)
>>> next(a)
1
>>> next(a)
9
>>> next(a)
36
>>> next(a)
100
>>> next(a)
Traceback (most recent call last):
...
StopIteration
```

Generator expression can be used inside functions. When used in such a way, the round parentheses can be dropped.

```
>>> sum(x**2 for x in my_list)
146
```

```
>>> max(x**2 for x in my_list)
100
```

Why generators are used in Python?

There are several reasons which make generators an attractive implementation to go for.

1. **Easy to Implement**
   Generators can be implemented in a clear and concise way as compared to their iterator class counterpart. Following is an example to implement a sequence of power of 2's using iterator class.

   ```
   class PowTwo:
       def __init__(self, max = 0):
           self.max = max

       def __iter__(self):
           self.n = 0
           return self

       def __next__(self):
           if self.n > self.max:
               raise StopIteration

           result = 2 ** self.n
           self.n += 1
           return result
   ```

   This was lengthy. Now lets do the same using a generator function.

   ```
   def PowTwoGen(max = 0):
       n = 0
       while n < max:
           yield 2 ** n
           n += 1
   ```

   Since, generators keep track of details automatically, it was concise and much cleaner in implementation.

2. **Memory Efficient**
   A normal function to return a sequence will make the entire sequence in memory before returning the result. This is an overkill if the number of items in the sequence is very large. Generator implementation of such

sequence is memory friendly and is preferred since it only produces one item at a time.

3. **Represent Infinite Stream**

   Generators are excellent medium to represent an infinite stream of data. Infinite streams cannot be stored in memory and since generators produce only one item at a time, it can represent infinite stream of data. The following example can generate all the even numbers (at least in theory).

```
def all_even():
    n = 0
    while True:
        yield n
        n += 2
```

4. **Pipelining Generators**

   Generators can be used to pipeline a series of operations. This is best illustrated using an example.

   Suppose we have a log file from a famous fast food chain. The log file has a column (4th column) that keeps track of the number of pizza sold every hour and we want to sum it to find the total pizzas sold in 5 years. Assume everything is in string and numbers that are not available are marked as 'N/A'. A generator implementation of this could be as follows.

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

   This pipelining is efficient and easy to read (and yes, a lot cooler!).