

## 19 | 深入理解迭代器和生成器

2019-06-21 景霄

Python核心技术与实战

[进入课程 >](#)



讲述：冯永吉

时长 10:34 大小 8.48M



你好，我是景霄。

在第一次接触 Python 的时候，你可能写过类似 `for i in [2, 3, 5, 7, 11, 13]: print(i)` 这样的语句。`for in` 语句理解起来很直观形象，比起 C++ 和 java 早期的

```
for (int i = 0; i < n; i ++) printf("%d\n",  
a[i])
```

 这样的语句，不知道简洁清晰到哪里去了。

但是，你想过 Python 在处理 for in 语句的时候，具体发生了什么吗？什么样的对象可以被 for in 来枚举呢？

这一节课，我们深入到 Python 的容器类型实现底层去走走，了解一种叫做迭代器和生成器的东西。

## 你肯定用过的容器、可迭代对象和迭代器

容器这个概念非常好理解。我们说过，在 Python 中一切皆对象，对象的抽象就是类，而对象的集合就是容器。

列表 (list: [0, 1, 2])，元组 (tuple: (0, 1, 2))，字典 (dict: {0:0, 1:1, 2:2})，集合 (set: set([0, 1, 2])) 都是容器。对于容器，你可以很直观地想象成多个元素在一起的单元；而不同容器的区别，正是在于内部数据结构的实现方法。然后，你就可以针对不同场景，选择不同时间和空间复杂度的容器。


所有的容器都是可迭代的 (iterable)。这里的迭代，和枚举不完全一样。迭代可以想象成是你去买苹果，卖家并不告诉你他有多少库存。这样，每次你都需要告诉卖家，你要一

个苹果，然后卖家采取行为：要么给你拿一个苹果；要么告诉你，苹果已经卖完了。你并不需要知道，卖家在仓库是怎么摆放苹果的。

严谨地说，迭代器 (iterator) 提供了一个 next 的方法。调用这个方法后，你要么得到这个容器的下一个对象，要么得到一个 StopIteration 的错误（苹果卖完了）。你不需要像列表一样指定元素的索引，因为字典和集合这样的容器并没有索引一说。比如，字典采用哈希表实现，那么你就只需要知道，next 函数可以不重复不遗漏地一个一个拿到所有元素即可。

而可迭代对象，通过 iter() 函数返回一个迭代器，再通过 next() 函数就可以实现遍历。for in 语句将这个过程隐式化，所以，你只需要知道它大概做了什么就行了。

我们来看下面这段代码，主要向你展示怎么判断一个对象是否可迭代。当然，这还有另一种做法，是 isinstance(obj, Iterable)。

 复制代码

```
1 def is_iterable(param):
2     try:
3         iter(param)
4         return True
```

```
5     except TypeError:
6         return False
7
8  params = [
9      1234,
10     '1234',
11     [1, 2, 3, 4],
12     set([1, 2, 3, 4]),
13     {1:1, 2:2, 3:3, 4:4},
14     (1, 2, 3, 4)
15 ]
16
17 for param in params:
18     print('{} is iterable? {}'.format(param, is_iterabl
19
20 ##### 输出 #####
21
22 1234 is iterable? False
23 1234 is iterable? True
24 [1, 2, 3, 4] is iterable? True
25 {1, 2, 3, 4} is iterable? True
26 {1: 1, 2: 2, 3: 3, 4: 4} is iterable? True
27 (1, 2, 3, 4) is iterable? True
```


通过这段代码，你就可以知道，给出的类型中，除了数字1234 之外，其它的数据类型都是可迭代的。

## 生成器，又是什么？

据我所知，很多人对生成器这个概念会比较陌生，因为生成器在很多常用语言中，并没有相对应的模型。


这里，你只需要记着一点：**生成器是懒人版本的迭代器**。

我们知道，在迭代器中，如果我们想要枚举它的元素，这些元素需要事先生成。这里，我们先来看下面这个简单的样例。

 复制代码

```
1 import os
2 import psutil
3
4 # 显示当前 python 程序占用的内存大小
5 def show_memory_info(hint):
6     pid = os.getpid()
7     p = psutil.Process(pid)
8
9     info = p.memory_full_info()
10    memory = info.uss / 1024. / 1024
11    print('{} memory used: {} MB'.format(hint, memory))
```



