# 第12章. 迭代

## 12-1. 迭代一个对象的基本方法

（教程：4.2、4.4）
（语言参考手册：8.3）

很多情况下，我们需要遍历一个容器的所有元素。当该容器是序列时，我们可以先取得其长度，然后通过while语句达到这一目的。下面是一个例子：

```
#这是一个用while语句迭代序列的例子。
def print_sequence(seq):
    #取得序列的长度。
    length = len(seq)
    #基于索引遍历序列中的元素。
    i = 0
    while i < length:
        print(seq[i])
        i += 1
    else:
        print("Totally " + str(length) + " elements.")
```

请将这段代码保存为iterate1.py，然后通过如下命令行和语句验证：

```
$ python3 -i iterate1.py

>>> print_sequence('abc')
a
b
c
Totally 3 elements.
>>> print_sequence(bytearray([1, 2, 234, 69, 5]))
1
2
234
69
5
Totally 5 elements.
>>> print_sequence(['a', 32.91, (1, 2, 3), NotImplemented])
a
32.91
(1, 2, 3)
NotImplemented
Totally 4 elements.
>>> print_sequence(range(10, 20, 5))
10
15
Totally 2 elements.
>>>
```

然而while语句只能遍历序列，因为序列具有本质为一个等差数列的索引。当遍历其他容器时while语句就力不从心了。迭代一个容器的基本方法是使用for语句，其语法为：

```
for targets in expr:
    suite1
else:
    suite2
```

其中targets可以是一个标识符，也可以是一个逗号分隔的标识符列表（这种情况属于解包赋值），用于引用每次迭代取得的元素；expr是一个求值结果为容器的表达式。该语句的含义是：首先对expr求值，然后迭代得到的容器，将每次迭代取得的元素赋值给targets以执行suite1，当完成了该容器的遍历后再执行suite2。注意for语句每次迭代都会对targets进行一次赋值，因此也是一种标识符绑定操作，这在第3章已经提到了。

下面的是一个迭代字典的例子：

```
#这是一个用for语句迭代字典的例子。
def print_dictionary(dict):
    #遍历字典中的元素。
    for key in dict:
        print(str(key) + ": " + str(dict[key]))
    else:
        print("Traversal finished.")
```

请将这段代码保存为iterate2.py，然后通过如下命令行和语句验证：

```
$ python3 -i iterate2.py

>>> print_dictionary({0: 'a', 1: 'b', 2: 'c'})
0: a
1: b
2: c
Traversal finished.
>>> print_dictionary(dict(a=None, b=NotImplemented, c=Ellipsis))
a: None
b: NotImplemented
c: Ellipsis
Traversal finished.
>>>
```

上面的例子说明，for语句迭代一个容器不需要知道它的长度，也不依赖于索引。事实上，for语句依赖的是“迭代器（iterators）”，会在下面详细讨论，这里只需要知道迭代器能够保证对容器中的每个元素都恰好访问一次。

上面的例子还说明，当用for语句迭代一个映射时，取得的是键值对中的键。当然，取得键之后很容易就可以获得相应的值。然而如果想通过for语句直接取得键、值或键值对，那么可以将被迭代的对象设置为相应的视图，例如对于字典来说可以调用dict.keys()、dict.values()和dict.items()。

下面的例子说明了迭代视图的作用：

```
#这是一个用for语句迭代字典的视图例子。


#迭代键构成的视图。
def print_dictionary_keys(dict):
    #遍历字典中的键。
    for key in dict.keys():
        print(key)
    else:
        print("Traversal finished.")


#迭代值构成的视图。
def print_dictionary_values(dict):
    #遍历字典中的值。
    for value in dict.values():
        print(value)
    else:
        print("Traversal finished.")


#迭代键值对构成的视图。
def print_dictionary_items(dict):
    #遍历字典中的键值对。
    for item in dict.items():
        print(item)
    else:
        print("Traversal finished.")
```

