



链滴

Python 关键字 yield 详解以及 Iterable 和 Iterator 区别

作者: [hiquanta](#)

原文链接: <https://ld246.com/article/1509073585305>

来源网站: [链滴](#)

许可协议: [署名-相同方式共享 4.0 国际 \(CC BY-SA 4.0\)](#)

<p>迭代器(Iterator)</p>

<p>为了理解 yield 是什么，首先要明白生成器(generator)是什么，在讲生成器之前先说说迭代器(iterator)，当创建一个列表(list)时，你可以逐个的读取每一项，这就叫做迭代 (iteration) 。</p>

mylist = [1 , 2 , 3]

for i in mylist :

print (i)

1

2

3

<p>Mylist 就是一个迭代器，不管是使用复杂的表达式列表，还是直接创建一个列表，都是可迭代的象。</p>

mylist = [x*x for x in range(3)]

for i in mylist :

print (i)

0

1

4

<p>你可以使用 “for… in …” 来操作可迭代对象，如：list,string,files,这些迭代对象非常方便我们使用，因为你可以按照你的意愿进行重复的读取。但是你不得不预先存储所有的元素在内存中，那些对象有很多元素时，并不是每一项都对你有用。</p>

<p>生成器(Generators)</p>

<p>生成器同样是可迭代对象，但是你能只能读取一次，因为它并没有把所有值存放内存中，它动态的成值：</p>

mygenerator = (x*x for x in range(3))

for i in mygenerator :

print (i)

0

1

4

<p>使用()和[]结果是一样的，但是，第二次执行 “ for in mygenerator” 不会有任何结果返回，因为它只能使用一次。首先计算 0，然后计算 1，之后计算 4，依次类推。</p>

<p>Yield</p>

<p>**Yield 是关键字，用起来像 return，yield 在告诉程序，要求函数返回一个生成 **器。</p>

<p>def createGenerator() :</p>

<p>mylist = range(3)</p>

<p>for i in mylist :</p>

<p>yield i*i</p>

<p>mygenerator = createGenerator() # create a generator</p>

```

</li>
<li>
<p>print (mygenerator) # mygenerator is an object!</p>
</li>
<li>
<p>&lt; **generator object **createGenerator at 0xb7555c34 &gt;</p>
</li>
<li>
<p>for i in mygenerator:</p>
</li>
<li>
<p>print (i)</p>
</li>
<li>
<p>0</p>
</li>
<li>
<p>1</p>
</li>
<li>
<p>4</p>
</li>
</ol>

```

<p>这个示例本身没什么意义，但是它很清晰地说明函数将返回一组仅能读一次的值，要想掌握 yield，首先必须理解的是：当你调用生成器函数的时候，如上例中的 createGenerator()，程序并不会执行函数体内的代码，它仅仅只是返回生成器对象，这种方式颇为微妙。函数体内的代码只有直到每次循环迭代(for)生成器的时候才会运行。</p>

<p>函数第一次运行时，它会从函数开始处直到碰到 yield 时，就返回循环的第一个值，然后，交互运行、返回，直到没有值返回为止。如果函数在运行但是并没有遇到 yield，就认为该生成器是空，因可能是循环终止，或者没有满足任何“if/else”。</p>

<p>接下来读一小段代码来理解生成器的优点：</p>

<p>控制生成器穷举</p>

```

<ol>
<li>
<blockquote>
<blockquote>
<blockquote>
<p>class Bank(): # 创建银行，构造 ATM 机</p>
</blockquote>
</blockquote>
</blockquote>
</li>
<li>...    crisis = False</li>
<li>...    def create_atm( self ):</li>
<li>...        while not self.crisis:</li>
<li>...            yield "$100"</li>
<li>
<blockquote>
<blockquote>
<blockquote>
<p>hsbc = Bank() # 没有危机时，你想要多少，ATM 就可以吐多少</p>
</blockquote>
</blockquote>
</blockquote>