 复制代码

```
1 def test_iterator():
2     show_memory_info('initing iterator')
3     list_1 = [i for i in range(100000000)]
4     show_memory_info('after iterator initiated')
```

```

5     print(sum(list_1))
6     show_memory_info('after sum called')
7
8 def test_generator():
9     show_memory_info('initing generator')
10    list_2 = (i for i in range(100000000))
11    show_memory_info('after generator initiated')
12    print(sum(list_2))
13    show_memory_info('after sum called')
14
15 %time test_iterator()
16 %time test_generator()
17
18 ##### 输出 #####
19
20 initing iterator memory used: 48.9765625 MB
21 after iterator initiated memory used: 3920.30078125 MB
22 4999999950000000
23 after sum called memory used: 3920.3046875 MB
24 Wall time: 17 s
25 initing generator memory used: 50.359375 MB
26 after generator initiated memory used: 50.359375 MB
27 4999999950000000
28 after sum called memory used: 50.109375 MB
29 Wall time: 12.5 s

```



声明一个迭代器很简单，`[i for i in range(100000000)]`就可以生成一个包含一亿元素的列表。每个元素在生成后都会保存到内存中，你通过代码可以

看到，它们占用了巨量的内存，内存不够的话就会出现 OOM 错误。

不过，我们并不需要在内存中同时保存这么多东西，比如对元素求和，我们只需要知道每个元素在相加的那一刻是多少就行了，用完就可以扔掉了。

于是，生成器的概念应运而生，在你调用 `next()` 函数的时候，才会生成下一个变量。生成器在 Python 的写法是用小括号括起来，`(i for i in range(100000000))`，即初始化了一个生成器。


这样一来，你可以清晰地看到，生成器并不会像迭代器一样占用大量内存，只有在被使用的时候才会调用。而且生成器在初始化的时候，并不需要运行一次生成操作，相比于 `test_iterator()`，`test_generator()` 函数节省了一次生成一亿个元素的过程，因此耗时明显比迭代器短。

到这里，你可能说，生成器不过如此嘛，我有的是钱，不就是多占一些内存和计算资源嘛，我多出点钱就是了呗。

哪怕你是土豪，请坐下先喝点茶，再听我继续讲完，这次，我们来实现一个自定义的生成器。

## 生成器，还能玩什么花样？

数学中有一个恒等式， $(1 + 2 + 3 + \dots + n)^2 = 1^3 + 2^3 + 3^3 + \dots + n^3$ ，想必你高中就应该学过它。现在，我们来验证一下这个公式的正确性。老规矩，先放代码，你先自己阅读一下，看不懂的也不要紧，接下来我再来详细讲解。

 复制代码

```
1 def generator(k):
2     i = 1
3     while True:
4         yield i ** k
5         i += 1
6
7 gen_1 = generator(1)
8 gen_3 = generator(3)
9 print(gen_1)
10 print(gen_3)
11
12 def get_sum(n):
13     sum_1, sum_3 = 0, 0
14     for i in range(n):
15         next_1 = next(gen_1)
16         next_3 = next(gen_3)
17         print('next_1 = {}, next_3 = {}'.format(next_1,
18             sum_1 += next_1
19             sum_3 += next_3
20         print(sum_1 * sum_1, sum_3)
21
22 get_sum(8)
```



```
23
24 ##### 输出 #####
25
26 <generator object generator at 0x000001E70651C4F8>
27 <generator object generator at 0x000001E70651C390>
28 next_1 = 1, next_3 = 1
29 next_1 = 2, next_3 = 8
30 next_1 = 3, next_3 = 27
31 next_1 = 4, next_3 = 64
32 next_1 = 5, next_3 = 125
33 next_1 = 6, next_3 = 216
34 next_1 = 7, next_3 = 343
35 next_1 = 8, next_3 = 512
36 1296 1296
```



这段代码中，你首先注意一下 `generator()` 这个函数，它返回了一个生成器。

接下来的 `yield` 是魔术的关键。对于初学者来说，你可以理解为，函数运行到这一行的时候，程序会从这里暂停，然后跳出，不过跳到哪里呢？答案是 `next()` 函数。那么 `i ** k` 是干什么的呢？它其实成了 `next()` 函数的返回值。


这样，每次 `next(gen)` 函数被调用的时候，暂停的程序就又复活了，从 `yield` 这里向下继续执行；同时注意，局部变量 `i` 并没有被清除掉，而是会继续累加。我们可以看到 `next_1` 从 1 变到 8，`next_3` 从 1 变到 512。