请将这段代码保存为iterate3.py，然后通过如下命令行和语句验证：

```
$ python3 -i iterate3.py

>>> d = {"given_name": "Joe", "surname": "Moore", "age": 43}
>>> print_dictionary_keys(d)
given_name
surname
age
Traversal finished.
>>> print_dictionary_values(d)
Joe
Moore
43
Traversal finished.
>>> print_dictionary_items(d)
('given_name', 'Joe')
('surname', 'Moore')
('age', 43)
Traversal finished.
>>>
```

就像while语句存在省略了else子句的变体一样，for语句也可以省略else子句，即：

```
for targets in expr:
    suite
```

下面用迭代集合为例子说明for语句的这种变体：

```
#这是一个用for语句迭代集合的例子。
def print_set(st):
    #遍历集合中的元素。
    for ele in st:
        print(ele)
```

请将这段代码保存为iterate4.py，然后通过如下命令行和语句验证：

```
$ python3 -i iterate4.py

>>> print_set({0j, 1j, 2j})
0j
1j
2j
>>> print_set(frozenset('abcd'))
d
b
c
a
>>>
```

与while语句类似，在for语句的suite1中可以通过break语句跳出迭代，也可以通过continue语句提前开始下一次迭代。下面的例子仅打印一个等差数列小于10的部分：

```
>>> for n in range(5, 20):
...     if n >= 10:
...         break
...     print(n)
...
5
6
7
8
9
>>>
```

而下面的例子仅打印一个整数集合中的偶数：

```
>>> st = {-27, 94, 82, 53, 417, 66}
>>> for n in st:
...     if n % 2 == 1:
...         continue
```

```
...     print(n)
...
66
82
94
>>>
```

当Python脚本以多线程方式被执行时，为了迭代一个容器，应先调用它的copy()属性以获得一个拷贝，然后再迭代该拷贝。这样能避免该容器被多个线程同时访问导致的冲突。当然，通过线程锁也可以解决此类冲突，但这会大幅度提高对编程技能的要求。

同样，for语句也可以嵌套，下面是一个例子：

```
>>> for x in range(1, 4):
...     for y in ['a', 'b']:
...         print(x*y)
...
a
b
aa
bb
aaa
bbb
>>>
```

最后需要强调，第11章介绍的所有内置容器类型的实例都是可迭代的，但存在不可迭代的自定义容器类型的实例，也存在可迭代但不属于容器类型的实例。我们把能被迭代的对象统称为“可迭代对象（iterables）”，不论其类型是什么。

## 12-2. 解包赋值

（教程：4.8.5、5.6）
（语言参考手册：6.2.3~6.2.7、6.15、7.1、7.2、7.5）

除了for语句之外，还有一种语法可能导致一个对象被迭代，即“解包赋值（unpacking assignment）”。这是对第3章中介绍的赋值语句语法的扩展，可以概括为：

```
identifier_list = expression_list
```

其中identifier_list代表一个标识符列表，expression_list代表一个表达式列表。这种赋值要求表达式列表与标识符列表形成“一一对应”。然而这种“一一对应”又存在非常多的情况，下面将尽可能全面地介绍所有可能的情况。

标识符列表是用逗号分隔的，其中的项可以是：

- 标识符。
- 对象的属性。
- 可变序列或可变映射的抽取。
- 可变序列的切片。
- 可迭代对象（其内可以包含所有这7种情况的项）。
- *可迭代对象（其内可以包含所有这7种情况的项）。
- *标识符。

情况1~4不可能导致一个对象被迭代，在之前已经讨论过了。情况5仅当标识符列表没有逗号时才会导致该可迭代对象被迭代，否则该可迭代对象仅会被视为列表中的一项。情况6总是导致该可迭代对象被迭代。情况7导致该标识符被赋值一个列表，以容纳表达式列表中所有未匹配的项。

