**第二十四章、Python中的生成器**

**第一部分**

**一、列表生成式的生成器**

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：

*#encoding=utf-8*list1=[x \* x **for** x **in** range(10)]  
**print**(type(list1))  
  
list2=(x \* x **for** x **in** range(10))  
**print**(type(list2))

**执行结果：**

**<type 'list'>**

**<type 'generator'>**

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generator的next()方法：

**print** list2.next(),  
**print** list2.next(),  
**print** list2.next(),  
**print** list2.next(),  
**print** list2.next(),  
**print** list2.next(),  
**print** list2.next(),  
**print** list2.next(),  
**print** list2.next(),  
**print** list2.next(),

0 1 4 9 16 25 36 49 64 81

如果list2.next()的个数大于生成器本身的生成数的个数，就会抛出StopIteration异常。

下面的例子说明了这一点：

**try**:  
 **print** list2.next(),  
 **print** list2.next(),  
 **print** list2.next(),  
 **print** list2.next(),  
 **print** list2.next(),  
 **print** list2.next(),  
 **print** list2.next(),  
 **print** list2.next(),  
 **print** list2.next(),  
 **print** list2.next()  
 **print** list2.next()  
**except** StopIteration **as** e :  
 **print 'here is end'**

执行结果：

0 1 4 9 16 25 36 49 64 81

here is end

当然，用上面的方法显得有点“变态”,于是改进如下：

**for t in range(10):  
 print list2.next(),**

但是，最好的方法是使用for循环，因为generator也是可迭代对象，for循环会自动调用迭代对象的next()方法，直到遇到StopIteration。  
**for** l **in** list2:  
 **print**(l),

所以，我们创建了一个generator后，基本上永远不会调用next()方法，而是通过for循环来迭代它。

1. **生成器函数**

**2.1 定义**

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，这就是定义generator的另一种方法。

**def** sum():  
 a=23  
 b=57  
 **return** a+b  
**print** type(sum())  
  
**def** sum2():  
 a = 23  
 b = 57  
 **yield** a + b  
**print**(type(sum2()))

执行结果：

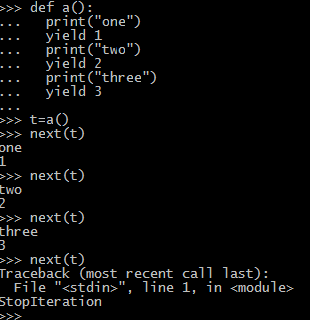
<type 'int'>

<generator object sum2 at 0x00000000021D3240>

<type 'generator'>

2.2 执行流程

generator和函数的执行流程不一样。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。如下例：



需要注意的是，在python2.7中，使用t.next()。

**第二部分**

**2.3、如何生成斐波那契數列**

斐波那契（Fibonacci）數列是一个非常简单的递归数列，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。用计算机程序输出斐波那契數列的前 N 个数是一个非常简单的问题，许多初学者都可以轻易写出如下函数：

**def** fibo(n):  
 a,b,num=0,1,1  
 **while** num<=n:  
 print(b)  
 a,b=b,a+b  
 num=num+1  
 fibo(5)

1

1

2

3

5

结果没有问题，但有经验的开发者会指出，直接在 Fibonacci 函数中用 print 打印数字会导致该函数可复用性较差，因为 Fibonacci 函数返回 None，其他函数无法获得该函数生成的数列。

要提高 Fibonacci 函数的可复用性，最好不要直接打印出数列，而是返回一个 List。

以下是 Fibonacci 函数改写后的第二个版本：

def Fibonacci(n):

a,b,num=0,1,1

list=[];

while num<=n:

a,b=b,a+b

list.append(b)

num+=1

return list

print Fibonacci(5)

改写后的Fibonacci函数通过返回List能满足复用性的要求，但是更有经验的开发者会指出，该函数在运行中占用的内存会随着参数 max 的增大而增大，如果要控制内存占用，最好不要用 List来保存中间结果，而是通过 iterable 对象来迭代，而是在每次迭代中返回下一个数值，内存空间占用很小。利用 iterable 我们可以把 Fibonacci 函数改写为一个支持 iterable 的 class，以下是第三个版本的 Fibonacci：