聪明的你应该注意到了，这个生成器居然可以一直进行下去！没错，事实上，迭代器是一个有限集合，生成器则可以成为一个无限集。我只管调用 `next()`，生成器根据运算会自动生成新的元素，然后返回给你，非常便捷。

到这里，土豪同志应该也坐不住了吧，那么，还能再给力一点吗？

别急，我们再来看一个问题：给定一个 `list` 和一个指定数字，求这个数字在 `list` 中的位置。


下面这段代码你应该不陌生，也就是常规做法，枚举每个元素和它的 `index`，判断后加入 `result`，最后返回。

 复制代码

```
1 def index_normal(L, target):
2     result = []
3     for i, num in enumerate(L):
4         if num == target:
5             result.append(i)
6     return result
7
8 print(index_normal([1, 6, 2, 4, 5, 2, 8, 6, 3, 2], 2))
9
10 ##### 输出 #####
11
12 [2, 5, 9]
```

---

那么使用迭代器可以怎么做呢？二话不说，先看代码。

 复制代码

```
1 def index_generator(L, target):
2     for i, num in enumerate(L):
3         if num == target:
4             yield i
5
6 print(list(index_generator([1, 6, 2, 4, 5, 2, 8, 6, 3,
7
8 ##### 输出 #####
9
10 [2, 5, 9]
```

聪明的你应该看到了明显的区别，我就不做过多解释了。唯一需要强调的是，`index_generator` 会返回一个 `Generator` 对象，需要使用 `list` 转换为列表后，才能用 `print` 输出。

这里我再多说两句。在 Python 语言规范中，用更少、更清晰的代码实现相同功能，一直是被推崇的做法，因为这样能够很有效提高代码的可读性，减少出错概率，也方便别人快速准确理解你的意图。当然，要注意，这里“更少”的前提

是清晰，而不是使用更多的魔术操作，虽说减少了代码却反而增加了阅读的难度。


回归正题。接下来我们再来看一个问题：给定两个序列，判定第一个是不是第二个的子序列。（LeetCode 链接如下：<https://leetcode.com/problems/is-subsequence/>）

先来解读一下这个问题本身。序列就是列表，子序列则指的是，一个列表的元素在第二个列表中都按顺序出现，但是并不必挨在一起。举个例子，`[1, 3, 5]` 是 `[1, 2, 3, 4, 5]` 的子序列，`[1, 4, 3]` 则不是。

要解决这个问题，常规算法是贪心算法。我们维护两个指针指向两个列表的最开始，然后对第二个序列一路扫过去，如果某个数字和第一个指针指的一样，那么就把第一个指针前进一步。第一个指针移出第一个序列最后一个元素的时候，返回 `True`，否则返回 `False`。

不过，这个算法正常写的话，写下来怎么也得十行左右。

那么如果我们用迭代器和生成器呢？


 复制代码

```
1 def is_subsequence(a, b):
```

```
2     b = iter(b)
3     return all(i in b for i in a)
4
5 print(is_subsequence([1, 3, 5], [1, 2, 3, 4, 5]))
6 print(is_subsequence([1, 4, 3], [1, 2, 3, 4, 5]))
7
8 ##### 输出 #####
9
10 True
11 False
```

这简短的几行代码，你是不是看得一头雾水，不知道发生了什么？

来，我们先把这段代码复杂化，然后一步步看。

 复制代码


```
1 def is_subsequence(a, b):
2     b = iter(b)
3     print(b)
4
5     gen = (i for i in a)
6     print(gen)
7
8     for i in gen:
9         print(i)
10
11     gen = ((i in b) for i in a)
12     print(gen)
```

```
13
14     for i in gen:
15         print(i)
16
17     return all(((i in b) for i in a))
18
19 print(is_subsequence([1, 3, 5], [1, 2, 3, 4, 5]))
20 print(is_subsequence([1, 4, 3], [1, 2, 3, 4, 5]))
21
22 ##### 输出 #####
23
24 <list_iterator object at 0x000001E7063D0E80>
25 <generator object is_subsequence.<locals>.<genexpr> at
26 1
27 3
28 5
29 <generator object is_subsequence.<locals>.<genexpr> at
30 True
31 True
32 True
33 False
34 <list_iterator object at 0x000001E7063D0D30>
35 <generator object is_subsequence.<locals>.<genexpr> at
36 1
37 4
38 3
39 <generator object is_subsequence.<locals>.<genexpr> at
40 True
41 True
42 False
43 False
```