表达式列表也是用逗号分隔的，其中的项可以是：

- 不可迭代对象。
- 可迭代对象（其内可以包含所有这3种情况的项）。
- *可迭代对象（其内可以包含所有这3种情况的项）。

对于情况1，该对象肯定不会被迭代。对于情况2，仅当标识符列表中至少有一个逗号而表达式列表中没有逗号时该可迭代对象才会被迭代。对于情况3，该可迭代对象总是会被迭代。

“解包（unpacking）”的含义其实就是迭代一个对象，以获得一个对象序列。如果被迭代的是一个容器， 那么得到的是它的元素形成的序列。即便标识符列表和表达式列表中都不存在可迭代对象，上述格式的赋值操作也存在解包，因为逗号分隔的列表本身就隐式创建了一个元组。

下面是一个最简单的解包赋值的例子：

```
>>> a, b, c = 1, 2, 3
>>> a
1
>>> b
2
>>> c
3
>>>
```

该例子中的标识符列表所有项都属于情况1，表达式列表所有项同样都属于情况1。

下面这个解包赋值的例子中，标识符列表所有项都属于情况1，表达式列表所有项都属于情况2：

```
>>> a, b, c = 'ab', 'cde', 'f'
>>> a
'ab'
>>> b
'cde'
>>> c
'f'
>>> a, b, c = 'abc'
>>> a
'a'
>>> b
'b'
>>> c
'c'
>>> a, = 'abc'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 1)
>>> a, b, c = 'a', 'bc'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not enough values to unpack (expected 3, got 2)
>>> a, b, c = 'abcd'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 3)
>>>
```

该例子说明了表达式列表包含可迭代对象时，仅当其自身没有逗号（亦即表达式列表就是一个可迭代对象），而标识符列表中有逗号时，才会导致可迭代对象被迭代。而可迭代对象被迭代后，仅当标识符列表中的项数与可迭代对象的元素数相同时才能赋值成功。

下面这个解包赋值的例子中，标识符列表中包含属于情况5的项，而这些项内部又包含属于情况2的项；表达式列表则包含属于情况1或情况2的项：

```
>>> class C:
...     pass
...
>>> obj = C()
>>> (obj.a, obj.b, obj.c) = 1, 2, 3
>>> obj.__dict__
{'a': 1, 'b': 2, 'c': 3}
>>> (obj.a, obj.b, obj.c) = (4, 5), 6
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not enough values to unpack (expected 3, got 2)
>>> (obj.a, obj.b), obj.c = 4, 5, 6
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
>>> (obj.a, obj.b), obj.c = (4, 5), 6
>>> obj.__dict__
{'a': 4, 'b': 5, 'c': 6}
>>>
```

该例子说明在解包赋值中解包是递归的。当一个可迭代对象被赋值给包含标识符的可迭代对象时，两者会被进一步解包。

下面继续前面的例子，但在标识符列表引入属于情况6的项，在表达式列表中引入属于情况3的项：

```
>>> *(obj.a, obj.b), obj.c = 7, 8, 9
>>> obj.__dict__
{'a': 7, 'b': 8, 'c': 9}
>>> obj.a, obj.b, obj.c = 10, *(11, 12)
>>> obj.__dict__
{'a': 10, 'b': 11, 'c': 12}
>>>
```

这个例子说明添加“*”的作用是改变解包赋值时对可迭代对象的默认处理规则，强制解包一个可迭代对象。但如果一个可迭代对象按照默认处理规则就会被解包，则添加“*”会被视为语法错误。

下面的例子在标识符表达式中包含属于情况7的项：

```
>>> head, *body, tail = 'abcde'
>>> head
'a'
>>> body
['b', 'c', 'd']
>>> tail
'e'
>>> head, *body, tail = 'abc'
>>> head
'a'
>>> body
['b']
>>> tail
'c'
>>> head, *body, tail = 'ab'
>>> head
'a'
>>> body
[]
>>> tail
'b'
>>> head, *body, tail = 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not enough values to unpack (expected at least 2, got 1)
>>>
```

