生成器

廖雪峰
资深软件开发工程师，业余马拉松选手。

通过列表生成式，我们可以直接创建一个列表。但是，受到内存限制，列表容量肯定是有限的。而且，创建一个包含100万个元素的列表，不仅占用很大的存储空间，如果我们仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。

所以，如果列表元素可以按照某种算法推算出来，那我们是否可以在循环的过程中不断推算出后续的元素呢？这样就不必创建完整的list，从而节省大量的空间。在Python中，这种一边循环一边计算的机制，称为生成器：generator。

要创建一个generator，有很多种方法。第一种方法很简单，只要把一个列表生成式的 `[]` 改成 `()` ，就创建了一个generator：

```
>>> L = [x * x for x in range(10)]
>>> L
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> g = (x * x for x in range(10))
>>> g
<generator object <genexpr> at 0x1022ef630>
```

创建 `L` 和 `g` 的区别仅在于最外层的 `[]` 和 `()` ， `L` 是一个list，而 `g` 是一个generator。

我们可以直接打印出list的每一个元素，但我们怎么打印出generator的每一个元素呢？

如果要一个一个打印出来，可以通过 `next()` 函数获得generator的下一个返回值：

```
>>> next(g)
0
>>> next(g)
1
>>> next(g)
4
>>> next(g)
9
>>> next(g)
16
>>> next(g)
25
>>> next(g)
36
>>> next(g)
49
>>> next(g)
64
>>> next(g)
81
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

我们讲过，generator保存的是算法，每次调用 `next(g)` ，就计算出 `g` 的下一个元素的值，直到计算到最后一个元素，没有更多的元素时，抛出 `StopIteration` 的错误。

当然，上面这种不断调用 `next(g)` 实在是太变态了，正确的方法是使用 `for` 循环，因为generator也是可迭代对象：

```
>>> g = (x * x for x in range(10))
>>> for n in g:
...     print(n)
...
0
1
4
9
16
```

```
25
36
49
64
81
```

所以，我们创建了一个generator后，基本上永远不会调用 `next()` ，而是通过 `for` 循环来迭代它，并且不需要关心 `StopIteration` 的错误。

generator非常强大。如果推算的算法比较复杂，用类似列表生成式的 `for` 循环无法实现的时候，还可以用函数来实现。

比如，著名的斐波拉契数列（Fibonacci），除第一个和第二个数外，任意一个数都可由前两个数相加得到：

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

斐波拉契数列用列表生成式写不出来，但是，用函数把它打印出来却很容易：

```
def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        print(b)
        a, b = b, a + b
        n = n + 1
    return 'done'
```

注意，赋值语句：

```
a, b = b, a + b
```

相当于：

```
t = (b, a + b) # t是一个tuple
a = t[0]
b = t[1]
```

但不必显式写出临时变量t就可以赋值。

上面的函数可以输出斐波那契数列的前N个数：


```
1
2
3
5
8
'done'
```

仔细观察，可以看出， `fib` 函数实际上是定义了斐波拉契数列的推算规则，可以从第一个元素开始，推算出后续任意的元素，这种逻辑其实非常类似generator。

也就是说，上面的函数和generator仅一步之遥。要把 `fib` 函数变成generator函数，只需要把 `print(b)` 改为 `yield b` 就可以了：

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

这就是定义generator的另一种方法。如果一个函数定义中包含 `yield` 关键字，那么这个函数就不再是一个普通函数，而是一个generator函数，调用一个generator函数将返回一个generator：

```
>>> f = fib(6)
>>> f
<generator object fib at 0x104feaaa0>
```

这里，最难理解的就是generator函数和普通函数的执行流程不一样。普通函数是顺序执行，遇到 `return` 语句或者最后一行函数语句就返回。而变成generator的函数，在每次调用 `next()` 的时候执行，遇到 `yield` 语句返回，再次执行时从上次返回的 `yield` 语句处继续执行。

举个简单的例子，定义一个generator函数，依次返回数字1，3，5：

```
def odd():
    print('step 1')
    yield 1
    print('step 2')
    yield(3)
    print('step 3')
    yield(5)
```

调用该generator函数时，首先要生成一个generator对象，然后用 `next()` 函数不断获得下一个返回值：

```
>>> o = odd()
>>> next(o)
step 1
1
>>> next(o)
step 2
3
>>> next(o)
step 3
5
>>> next(o)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

