Python Generators

programiz.com/python-programming/generator

Join our newsletter for the latest updates.

In this tutorial, you'll learn how to create iterations easily using Python generators, how it is different from iterators and normal functions, and why you should use it.

Generators in Python

There is a lot of work in building an iterator in Python. We have to implement a class with `__iter__()` and `__next__()` method, keep track of internal states, and raise `StopIteration` when there are no values to be returned.

This is both lengthy and counterintuitive. Generator comes to the rescue in such situations.

Python generators are a simple way of creating iterators. All the work 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).

Create Generators in Python

It is fairly simple to create a generator in Python. It is as easy as defining a normal function, but with a `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.

Differences between Generator function and Normal function

Here is how a generator function differs from a normal function.

- Generator function contains one or more `yield` statements.
- 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 their 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.

```
# A simple generator function
def my_gen():
    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. Run these in the Python shell to see the output.

```
>>> # 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.

```
# A simple generator function
def my_gen():
    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


# Using for loop
for item in my_gen():
    print(item)
```

When you run the program, the output will be:

```
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]


# For loop to reverse the string
for char in rev_str("hello"):
    print(char)
```

Output

```
o
l
l
e
h
```

In this example, we have used the `range()` function to get the index in reverse order using the for loop.

Note: This generator function not only works with strings, but also with other kinds 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.

Similar to the lambda functions which create anonymous functions, generator expressions create anonymous generator functions.

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 a list comprehension produces the entire list while the generator expression produces one item at a time.

They have lazy execution (producing items only when asked for). For this reason, a generator expression is much more memory efficient than an equivalent list comprehension.

```
# Initialize the list
my_list = [1, 3, 6, 10]

# square each term using list comprehension
list_ = [x**2 for x in my_list]

# same thing can be done using a generator expression
# generator expressions are surrounded by parenthesis ()
generator = (x**2 for x in my_list)

print(list_)
print(generator)
```

Output

```
[1, 9, 36, 100]
<generator object <genexpr> at 0x7f5d4eb4bf50>
```

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

Here is how we can start getting items from the generator:

```
# Initialize the list
my_list = [1, 3, 6, 10]

a = (x**2 for x in my_list)
print(next(a))

print(next(a))

print(next(a))

print(next(a))

next(a)
```

When we run the above program, we get the following output:

```
1
9
36
100
Traceback (most recent call last):
  File "<string>", line 15, in <module>
StopIteration
```

Generator expressions can be used as function arguments. 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
```

Use of Python Generators

There are several reasons that make generators a powerful implementation.

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 using an iterator class.

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

    def __iter__(self):
        return self

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

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

The above program was lengthy and confusing. Now, let's 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, the implementation was concise and much cleaner.

2. Memory Efficient

A normal function to return a sequence will create 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 sequences is memory friendly and is preferred since it only produces one item at a time.

3. Represent Infinite Stream

Generators are excellent mediums to represent an infinite stream of data. Infinite streams cannot be stored in memory, and since generators produce only one item at a time, they can represent an infinite stream of data.

The following generator function can generate all the even numbers (at least in theory).

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

4. Pipelining Generators

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

Suppose we have a generator that produces the numbers in the Fibonacci series. And we have another generator for squaring numbers.

If we want to find out the sum of squares of numbers in the Fibonacci series, we can do it in the following way by pipelining the output of generator functions together.

```
def fibonacci_numbers(nums):
    x, y = 0, 1
    for _ in range(nums):
        x, y = y, x+y
        yield x

def square(nums):
    for num in nums:
        yield num**2

print(sum(square(fibonacci_numbers(10))))
```

Output

```
4895
```

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