需要强调的是，标识符列表中至多有1个属于情况7的项，而表达式列表在解包后的对象数不能小于标识符列表在解包后的标识符数减去1。在赋值时，首先从头向尾赋值属于情况7的项之前的项，再从尾向头赋值属于情况7的项之后的项，最后表达式列表中剩下的对象形成的列表（可以是空列表）将被赋值给属于情况7的项。

而标识符列表中属于情况3和4的项，与属于情况2的项没有本质上的区别，且在第11章中已经详细讨论过，所以这里省略。

解包赋值是对第3章介绍的赋值语句语法的扩展。相应的，第3章介绍的del语句的语法也可以扩展为：

```
del identifier_list
```

即一次删除多个标识符。注意这会严格按照标识符列表的顺序依次删除列出的标识符。下面是一些例子：

```
>>> del a, b, c
>>> del (obj.a, obj.b)
>>> obj.__dict__
{'c': 12}
>>> del obj, obj.c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'obj' is not defined
>>>
```

解包赋值对于第11章中介绍的创建元组、列表、字典和集合的语法也产生了影响。首先，用于创建元组的表达式列表中可以包含“*iterable”格式的项，例如：

```
>>> 1, 2, *'abc', 3
(1, 2, 'a', 'b', 'c', 3)
>>>
```

然后，用于创建列表的列表显示的语法也随之被扩展，例如：

```
>>> [1, 2, *'abc', 3]
[1, 2, 'a', 'b', 'c', 3]
>>>
```

再次，用于创建字典的字典显示中可以包含“**mapping”格式的项，例如：

```
>>> {1: 'one', **dict(a=1.0, b=2.0), 2: 'two'}
{1: 'one', 'a': 1.0, 'b': 2.0, 2: 'two'}
>>>
```

最后，用于创建集合的集合显示中也可以包含“*iterable”格式的项，例如：

```
>>> {0, *'abc', 1}
{0, 1, 'a', 'b', 'c'}
>>>
```

所有这些扩展中只有“**mapping”格式的项是特殊的，不能出现在解包赋值中。

现在请回忆在第4章中讨论函数调用时指出函数调用隐含一次解包赋值。结合任意实参列表和解包实参列表：“*name”形式的形式参数可以接受任意多个基于位置对应的实际参数；“**name”形式的形式参数可以接收任意多个基于关键字对应的实际参数；“*expr”形式的实际参数会被传入多个基于位置匹配的形式参数；而“**expr”形式的实际参数会被传入多个基于关键字匹配的形式参数。这些其实都是与解包赋值及其衍生技术相一致的。

解包赋值也可以与for语句联合使用。下面的例子在每次迭代过程中都将映射的键值对解包赋值给两个变量——key引用键，value引用值：

```
>>> for key, value in {0:'a', 1:'b', 2:'c'}.items():
...     print(str(key) + '->' + str(value))
...
0->a
1->b
2->c
>>>
```

## 12-3. 推导式

（教程：5.1.3、5.1.4）
（语言参考手册：6.2.4～6.2.7）

很多情况下，一组对象可以从另一组对象推导出来。一个典型例子是数学中那些存在通项公式的数列，例如等差数列的通项公式是$a_n = a_0 + nd$，等比数列的通项公式是$a_n = a_0 * q^n$。这些通项公式隐含了这样一种推导过程：我们已经知道自然数形成的集合，而数列中的第n项可以通过将n带入其通项公式计算出来。

另一个典型例子是英文中的构词法。假如我们知道了一组单词，且知道某些前缀和某些后缀，那就能以这组单词为基础，使用不同的前缀和后缀的组合构造出大量新单词。当然，单词、前缀和后缀并非能够任意组合，某些前缀和某些后缀冲突，某些单词不支持某些前缀/后缀，因此在构词时还需要遵循某些筛选条件。

