浅谈Python生成器generator

原创 Zerg_Wang 于 2019-04-09 11:05:52 发布 257 收藏 1 编辑 版权
分类专栏： Python

普通函数在return执行后便会完全退出，其内部的临时变量也会被销毁，然而在 Python 中，生成器函数允许自身“暂时”退出，在下一次调用它自己的时候，会从上次退出之处接着执行相应语句。而要实现这一功能，需要用到yield关键字。一旦函数中使用了yield，该函数则自动变为生成器函数。

生成器函数是一类特殊的迭代器，实际上它与普通函数区别并不大，以下将介绍生成器的特殊使用及其方法：

生成器的使用

yield

yield相当于return，后面可接多种类型的返回值。生成器函数在执行到yield后会暂时退出，回到调用它的函数，然而生成器函数内的临时变量、执行程度都会保留，下次调用该函数时，会从yield的下一句开始继续执行，直到生成器函数的最后。

生成器函数可以有多个yield，也可以将yield写入循环反复执行。

举个例子：（右侧序号为程序执行顺序）

```
def example():
    print('step 1')        3
    yield 1                4
    print('step 2')        7
    yield 2                8
    print('step 3')        11
    yield 3                12


gen = example()            1
next(gen)                  2
print('step 1 finish')     5
next(gen)                  6
print('step 1 finish')     9
next(gen)                  10
print('step 1 finish')     13
```


输出为：

```
step 1
step 1 finish
step 2
step 1 finish
step 3
step 1 finish

Process finished with exit code 0
```

next

举个例子：（输出斐波那契数列的前100项）

```
1  def fo():
2      a, b = 0, 1
3      while True:
4          yield b
5          a, b = b, a+b
6
7  gen = fo()
8  for i in range(100):
9      print(next(gen))
```

生成器函数通过yield返回，但函数本身的返回值并非yield后所接的值，而是其生成器，因此不可直接print(gen)或者print(fo())。

要得到yield的结果，需要用到next()来进行迭代，而使用next，则先要通过gen = fo()这一步，这一步是使用生成器函数fo()生成了一个生成器gen（相当于实例化一个对象），因为生成器自带有next方法，可以用next()进行迭代（除了next(gen)，也可以写成gen.__next__()）。

实际上，生成器函数fo()也可以调用next方法，但它本身并无生成器迭代的功能，直接print(next(fo()))的结果就是，生成器函数退化为普通函数，yield成为了return，100次循环就会输出100个1。

除了直接使用next方法，还可以通过for循环调用，这是因为for循环内含next方法。此外，如果生成器函数所有的yield执行完毕，或者执行到return语句时，再使用next会报错：StopIteration，而for循环会自动忽略这一错误。

输出100以内的斐波那契数列：（与上面的功能略有不同）

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

for i in fo(100):
    print(i)
```

如果要捕获生成器函数的return值，或者防止StopIteration错误中断程序，我们可以：

```
def fo():
    yield 1
    yield 2
    return 3

gen = fo()
while True:
    try:
        print(next(gen))
    except StopIteration as e:
        print(e)
        break
```

若没有return或return后不带值，e的值为空。

Send

通过send函数，可以实现往生成器函数中传值的功能：

```
def fo():
    while True:
        t = yield '在这停顿'
        print(t)

gen = fo()
print(next(gen))
print(next(gen))
print(gen.send(5))
```

输出为：

```
在这停顿
None
在这停顿
5
在这停顿
```

第一次执行输出t，发现结果为None，这是因为yield后的值返回出后被next方法捕获，也就没有任何值被赋给t，之后在gen.send(5)处，send方法本身也有next的功能，回到上次中断处，将传入的5赋值给了t（因此传入的值可以与yield值的类型不同）。可以理解为，要给生成器函数内某变量传入值，应该在函数执行的相应阶段通过send传入，接收方必须使用t = yield ...的形式接收。

close

关闭生成器，之后若再用next或其他方法调用，无论有没有yield都会直接抛出StopIteration错误。

```
def fo():
    yield 1
    yield 2

gen = fo()
print(next(gen))
gen.close()
print(next(gen))

```

最后执行next(gen)，虽然之前才执行到yield 1，但因为close了生成器，这里就无法输出2，直接报错：

```
1
Traceback (most recent call last):
  File "C:/Users/Zerg Wang/Desktop/temporal.py", line 8, in <module>
    print(next(gen))
StopIteration
```

生成器作用

模拟并发

在函数A执行时，可用yield将其挂起，再运行函数B，然后再用next唤醒A继续执行。用这种方法可以非常简便地模拟并发，不必引入复杂繁琐的多线程代码。（然而，这仅仅是模拟，要实现真正的多线程还需走正道……）

节省空间

生成器保存的是算法，只有在next()调用时，才会开始执行，直到下一个yield。

假如说有某个函数，现在传入一个极大的数据规模，大到现有内存完全不够用，这个时候可以将这个函数改写成生成器函数，分批处理，分批返回结果，则可解决上述问题。

例如在文件读写时，为了防止f.read()时无法预知的内存使用，可设定每次读入的数据量，分批读入，代码也比较简单，易于实现。

```
def read_file(path):
    size = 1024
    with open(path, 'rb') as f:
        while True:
            block = f.read(size)
            if block:
                yield block
            else:
                return
```

简化代码

在实现上述功能的同时，我们可以发现代码都较为简洁，这同样也是生成器的一大特点。这里再举一个例子：

```
fo = (x*x for x in range(100))
for i in fo:
    print(i)
```

得到前100个平方数，写起来极为简单。（当然也可以用列表实现，把fo后的小括号换成方括号就行）

小结

生成器使用简便，功能强大，前文已经介绍过了。这里讲讲我的一些想法。对于节省空间这一点，我认为生成器其实是一种”妥协“，以时间换空间，同样是生成斐波那契数列，生成器固然方便、迅速，但无法存储数列的每一项（否则就和列表一样了），在小数据量的情况下对于多次查询任务，效率会较低（每次都要从头算起，除非将查询排序）。当然，在超大数据量下（爆内存那种）生成器还是有绝对优势的。

PS：若是极为简单的任务，例如上面输出前100个平方数的，其实直接使用for循环会更快，而且更简单：

```
for i in range(100):
    print(i)
```

参考资料

http://www.runoob.com/w3cnote/python-yield-used-analysis.html

http://3g.163.com/dy/article/DCVHKB5E0516U4OP.html

http://python.jobbole.com/87613/