Generator in Python

How is the Generator different from iterator and what is the usual function, why should we use it? Let's find out all through this article.

In this article, we will work with you to learn how to create an Iterator using Generator in Python. How is the Generator different from iterator and what is the usual function, why should we use it? Find out all through the following content.

Generator in Python

To build an iterator, we need to follow a lot of steps such as: deploy class with *__iter __ ()* and *__next __ () methods,* monitor internal conditions, *StopIteration* occurs when no value is available return.

Generator is used to solve these problems. Generator is a simple way to create iterators.

Simply put, a **generator is a function that returns an object (iterator) that we can repeat (a value at a time).** They also create a list-type object, but you can only browse the elements of the generator once because the generator does not store data in memory, but each iteration will create the next element in the sequence and returns that element.

How to create Generator in Python

To create a generator in Python, you use the keyword `def` just like when defining a function. In the generator, use the `yield` statement to return the elements instead of the `return` statement as usual.

If a function contains at least one `yield` (can have multiple `yield` and add `return`) then this is definitely a generator function. In this case, both `yield` and `return` return values ??from the function.

The difference here is that `return` will completely terminate a function, while `yield` will only pause the states inside the function and then can continue when called in the next time.

For example, when you call the *__next __ () method* for the first time, the generator performs the calculation of the value and then meets the `yield` keyword, it returns the elements at that location, when you call the *__next__* method. *()* for the second time, the generator does not start running at the first position but begins immediately after the first `yield` keyword. Just like that generator creates elements in the sequence, until no more keyword `yield` , then release exception *StopIteration.*

Difference of function generator and regular function

Here are some differences between the generator function and the regular function:

1. The generator function contains one or more `yield` statements.
2. When called, the generator returns an object *(iterator)* but does not start executing immediately.
3. Methods like *__iter __* () and *__next __ ()* are automatically deployed. So we can iterate through the entries using *next ().*
4. `Yield` will pause the function, local variables and their state are remembered between consecutive calls. Each time the `yield` command is run, it will generate a new value.
5. Finally, when the function ends, *StopIteration* will occur if the function continues.

Below is an example to illustrate all the points mentioned above. We have a generator function named `my_gen()` with some `yield` statements.

```
 # Hàm generator ??n gi?
n # Viet boi TipsMake.com def my_gen(): n = 1 print('Doan text nay duoc in dau ti
?a các câu l?
nh yield yield nn += 1 print('Doan text nay duoc in thu hai') yield nn += 1 print
```

Run them in Python shell to see output:

```
 >>> # Tr? v? m?t ??i t??ng nh?ng không b?t ??u th?c thi ngay l?p t?
c. >>> a = my_gen() >>> # Chúng ta có th? l?p qua các m?c b?ng cách s? d?
ng next(). >>> next(a) Doan text nay duoc in dau tien 1 >>> # Yield s? t?
m d?ng hàm, quy?n ?i?u khi?n chuy?n ??n ng??i g?i >>> # Các bi?n c?c b?
 và tr?ng thái c?a chúng ???c ghi nh? gi?a các l?nh g?i liên ti?
p. >>> next(a) Doan text nay duoc in thu hai 2 >>> next(a) Doan text nay duoc in
?i cùng, khi hàm k?t thúc, StopIteration s? x?y ra n?u ti?p t?c g?
i hàm. >>> next(a) Traceback (most recent call last): . StopIteration >>> next(a
```

It can be seen in the above example that the value of variable `n` is remembered between calls, unlike normal functions that end immediately after each call.

When you first call the *next ()* method, the generator performs value calculations and returns the element at that location, when the second *(()* method is called, the generator does not start running in place. The first position starts immediately after the first `yield` keyword. Just like that generator creates elements in the sequence, until no more keyword `yield` , then release exception *StopIteration.*

To restart the process, create another generator object using the object like `a = my_gen()` .

Note : Generator can be used directly for for loops.

The *for* loop takes an iterator and repeats it with the *next ()* function *, which* automatically ends when *StopIteration* occurs.

```
 # Hàm generator ??n gi?
n def my_gen(): n = 1 print('Doan text nay duoc in dau tien') # Hàm Generator ch
?a câu l?
nh yield yield nn += 1 print('Doan text nay duoc in thu hai') yield nn += 1 print
? d?ng vòng l?p for for item in my_gen(): print(item)
```

Run the program, the result is:

```
Doan text nay duoc in dau tien 1 Doan text nay duoc in thu hai 2 Doan text nay du
```

Generator with Python loops

Example of generator reversing sequence.

```
def rev_str(my_str): length = len(my_str) for i in range(length - 1,-1,-1): yielo
?p for ??o ng??c chu?i # Vi?t b?
i TipsMake.com # Output: # o # l # l # e # h for char in rev_str("hello"): printo
```

This example uses the *range ()* function to get the index in reverse order in the *for* loop .

Expression generator

Generator can easily be created using the **expression generator** .

Just like Lambda creates an anonymous function in Python, the generator also creates an anonymous generator expression. The syntax is similar to the list comprehension syntax, but square brackets are replaced with round brackets.

List comprehension returns a list, while the generator expression returns a generator at a time when requested. For this reason, generator expressions use less memory, providing more efficiency than equivalent comprehension lists.

```
 # Kh?i t?o danh sách my_list = [1, 3, 6, 10] # bình ph??ng m?i ph?n t? b?
ng cách s? d?
ng list comprehension # Output: [1, 9, 36, 100] [x**2 for x in my_list] # k
?t qu? t??ng t? khi s? d?ng bi?u th?
c generator # Output: at 0x0000000002EBDAF8> (x**2 for x in my_list)
```

As can be seen from the above example, the generator expression does not produce the necessary result immediately but returns the generator object, each iteration they will create the next element in the sequence and return that element.

```
 # Kh?i t?
o danh sách my_list = [1, 3, 6, 10] a = (x**2 for x in my_list) # Output: 1 print
```

The generator expression used inside functions can omit round brackets.

```
 >>> sum(x**2 for x in my_list) 146 >>> max(x**2 for x in my_list) 100
```

Why should use generator in Python?

Using the generator will bring many attractive effects.

1. Simplify the code, easy to deploy

Generator can help the code to be implemented more clearly and concisely than the same iterator class. To illustrate this, we will take a concrete example.

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

This code is quite long. Now try using the generator function.

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

Here, the generator performs much more neatly and neatly.

2. Use less memory

A normal function when returning a list will store all lists in memory. In most cases, it is not good to use such a large amount of memory.

Generator will use less memory because they only actually produce results when called, generate one element at a time, be effective if we don't need to browse it too many times.

3. Create infinite lists

Generator is a great means to create an infinite flow of data. These infinite streams do not need to store the whole in memory because the generator only generates one element at a time, so it can represent infinite data flow.

The following example can create all even numbers.

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

In general, the choice of using a generator depends on the actual requirements of the job. Think and choose carefully to get the best option for you.

Previous article: Iterator object in Python

Next lesson: Closure in Python