Python中的“推导式（comprehensions）”与上述两个例子的原理相同：如果我们已经知道了一组可迭代对象，则可以制定一种可应用于这组可迭代对象的推导规则，以基于它们推导出另一组可迭代对象。而推导式就是这种推导规则的描述，它的语法为：

```
expr0 comp_for {comp_for | comp_if}*
```

其中expr0是一个涉及若干标识符的表达式；comp_for的语法与for语句的第一行类似，即：

```
for identifier in expr
```

其中identifier是expr0中涉及的某个标识符，expr则是求值结果为某可迭代对象的表达式，含义是迭代对expr求值得到的可迭代对象，将获得的对象依次赋值给identifier；comp_if的语法则与if语句的第一行类似，即：

```
if expr
```

其中expr是涉及位于它之前的comp_for中出现过的标识符的一个表达式，含义是如果expr的求值结果经逻辑值检测后为False，则不继续后面的操作，开始新的迭代。

我们应这样理解推导式：它代表着若干相互嵌套的for语句和if语句，for语句实现迭代，if语句实现筛选。推导式的功能是生成expr0中涉及的标识符的合法的值组合，而将这些值组合赋值给这些标识符后，对expr0求值得到的对象即由该推导式推导出的对象。

推导式无法单独被使用，具体有如下使用方式：

- 被“()”括起来构成生成器表达式。
- 被“[]”括起来构成列表显示，称为“列表推导式”。
- 被“{}”括起来构成字典显示，称为“字典推导式”。
- 被“{}”括起来构成集合显示，称为“集合推导式”。

注意字典推导式中的expr0的格式需略微修改，即通过如下格式：

```
expr_key: expr_value
```

以获得键值对。本节讨论推导式的后三种用法，而生成器表达式会在本章后面被讨论。

下面的例子用推导式创建了一个列表：

```
>>> l = [x*y for x in range(1, 4) for y in 'ab' + 'c' if not (x==2 and
y=='b')]
>>> l
['a', 'b', 'c', 'aa', 'cc', 'aaa', 'bbb', 'ccc']
>>>
```

该例子中的列表推导式与下面这段代码等价：

```
>>> l = []
>>> for x in range(1, 4):
...     for y in 'ab' + 'c':
...         if not (x==2 and y=='b'):
...             l.append(x*y)
...
>>> l
['a', 'b', 'c', 'aa', 'cc', 'aaa', 'bbb', 'ccc']
>>>
```

但显然使用列表推导式要简洁得多。

下面的例子用推导式创建了一个集合：

```
>>> st = {(x, y) for x in range(10) if x%3 != 0 for y in range(x, 10) if
y*x > 50}
>>> st
{(7, 9), (8, 8), (8, 9), (7, 8)}
>>>
```

该例子中的集合推导式与下面这段代码等价：

```
>>> st = set()
>>> for x in range(10):
...     if x%3 != 0:
...         for y in range(x, 10):
...             if y*x > 50:
...                 st.add((x, y))
...
>>> st
{(7, 9), (8, 8), (8, 9), (7, 8)}
>>>
```

注意在该例子中对expr0求值得到一个元组，此时必须写作“(x, y)”而不能写作“x, y”，否则会引起混淆（因为不添加圆括号的话可以理解为一个表达式列表，第一项为x，第二项为一个以y开头的推导式）。

下面的例子则用推导式创建了一个字典：

```
>>> d = {x: x**2 for x in range(10)}
>>> d
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
>>>
```

该例子中的字典推导式与下面这段代码等价：

```
>>> d = {}
>>> for x in range(10):
...     d[x] = x**2
...
>>> d
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
>>>
```

