

Python

Generators

by laike9m

laike9m@gmail.com

<https://github.com/laike9m>

学习编程最难的是什么？

You don't know what you don't know

听说过，但不理解 >> 没听说过

Generator function

当你看见 yield, 就说明这是一个生成器函数, 不要把它看作普通的函数

- When invoked, returns a generator object
- Generator objects implement the iterator interface: `.next` (`.__next__` in Python 3)

```
>>> def gen_123():
...     yield 1
...     yield 2
...     yield 3
...
>>> for i in gen_123(): print(i)
1
2
3
>>> g = gen_123()
>>> g
<generator object gen_123 at ...>
>>> next(g)
1
>>> next(g)
2
>>> next(g)
3
>>> next(g)
Traceback (most recent call last):
...
StopIteration
```

Python ≥ 2.6

Generator behavior

- Invoking a generator function builds the generator object but *does not* execute the body of the function

```
>>> def gen_ab():
...     print('starting...')
...     yield 'A'
...     print('here comes B:')
...     yield 'B'
...     print('the end.')
...
>>> for s in gen_ab(): print(s)
starting...
A
here comes B:
B
the end.
>>> g = gen_ab()
>>> next(g)
starting...
'A'
>>> next(g)
here comes B:
'B'
>>> next(g)
Traceback (most recent call last):
...
StopIteration
```

Generator behavior

- The body is executed only when **next** is called, and only up to the following **yield**

```
>>> def gen_ab():
...     print('starting...')
...     yield 'A'
...     print('here comes B:')
...     yield 'B'
...     print('the end.')
...
>>> for s in gen_ab(): print(s)
starting...
A
here comes B:
B
the end.
>>> g = gen_ab()
>>> next(g)
starting...
'A'
>>> next(g)
here comes B:
'B'
>>> next(g)
Traceback (most recent call last):
...
StopIteration
```

到底什么意思？

```
>>> def gen_ab():
```

```
...     print('starting...')
```

```
...     yield 'A'
```

```
...     print('here comes B:')
```

```
...     yield 'B'
```

```
...     print('the end.')
```

g = gen_ab(), 停在这里

第一次调用next(), 停在这里

第二次调用next(), 停在这里

生成器函数可以看成一串事件，yield暂停执行，next恢复执行
“yield是具有暂停功能的return”

```
>>> def gen_ab():
...     print('starting...')
...     yield 'A'
...     print('here comes B:')
...     yield 'B'
...     print('the end.')
...
>>> for s in gen_ab(): print(s)
starting...
A
here comes B:
B
the end.
>>> g = gen_ab()
>>> next(g)
starting...
'A'
>>> next(g)
here comes B:
'B'
>>> next(g)
Traceback (most recent call last):
...
StopIteration
```

斐波那契生成器(Fibonacci)

斐波那契数列:

0,1,1,2,3,5,8,11,19,...

前两个数是0,1, 后一个数是前两个数之和

斐波那契数列生成器

```
def fib(max):  
    a, b = 0, 1 ①  
    while a < max:  
        yield a ②  
        a, b = b, a + b ③
```

```
>>> def fib(max):  
    a, b = 0, 1  
    while a < max:  
        yield a  
        a, b = b, a+b
```

```
>>> for fib_number in fib(100):  
    print(fib_number)
```

```
0  
1  
1  
2  
3  
5  
8  
13  
21  
34  
55  
89
```

for 循环
每次都会调用next()

更深入的讨论: iterator(迭代器)

每次都把返回的a输出
注意yield是“具有暂停功能的return”
return a (= yield a)
fib_number = a
然后打印出来

不要害怕生成器函数里的循环

- 循环就是一个事件流，只不过里面包含了一些条件判断

```
def fib(max):  
    a, b = 0, 1  
    while a < max:  
        yield a  
        a, b = b, a+b
```

等价于

```
def fib(max):  
    a, b = 0, 1  
    if a < max:  
        yield a  
        a, b = b, a+b  
    if a < max:  
        yield a  
        a, b = b, a+b  
    if a < max:  
        yield a  
        a, b = b, a+b  
    ...
```

练习

- 一起写一个斐波那契生成器
- enumerate 函数

内置enumerate函数

`enumerate(iterable, start=0)`

Return an enumerate object. *iterable* must be a sequence, an *iterator*, or some other object which supports iteration. The `__next__()` method of the iterator returned by `enumerate()` returns a tuple containing a count (from *start* which defaults to 0) and the values obtained from iterating over *iterable*.

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

list() : 调用next()直到不能调用为止，并且把返回值存入列表，实质就是列表解析
`[i for i in enumerate(seasons)]`

自己写enumerate函数

```
def enumerate(sequence, start=0):  
    n = start  
    for elem in sequence:  
        yield n, elem  
        n += 1
```

生成器表达式

Generator expression

- When evaluated, returns a generator object

g.next()是Python2.X的写法，对应Python3.X中的next(g)

```
>>> g = (n for n in [1, 2, 3])
>>> for i in g: print i
...
1
2
3
>>> g = (n for n in [1, 2, 3])
>>> g
<generator object <genexpr> at
0x109a4deb0>
>>> g.next()
1
>>> g.next()
2
>>> g.next()
3
>>> g.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

总结

- 有yield的函数：生成器函数
- yield是带暂停功能的return
- for, list 本质上是在调用next(obj)
- 把生成器函数看成事件流，yield暂停，next继续执行
- 生成器函数内部通常包含循环，循环也是事件流
- 把列表解析的中括号换成小括号就是生成器表达式

为什么要用生成器

- 快
- 内存占用小
需要的时候才会产生
- 语义上的含义
一次性使用
- 保存函数的当前执行环境，包括所有局部变量等

2120

votes

16

answers

397k views

The Python yield keyword explained

What is the use of the yield keyword in Python? What does it do? For example, I'm trying to understand this code (**):
`def node._get_child_candidates(self, distance, min_dist, max_dist):` if ...

python

iterator

generator

yield

asked Oct 23 '08 at 22:21



Alex. S.

13.2k ● 7 ● 28 ● 42

<http://stackoverflow.com/questions/231767/the-python-yield-keyword-explained>

Q&A