可以看到， `odd` 不是普通函数，而是generator函数，在执行过程中，遇到 `yield` 就中断，下次又继续执行。执行3次 `yield` 后，已经没有 `yield` 可以执行了，所以，第4次调用 `next(o)` 就报错。

> ⚠ 注意
>
> 调用generator函数会创建一个generator对象，多次调用generator函数会创建多个相互独立的generator。

有的童鞋会发现这样调用 `next()` 每次都返回1：

```
>>> next(odd())
step 1
1
>>> next(odd())
step 1
1
>>> next(odd())
step 1
1
```

原因在于 `odd()` 会创建一个新的generator对象，上述代码实际上创建了3个完全独立的generator，对3个generator分别调用 `next()` 当然每个都会返回第一个值。

正确的写法是创建一个generator对象，然后不断对这一个generator对象调用 `next()` ：

```
>>> g = odd()
>>> next(g)
step 1
1
>>> next(g)
step 2
3
>>> next(g)
step 3
5
```

回到 `fib` 的例子，我们在循环过程中不断调用 `yield` ，就会不断中断。当然要给循环设置一个条件来退出循环，不然就会产生一个无限数列出来。

同样的，把函数改成generator函数后，我们基本上从来不会用 `next()` 来获取下一个返回值，而是直接使用 `for` 循环来迭代：

```
>>> for n in fib(6):
...     print(n)
...
1
1
2
3
5
8
```

但是用 `for` 循环调用generator时，发现拿不到generator的 `return` 语句的返回值。如果想要拿到返回值，必须捕获 `StopIteration` 错误，返回值包含在 `StopIteration` 的 `value` 中：

```
>>> g = fib(6)
>>> while True:
...     try:
...         x = next(g)
...         print('g:', x)
...     except StopIteration as e:
...         print('Generator return value:', e.value)
...         break
...
g: 1
g: 1
g: 2
g: 3
g: 5
g: 8
Generator return value: done
```

关于如何捕获错误，后面的错误处理还会详细讲解。

练习

杨辉三角定义如下：

```
          1
         / \
        1   1
       / \ / \
      1   2   1
     / \ / \ / \
    1   3   3   1
   / \ / \ / \ / \
  1   4   6   4   1
 / \ / \ / \ / \ / \
1   5   10  10  5   1
```

把每一行看做一个list，试写一个generator，不断输出下一行的list：

```
def triangles():
    pass

# 期待输出:
# [1]
# [1, 1]
# [1, 2, 1]
# [1, 3, 3, 1]
# [1, 4, 6, 4, 1]
# [1, 5, 10, 10, 5, 1]
# [1, 6, 15, 20, 15, 6, 1]
# [1, 7, 21, 35, 35, 21, 7, 1]
# [1, 8, 28, 56, 70, 56, 28, 8, 1]
# [1, 9, 36, 84, 126, 126, 84, 36, 9, 1]
n = 0
results = []
for t in triangles():
    results.append(t)
    n = n + 1
    if n == 10:
        break

for t in results:
    print(t)
```

```
if results == [
    [1],
    [1, 1],
    [1, 2, 1],
    [1, 3, 3, 1],
    [1, 4, 6, 4, 1],
    [1, 5, 10, 10, 5, 1],
    [1, 6, 15, 20, 15, 6, 1],
    [1, 7, 21, 35, 35, 21, 7, 1],
    [1, 8, 28, 56, 70, 56, 28, 8, 1],
    [1, 9, 36, 84, 126, 126, 84, 36, 9, 1]
]:
    print('测试通过!')
else:
    print('测试失败!')
```

参考源码

do_generator.py

小结

generator是非常强大的工具，在Python中，可以简单地把列表生成式改成generator，也可以通过函数实现复杂逻辑的generator。

要理解generator的工作原理，它是在 `for` 循环的过程中不断计算出下一个元素，并在适当的条件结束 `for` 循环。对于函数改成的generator来说，遇到 `return` 语句或者执行到函数体最后一行语句，就是结束generator的指令， `for` 循环随之结束。

请注意区分普通函数和generator函数，普通函数调用直接返回结果：

```
>>> r = abs(6)
>>> r
6
```

generator函数的调用实际返回一个generator对象：

```
>>> g = fib(6)
>>> g
<generator object fib at 0x1022ef948>
```


« 列表生成式

迭代器 »

Comments

Loading comments...