```

```

</li>
<li>
<blockquote>
<blockquote>
<blockquote>
<p>corner_street_atm = hsbc.create_atm()</p>
</blockquote>
</blockquote>
</blockquote>
</li>
<li>
<blockquote>
<blockquote>
<blockquote>
<p>print (corner_street_atm.next())</p>
</blockquote>
</blockquote>
</blockquote>
</li>
<li>$ 100</li>
<li>
<blockquote>
<blockquote>
<blockquote>
<p>print (corner_street_atm.next())</p>
</blockquote>
</blockquote>
</blockquote>
</li>
<li>$ 100</li>
<li>
<blockquote>
<blockquote>
<blockquote>
<p>print ([corner_street_atm.next() for cash in range( 5 )])</p>
</blockquote>
</blockquote>
</blockquote>
</li>
<li>[ '$100', '$100', '$100', '$100', '$100' ]</li>
<li>
<blockquote>
<blockquote>
<blockquote>
<p>hsbc.crisis = True # 危机来临, 银行没钱了</p>
</blockquote>
</blockquote>
</blockquote>
</li>
<li>
<blockquote>
<blockquote>
<blockquote>
<p>print (corner_street_atm.next())</p>

```

```

</blockquote>
</blockquote>
</blockquote>
</li>
<li> </li>
<li>
<blockquote>
<blockquote>
<blockquote>
<p>wall_street_atm = hsbc.ceate_atm() # 新建 ATM, 银行仍然没钱</p>
</blockquote>
</blockquote>
</blockquote>
</li>
<li>
<blockquote>
<blockquote>
<blockquote>
<p>print (wall_street_atm.next())</p>
</blockquote>
</blockquote>
</blockquote>
</li>
<li> </li>
<li>
<blockquote>
<blockquote>
<blockquote>
<p>hsbc.crisis = False # 麻烦就是, 即使危机过后银行还是空的</p>
</blockquote>
</blockquote>
</blockquote>
</li>
<li>
<blockquote>
<blockquote>
<blockquote>
<p>print (corner_street_atm.next())</p>
</blockquote>
</blockquote>
</blockquote>
</li>
<li> </li>
<li>
<blockquote>
<blockquote>
<blockquote>
<p>brand_new_atm = hsbc.create_atm() # 构造新的 ATM, 恢复业务</p>
</blockquote>
</blockquote>
</blockquote>
</li>
<li>
<blockquote>

```

```

<blockquote>
<blockquote>
<p>for cash in brand_new_atm :</p>
</blockquote>
</blockquote>
</blockquote>
</li>
<li>...    print cash</li>
<li>$ 100</li>
</ol>
<p>对于访问控制资源，生成器显得非常有用。</p>
<p><strong>迭代工具，你最好的朋友</strong></p>
<p>**迭代工具模块包含了操做指定的函数用于操作迭代器。 **想复制一个迭代器出来？链接两个迭
器？以 one liner (这里的 one-liner 只需一行代码能搞定的任务)用内嵌的列表组合一组值？不使用 li
t 创建 Map/Zip? ...，你要做的就是 import itertools, 举个例子吧：</p>
<p>四匹马赛跑到达终点排名的所有可能性：</p>
<ol>
<li>
<blockquote>
<blockquote>
<blockquote>
<p>horses = [ 1 , 2 , 3 , 4 ]</p>
</blockquote>
</blockquote>
</blockquote>
</li>
<li>
<blockquote>
<blockquote>
<blockquote>
<p>races = itertools.permutations(horses)</p>
</blockquote>
</blockquote>
</blockquote>
</li>
<li>
<blockquote>
<blockquote>
<blockquote>
<p>print (races)</p>
</blockquote>
</blockquote>
</blockquote>
</li>
</ol>