列表推导式、字典推导式和集合推导式相互间可以嵌套，也就是说一个列表推导式、字典推导式或集合推导式的expr0本身可以包含另一个列表推导式、字典推导式或集合推导式。Python官方手册的5.1.4给出了一个通过嵌套列表推导式实现矩阵转置的例子。下面给出的例子则将集合推导式嵌套在了字典推导式中：

```
>>> strings = ('apple', 'pear', 'banana')
>>> char_stat = {s: {char for char in s} for s in strings}
>>> char_stat
{'apple': {'p', 'e', 'l', 'a'}, 'pear': {'p', 'r', 'e', 'a'}, 'banana':
{'n', 'b', 'a'}}
>>>
```

## 12-4. 迭代器

（教程：9.8、5.6）
（标准库：内置函数、内置类型、内置异常）

for语句进行迭代时依赖迭代器。如果一个类同时满足了如下两个条件，那么它就是一个迭代器类，其实例就是迭代器：

➤ 实现了魔术属性__iter__，并使其总是返回调用它的实例。
➤ 实现了魔术属性__next__，并使其能够改变调用它的实例的内部状态，然后根据当前内部状态返回不同的对象。实例必须至少有一种内部状态将导致抛出StopIteration异常，且一旦进入这类内部状态就无法再改变。

上述条件被称为“迭代器协议（iterator protocol）”，因此迭代器就是实现了迭代器协议的对象。

第8章已经说明，StopIteration异常是一种较特殊的异常，代表着迭代器已经完成了迭代，其实例化语法为：

```
class StopIteration(*args)
```

通过args代表的实际参数列表中的第一项还会被StopIteration异常的value属性引用。当没有任何实际参数时，等价于以None为实际参数。一旦某个迭代器抛出了StopIteration异常，其内部状态就不再能够改变，因此该迭代器也就无法再使用了。换句话说，所有迭代器都是一次性的。

我们可以通过内置函数next()手工调用迭代器的__next__，其语法为：

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

其中iterator参数需被传入一个迭代器。default参数则可以被传入任意对象，当next()截获了StopIteration异常时会将该对象作为返回值。如果省略了default参数，则next()会将截获的StopIteration异常再次抛出。

下面让我们自定义一个迭代器，然后通过next()来检验它的功能：

```
#本脚本自定义一个迭代器类。
class MyIterator:
    def __init__(self, n):
        self.max_n = int(n)
        self.n = 0

    #满足迭代器协议对__iter__的要求。
    def __iter__(self):
        return self

    #满足迭代器协议对__next__的要求。
    def __next__(self):
        if self.n >= self.max_n:
            raise StopIteration()
        self.n += 1
        return self.n
```

请将这段代码保存为iterate5.py，然后通过如下命令行和语句验证：

```
$ python3 -i iterate5.py

>>> my_iter1 = MyIterator(3)
>>> next(my_iter1)
1
>>> next(my_iter1)
2
>>> next(my_iter1)
3
>>> next(my_iter1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/wwy/iterate5.py", line 14, in __next__
    raise StopIteration()
    ^^^^^^^^^^^^^^^^^^^^
StopIteration
>>> next(my_iter1)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/wwy/iterate5.py", line 14, in __next__
    raise StopIteration()
    ^^^^^^^^^^^^^^^^^^^^
StopIteration
>>> my_iter2 = MyIterator(2)
>>> next(my_iter2, -1)
1
>>> next(my_iter2, -1)
2
>>> next(my_iter2, -1)
-1
>>> next(my_iter2, -1)
-1
>>> my_iter3 = MyIterator(-10)
>>> next(my_iter3, False)
False
>>> next(my_iter3, False)
False
>>>
```