**import** sys  
**class** Fibo:  
 **def** \_\_init\_\_(self,max):  
 self.max=max  
 self.n, self.a, self.b = 0, 0, 1  
 **def** \_\_iter\_\_(self):  
 **return** self  
 **def** \_\_next\_\_(self):  
 **if** self.n<self.max:  
 r=self.b  
 self.a,self.b=self.b,self.a+self.b  
 self.n=self.n+1  
 **return** r  
 **else**:  
 sys.exit(0)  
t1=Fibo(5)  
print(t1.\_\_next\_\_())  
print(t1.\_\_next\_\_())  
print(t1.\_\_next\_\_())  
print(t1.\_\_next\_\_())  
print(t1.\_\_next\_\_())  
print(**"##################"**)  
t=Fibo(5)  
print(next(t))  
print(next(t))  
print(next(t))  
print(next(t))  
print(next(t))  
print(**"\*\*\*\*\*\*\*\*\*\*\*\*"**)  
**for** n **in** Fibo(5):  
 print(n)

然而，使用 class 改写的这个版本，代码远远没有第一版的 Fibonacci 函数来得简洁。如果我们想要保持第一版 Fibonacci 函数的简洁性，同时又要获得 iterable 的效果，yield 就派上用场了：

def Fibonacci(n):

a,b,num=0,1,1

while num<=n:

a,b=b,a+b

yield b

num+=1

for n in Fibonacci(5):

print n

简单地讲，yield 的作用就是把一个函数变成一个 generator，带有 yield 的函数不再是一个普通函数，Python 解释器会将其视为一个 generator，调用 Fibonacci(5) 不会执行Fibonacci 函数，而是返回一个 iterable 对象！

在 for 循环执行时，每次循环都会执行 Fibonacci 函数内部的代码，执行到 yield b 时，Fibonacci 函数就返回一个迭代值，下次迭代时，代码从 yield b 的下一条语句继续执行，而函数的本地变量看起来和上次中断执行前是完全一样的，于是函数继续执行，直到再次遇到 yield。

yield 的好处是显而易见的，把一个函数改写为一个 generator 就获得了迭代能力，比起用类的实例保存状态来计算下一个 next() 的值，不仅代码简洁，而且执行流程异常清晰。

**第三部分**

如何判断一个函数是否是一个特殊的 generator 函数？可以利用 isgeneratorfunction 判断：

>>>from inspect import isgeneratorfunction

>>> isgeneratorfunction(Fibonacci)

True

要注意区分 Fibonacci 和 Fibonacci(5)，Fibonacci 是一个 generator function，而 Fibonacci(5) 是调用 Fibonacci 返回的一个 generator，好比类的定义和类的实例的区别：

>>>import types

>>> isinstance(Fibonacci, types.GeneratorType)

False

>>> isinstance(Fibonacci(5), types.GeneratorType)

True

Fibonacci 是无法迭代的，而 Fibonacci(5) 是可迭代的：

>>>from collections import Iterable

>>> isinstance(Fibonacci, Iterable)

False

>>> isinstance(Fibonacci(5), Iterable)

True

每次调用 Fibonacci 函数都会生成一个新的 generator 实例，各实例互不影响：

>>>f1 = Fibonacci(3)

>>> f2 = Fibonacci(5)

>>> print 'f1:', f1.next()

f1: 1

>>> print 'f2:', f1.next()

f2: 1

>>> print 'f1:', f1.next()

f1: 2

>>> print 'f2:', f2.next()

f2: 1

>>> print 'f1:', f2.next()

f1: 1

>>> print 'f2:', f2.next()

f2: 2

>>> print 'f2:', f2.next()

f2: 3

>>> print 'f2:', f2.next()

f2: 5

2.3 return 的作用

在一个 generator function 中，如果没有 return，则默认执行至函数完毕，如果在执行过程中 return，则直接抛出 StopIteration 终止迭代。

在一个生成器中，如果没有return，则默认执行到函数完毕时返回StopIteration；

>>> def g1():

... yield 1

...

>>> g=g1()

>>> next(g) #第一次调用next(g)时，会在执行完yield语句后挂起，所以此时程序并没有执行结束。

1

>>> next(g) #程序试图从yield语句的下一条语句开始执行，发现已经到了结尾，所以抛出StopIteration异常。

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

StopIteration

如果遇到return,如果在执行过程中 return，则直接抛出 StopIteration 终止迭代。

>>> def g2():

... yield 'a'

... return

... yield 'b'

...

>>> g=g2()

>>> next(g) #程序停留在执行完yield 'a'语句后的位置。

'a'

>>> next(g) #程序发现下一条语句是return，所以抛出StopIteration异常，这样yield 'b'语句永远也不会执行。

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

StopIteration

如果在return后返回一个值，那么这个值为StopIteration异常的说明，不是程序的返回值。

生成器没有办法使用return来返回值。

**第四部分**

### ****生成器支持的方法****

3.1、close():手动关闭生成器函数，后面的调用会直接返回StopIteration异常。

>>> def g4():

... yield 1

... yield 2

... yield 3

...