首先，第二行的 `b = iter(b)`，把列表 `b` 转化成了一个迭代器，这里我先不解释为什么要这么做。


接下来的 `gen = (i for i in a)` 语句很好理解，产生一个生成器，这个生成器可以遍历对象 `a`，因此能够输出 1, 3, 5。而 `(i in b)` 需要好好揣摩，这里你是不是能联想到 `for in` 语句？

没错，这里的 `(i in b)`，大致等价于下面这段代码：

 复制代码

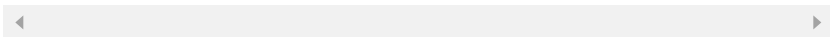
```
1 while True:
2     val = next(b)
3     if val == i:
4         yield True
```

这里非常巧妙地利用生成器的特性，`next()` 函数运行的时候，保存了当前的指针。比如再看下面这个示例：

 复制代码

```
1 b = (i for i in range(5))
2
3 print(2 in b)
4 print(4 in b)
5 print(3 in b)
```

```
6
7 ##### 输出 #####
8
9 True
10 True
11 False
```



至于最后的 `all()` 函数，就很简单了。它用来判断一个迭代器的元素是否全部为 `True`，如果是则返回 `True`，否则就返回 `False`。

于是到此，我们就很优雅地解决了这道面试题。不过你一定要注意，面试的时候尽量不要用这种技巧，因为你的面试官有可能并不知道生成器的用法，他们也没有看过我的极客时间专栏。不过，在这个技术知识点上，在实际工作的应用上，你已经比很多人更加熟练了。继续加油！

## 总结

总结一下，今天我们讲了四种不同的对象，分别是容器、可迭代对象、迭代器和生成器。

容器是可迭代对象，可迭代对象调用 `iter()` 函数，可以得到一个迭代器。迭代器可以通过 `next()` 函数来得到下一个元素，从而支持遍历。



生成器是一种特殊的迭代器（注意这个逻辑关系反之不成立）。使用生成器，你可以写出来更加清晰的代码；合理使用生成器，可以降低内存占用、优化程序结构、提高程序速度。

生成器在 Python 2 的版本上，是协程的一种重要实现方式；而 Python 3.5 引入 `async await` 语法糖后，生成器实现协程的方式就已经落后了。我们会在下节课，继续深入讲解 Python 协程。

## 思考题

最后给你留一个思考题。对于一个有限元素的生成器，如果迭代完成后，继续调用 `next()`，会发生什么呢？生成器可以遍历多次吗？

欢迎留言和我讨论，也欢迎你把这篇文章分享给你的同事、朋友，一起在交流中进步。

---

# Python 核心技术与实战

系统提升你的 Python 能力

景霄

Facebook 资深工程师



新版升级：点击「👤 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 18 | [名师分享] metaclass，是潘多拉魔盒还是阿拉丁...

下一篇 20 | 揭秘 Python 协程

## 精选留言 (39)

写留言

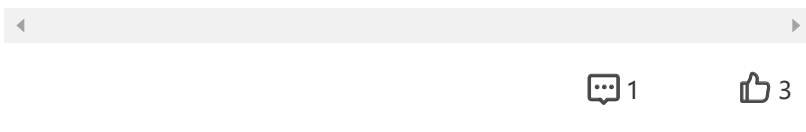


John Si 置顶

2019-06-21

我不知道如何把这技巧运用在编程中，老师能否举几个例子来说明一下呢？谢谢

作者回复: 例子已经在文中举了不少, 对于如何娴熟地在编程中运用, 这个需要长时间的积累, 从阅读别人高质量的源代码, 自己主动有意识地在自己的项目中思考, 最后才会形成质变, 内化成自己的能力, 从而清楚地知道哪里应该用高级语法, 高级工具, 哪里应该简单的一笔带过。Python 的生成器无疑是最有用的特性, 但也是最不广泛被使用的特性, 这一章的目的, 能够让你对生成器有基本的了解, 下次在代码中遇到, 能够说, “这个我知道, 这个我懂!” 便已足够。加油!

