

目录

📖 Python教程

Python简介

安装Python

第一个Python程序

Python基础

函数

高级特性

切片

迭代

列表生成式

生成器

迭代器

函数式编程

模块

面向对象编程

面向对象高级编程

错误、调试和测试

IO编程

进程和线程

正则表达式

常用内建模块

常用第三方模块

virtualenv

图形界面

网络编程

电子邮件

访问数据库

Web开发

异步IO

实战

FAQ

期末总结

关于作者

廖雪峰

自己的Python课程

Python商业爬虫全解密

让天下没有爬不到的数据!

Python爬虫 + 数据分析

Python机器学习 + 深度学习

找廖雪峰老师

廖雪峰老师

自己的Java课程

Java高级架构师

更专业 更权威

源码分析专题 + 微服务架构专题

高并发分布式专题 + 性能优化专题

找廖雪峰老师

python免费公开课

编程学习网

授课模式：在线直播+课后视频，从零基础到中高级开发工程师

查看详情

python免费公开课

编程学习网

授课模式：在线直播+课后视频，从零基础到中高级开发工程师

查看详情

生成器

阅读 2131703

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 0x1022a4f30>
```

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)` 实在是太笨了，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，基本上永远都不会调用 `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 = 0, a + b # t是一个tuple
a = t[0]
b = t[1]
```

但不必显式写出临时变量就可以赋值。

上面的函数可以输出斐波那契数列的前n个数：

```
>>> fib(0)
1
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，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：

```
>>> f = fib(0)
>>> f
<generator object fib at 0x104f5a0d0>
```

这里，最难理解的就是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。而生成器generator的函数，在每次调用 `next()` 的时候执行，遇到 `yield` 语句就返回，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，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时，首先会生成一个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，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。

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(0):
...     print(n)
...
1
1
2
3
5
8
```

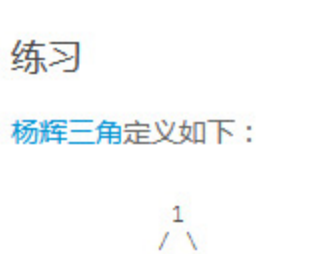
但是用 `for` 循环调用generator时，发现拿不到generator的 `return` 语句的返回值。如果拿到返回值，必须捕获 `StopIteration` 错误，返回值包含在 `StopIteration` 的 `value` 中：

```
>>> g = fib(0)
>>> 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：



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

```
# -*- coding: utf-8 -*-

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():
    print(t)
    results.append(t)
    n = n + 1
    if n == 10:
        break
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!')
```

▶ Run

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(0)
>>> r
0
```

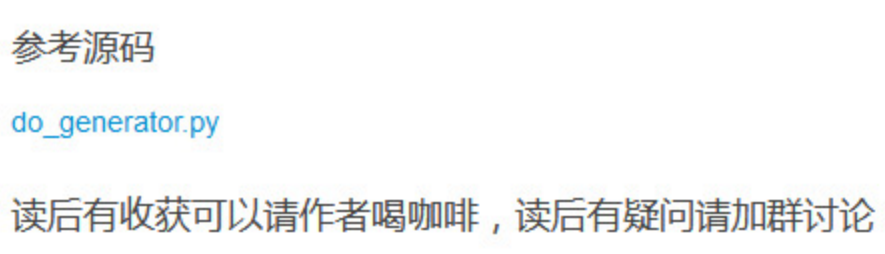
generator函数的“调用”实际返回一个generator对象：

```
>>> g = fib(0)
>>> g
<generator object fib at 0x1022a4f40>
```

参考源码

do_generator.py

读后有收获可以请作者喝咖啡，读后有疑问请加群讨论：



还可以分享给朋友：

分享到微博

← 上一页

下一页 →

廖雪峰

官方+独家

Python商业爬虫全解密

找廖雪峰老师

ACM金牌得主

全球顶尖名企一线数据科学家倾力指导

人工智能与自然语言/计算机视觉课程培训

无offer退全款

廖雪峰推荐

JAVA进阶教程

原价1699元

0元领取

爆款云产品拼购2折起

1核云主机低至199元/年，降低上云门槛

立即查看

E-LEARNING

Python全栈实战课程限时免费领取

120天课程免费领，限时从零基础到项目实战，全套学习内容，数据、分析、Web开发等Python开发技术

腾讯云

学生服务器

体验套餐10元/月

1核2G+1M带宽

50GB存储

立即抢购

评论

我的估计是最菜的了

野望叶生 created at 41 minutes ago, Last updated at 41 minutes ago

```
yield [1]

yield [1, 1]

n = 2

while True:

    m = 2

    L = [1.]

    L.append(n)
```