```

```
<blockquote>
<blockquote>
<blockquote>
<p>print (list(itertools.permutations(horses)))</p>
</blockquote>
</blockquote>
</blockquote>
```

```
</li>
<li>[( 1 , 2 , 3 , 4 ),</li>
<li>( 1 , 2 , 4 , 3 ),</li>
<li>( 1 , 3 , 2 , 4 ),</li>
<li>( 1 , 3 , 4 , 2 ),</li>
<li>( 1 , 4 , 2 , 3 ),</li>
<li>( 1 , 4 , 3 , 2 ),</li>
<li>( 2 , 1 , 3 , 4 ),</li>
<li>( 2 , 1 , 4 , 3 ),</li>
<li>( 2 , 3 , 1 , 4 ),</li>
<li>( 2 , 3 , 4 , 1 ),</li>
<li>( 2 , 4 , 1 , 3 ),</li>
<li>( 2 , 4 , 3 , 1 ),</li>
<li>( 3 , 1 , 2 , 4 ),</li>
<li>( 3 , 1 , 4 , 2 ),</li>
<li>( 3 , 2 , 1 , 4 ),</li>
<li>( 3 , 2 , 4 , 1 ),</li>
<li>( 3 , 4 , 1 , 2 ),</li>
<li>( 3 , 4 , 2 , 1 ),</li>
<li>( 4 , 1 , 2 , 3 ),</li>
<li>( 4 , 1 , 3 , 2 ),</li>
<li>( 4 , 2 , 1 , 3 ),</li>
<li>( 4 , 2 , 3 , 1 ),</li>
<li>( 4 , 3 , 1 , 2 ),</li>
<li>( 4 , 3 , 2 , 1 )]</li>
</ol>
```

<p>理解迭代的内部机制:</p>

<p>迭代(iteration)就是对可迭代对象(iterables, 实现了__iter__()方法)和迭代器(iterators, 实现了__next__()方法)的一个操作过程。可迭代对象是任何可返回一个迭代器的对象, 迭代器是应用迭代对象中迭代的对象, 换一种方式说的话就是: iterable对象的__iter__()方法可以返回 iterator 对象, iterator 通过调用 next()方法获取其中的每一个值(译者注), 读者可以结合 Java API 中的 Iterable 接口和 Iterator 接口进行类比。</p>

<p></p>

<p>java Iterable 接口:</p>

<p>public interface Iterable</p>

<p>Implementing this interface allows an object to be the target of the "foreach" statement.</p>

<p>方法:</p>

<p>Iterator<T>iterator()</p>

<p>Returns an iterator over a set of elements of type T.</p>

<p>Returns:</p>

<p>an Iterator.</p>

Iterator 接口: </p>

public interface Iterator </p>

An iterator over a collection. Iterator takes the place of Enumeration in the Java collections framework. Iterators differ from enumerations in two ways:</p>

Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.

Method names have been improved.

This interface is a member of the Java Collections Framework .</p>

boolean hasNext()

Returns true if the iteration has more elements.

E next()

Returns the next element in the iteration.

void remove()

Removes from the underlying collection the last element returned by the iterator (optional operation).</p>

<p>为什么一定要去实现 Iterable 这个接口呢? 为什么不直接实现 Iterator 接口呢? </p>

<p>看一下 JDK 中的集合类, 比如 List 一族或者 Set 一族,

都是实现了 Iterable 接口, 但并不直接实现 Iterator 接口。

仔细想一下这么做是有道理的。 **因为 Iterator 接口的核心方法 next()或者 hasNext()
是依赖于迭代器的当前迭代位置的。 **

如果 Collection 直接实现 Iterator 接口, 势必导致集合对象中包含当前迭代位置的数据(指针)。
当集合在不同方法间被传递时, 由于当前迭代位置不可预置, 那么 next()方法的结果会变成不可预知

除非再为 Iterator 接口添加一个 reset()方法, 用来重置当前迭代位置。

但即时这样, Collection 也只能同时存在一个当前迭代位置。

而 Iterable 则不然, 每次调用都会返回一个从头开始计数的迭代器。

多个迭代器是互不干扰的

来源: http://blog.csdn.net/23199249 </p>