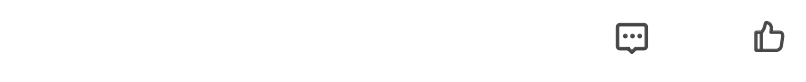


**Jingxiao** 置顶

2019-06-23

思考题答案:

很多同学的回复非常正确, 生成器只能遍历一次, 继续调用 next() 会 raise StopIteration。只有复位生成器才能重新进行遍历。



**Geek\_b04e76**

2019-06-21

上一篇的分享mateclass写得看不懂, 老师可否重新通俗写一下, 分享嘉宾的风格跟老师不太一样啊



16

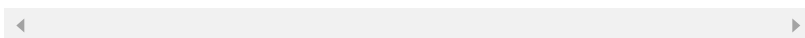


**farFlight**

2019-06-21

迭代完成后，继续调用 `next()` 会出现 `StopIteration`。  
生成器只能遍历一次，但是可以重新调用重新遍历。

作者回复: 正确



5



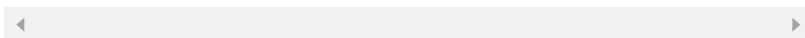
**SCAR**

2019-06-21

思考题：对于一个有限元素的生成器，如果迭代完成后，继续调用 `next()`，会跳出 `StopIteration`。生成器可以遍历多次吗？不行。也正是这个原因，老师代码复杂化那段代码，在

```
gen = ((i in b) for i in a)...
```

作者回复: 👍



4





**Destroy1**

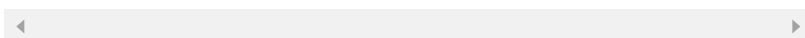
```
def is_subsequence(a, b):
```

```
    b = iter(b)
```

```
    print(b)
```

```
    gen = (i for i in a)...
```

作者回复: 解释的很好



👍 3



**tt**

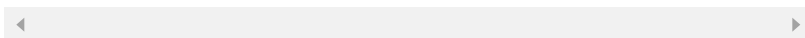
2019-06-21

明白为啥要把b转换成迭代器了，是为了下面的代码中可以用next():

```
while True:
```

```
    val = next(b)...
```

作者回复: 对，这里是个很巧妙的利用



👍 2



**Hurt**

2019-06-21

老师并没讲迭代器和可迭代对象区别，另外还有他们底层的实现，魔法方法



**鱼\_XueTr**

2019-06-21

会引发StopIteration。  
生成器只能使用一次。

作者回复:



**kyle**

2019-06-21

```
gen = ((i in b) for i in a)
```

实际上是先遍历 a，取出一个值赋给i，然后再判断i是否在b中，判断一次，b中的指针后移一位。

所以，第一轮的输出应该是:TRUE, TRUE, TRUE(前三...



**honnkyou**

2019-06-21

“接下来我们再来看一个问题：给定两个序列，判定第一个是不是第二个的子序列。”

老师，这个问题复杂版的代码为什么会有4个判断值。

true,true,true,false跟true,true,false,false



**Monroe He**

2019-06-24

列表元组字典集合都是可迭代对象，它们可以通过 `iter()` 函数返回一个迭代器，再通过 `next()` 函数就可以实现遍历，`for in` 语句将整个过程隐式化。

```
gen = ((i in b) for i in a)
```

```
print(gen) ...
```



**yshan**

2019-06-24

```
while True:
```

```
    val = next(b)
```

```
    if val == i:
```

```
        yield True
```

```
...
```





**yshan**

2019-06-24

抛出StopIteration



**Jon徐**

2019-06-24

迭代器和生成器是以前只是在模块中碰到用一下，没有深入的理解过。迭代器只能遍历一次，只能向前遍历，不能回退。最后的例子使用这一特性将生成器和迭代器配合使用，非常巧妙，回味了很久。



**小侠龙旋风**

2019-06-23

```
>>> gen = (i for i in range(3))
>>> next(gen)
0
>>> next(gen)
1...
```



**Geek\_59f23e**

2019-06-23



```
def next(iterator, default=None):
```

```
    """
```

```
    next(iterator[, default])
```

Return the next item from the iterator. If default...



**Wing·三金**

2019-06-22

思考题：其实开头就已经明示了答案，会出现  
StopIteration Error。遍历是一次性，参考下面这段代  
码：

```
def index_generator(L, target):...
```



**Geek\_59f23e**

2019-06-22

呃。是我弄错了 next函数不设置默认值时迭代完成后调  
用它会抛出StopIteration。

如果指定default参数值，则不会抛出StopIteration，而  
是返回default值。





**Geek\_59f23e**

2019-06-22

1、大家对next函数可能有些误区，迭代完成后继续调用next函数会返回默认值None。

iterator.\_\_next\_\_() 方法和 next(iterator, default=None) 函数的区别在于：前者迭代完成后会抛出StopIteration错误，中断程序运行，而后者会返回一...

作者回复: 👍