>>> g=g4()

>>> next(g)

1

>>> g.close()

>>> next(g) #关闭后，yield 2和yield 3语句将不再起作用

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

StopIteration

3.2、send()

生成器函数最大的特点是可以接受外部传入的一个变量，并根据变量内容计算结果后返回

。这是生成器函数最难理解的地方，也是最重要的地方，实现后面我会讲到的协程就全靠

它了。

def gen():

value=0

while True:

receive=yield value

if receive=='e':

break

value = 'got: %s' % receive

g=gen()

print(g.send(None))

print(g.send('aaa'))

print(g.send(3))

print(g.send('e'))

执行流程：

通过g.send(None)或者next(g)可以启动生成器函数,并执行到第一个yield语句结束的位置。此时，执行完了yield语句，但是没有给receive赋值。yield value会输出初始值0，注意：在启动生成器函数时只能send(None),如果试图输入其它的值都会得到错误提示信息:TypeError: can't send non-None value to a just-started generator。

通过g.send(‘aaa’)，会传入aaa，并赋值给receive，然后计算出value的值，并回到while头部，执行yield value语句有停止。此时yield value会输出”got: aaa”，然后挂起。

通过g.send(3)，会重复第2步，最后输出结果为”got: 3″

当我们g.send(‘e’)时，程序会执行break然后推出循环，最后整个函数执行完毕，所以会得到StopIteration异常。

最后的执行结果如下：

0

got: aaa

got: 3

Traceback (most recent call last):

File "h.py", line 14, in <module>

print(g.send('e'))

StopIteration

3.3、throw()

用来向生成器函数送入一个异常，可以结束系统定义的异常，或者自定义的异常。throw()后直接跑出异常并结束程序，或者消耗掉一个yield，或者在没有下一个yield的时候直接进行到程序的结尾。

def gen():

while True:

try:

yield 'normal value'

yield 'normal value 2'

print('here')

except ValueError:

print('we got ValueError here')

except TypeError:

break

g=gen()

print(next(g))

print(g.throw(ValueError))

print(next(g))

print(g.throw(TypeError))

输出结果为：

normal value

we got ValueError here

normal value

normal value 2

Traceback (most recent call last):

File "h.py", line 15, in <module>

print(g.throw(TypeError))

StopIteration

解释：

print(next(g))：会输出normal value，并停留在yield ‘normal value 2’之前。由于执行了g.throw(ValueError)，所以会跳过所有后续的try语句，也就是说yield ‘normal value 2’不会被执行，然后进入到except语句，打印出we got ValueError here。然后再次进入到while语句部分，消耗一个yield，所以会输出normal value。print(next(g))，会执行yield ‘normal value 2’语句，并停留在执行完该语句后的位置。

g.throw(TypeError)：会跳出try语句，从而print(‘here’)不会被执行，然后执行break语句，跳出while循环，然后到达程序结尾，所以跑出StopIteration异常。

**第五部分**

### ****综合案例****

1、下面给出一个综合例子，用来把一个多维列表展开，或者说扁平化多维列表)

#encoding=utf-8  
def flatten(nested):  
 #如果是List数据类型  
 if type(nested) == list:  
 for index in range(len(nested)):  
 for element in flatten(nested[index]):  
 yield element  
 else:  
 yield nested  
L=[1,[[2,3]],[2,4],[5,[6,[8,[9]]],7]]  
for num in flatten(L):  
 print(num),

1 2 3 2 4 5 6 8 9 7

2、另一个例子

另一个 yield 的例子来源于文件读取。如果直接对文件对象调用 read() 方法，会导致不可预测的内存占用。好的方法是利用固定长度的缓冲区来不断读取文件内容。通过 yield，我们不再需要编写读文件的迭代类，就可以轻松实现文件读取：

*#encoding=utf-8***def** read\_file(fpath):  
 BLOCK\_SIZE = 4096\*2  
 **with** open(fpath, **'rb'**) **as** f:  
 **while True**:  
 block = f.read(BLOCK\_SIZE)  
 **if** block:  
 **yield** block  
 **else**: **return  
  
for** w **in** read\_file(**r"Q:/1.mp4"**):  
 **with** open(**r"Q:/2.mp4"**,**"ab"**) **as** f2:  
 f2.write(w)