在这个例子中，我们自定义的迭代器接受一个整数，储存在实例属性max_n中，并用实例属性n代表其内部状态。__next__魔术属性将使迭代器的内部状态沿着0，1，……，max_n变化，而返回值是该内部状态本身（但状态0永远不会被返回）；当内部状态变成了大于等于max_n之后，调用__next__将导致抛出StopIterator异常。

for语句的内在执行逻辑是这样的：以对expr表达式求值得到对象作为参数调用内置函数iter()，进而获得一个迭代器；然后重复调用该迭代器的__next__并基于得到的对象执行suite1，直到StopIteration异常被抛出。for语句会自动截获并处理StopIteration异常，处理方式是执行else子句中包含的suite2，然后语句执行结束；如果省略了else子句则直接语句执行结束。

iter()有两种语法，for语句使用的是第一种：

```
iter(iterable)
```

这会基于通过iterable参数传入的对象产生一个迭代器，然后将该迭代器返回。此时iter()的运行步骤是这样的：

➤ 步骤一. 如果通过iterable参数传入的对象具有__iter__魔术属性，则调用__iter__并以其返回值作为自身的返回值。
➤ 步骤二. 否则，如果通过iterable参数传入的对象具有__getitem__魔术属性，且支持从0开始计数的索引，则生成一个类似MyIterator实例的迭代器，只不过该迭代器的内部状态i沿着0，1，……，len(iterable)变化，当i < len(iterable)时返回值是iterable[i]，当i == len(iterable)时抛出StopIteration异常。
➤ 步骤三. 否则，抛出TypeError异常。

反过来，“可迭代对象”就是以之为参数调用iter()能够获得迭代器而非抛出TypeError异常的对象。这意味着该对象具有__iter__或/和__getitem__魔术属性。第11章介绍的所有内置容器类型都实现了__iter__，而序列类型还额外实现了__getitem__。一般而言，我们自定义的容器类型也会实现__iter__或/和__getitem__。这就是几乎总是可以通过for语句迭代容器的原因。

另外，根据迭代器协议的规定，所有迭代器都具有__iter__，因此它们也都是可迭代对象。请通过下面的命令行和语句验证可以通过for语句迭代我们自定义的迭代器：

```
$ python3 -i iterate5.py

>>> for n in MyIterator(5):
...     print(n)
...
1
2
3
4
5
>>> for n in MyIterator(-1):
...     print(n)
...
>>>
```

需要强调，存在既不是容器又不是迭代器的可迭代对象。这种可迭代对象所属类只实现了__iter__，既没有实现__len__和__contains__，又没有实现__next__。下面是一个这种可迭代对象的例子：

```
#本脚本自定义一个可迭代对象类。    它储存一个序列，但被迭代时只取得序列中的偶数项。
class MyIterable:
    def __init__(self, seq):
        self.data = seq

    #定义专属于该可迭代对象类的迭代器类。
    class EvenIterator():
        def __init__(self, data):
            self.data = data
            self.length = len(data)
            self.index = 0

        def __iter__(self):
            return self

        def __next__(self):
            if self.index >= self.length:
                raise StopIteration()
            ele = self.data[self.index]
            self.index += 2
            return ele

    #返回属于EvenIterator类型的迭代器。
    def __iter__(self):
        return self.EvenIterator(self.data)
```

请将这段代码保存为iterate6.py，然后通过如下命令行和语句验证：

```
$ python3 -i iterate6.py

>>> for n in MyIterable([1, 2, 3, 4, 5, 6]):
...     print(n)
...
1
3
5
>>> for s in MyIterable("congratulations!"):
...     print(s)
...
c
n
r
t
l
t
o
s
>>>
```

这里需要指出的是，不具有__contains__的可迭代对象依然支持成员检测，而检测的方法就是迭代该对象，并判断指定的对象是否与某次迭代取得的对象相等。请看下面的例子：

```
$ python3 -i iterate6.py

>>> obj = MyIterable("abcde")
>>> 'a' in obj
True
>>> 'b' in obj
False
>>> 'c' in obj
True
>>> 'd' in obj
False
>>> 'e' in obj
True
>>> 'f' in obj
False
>>>
```

iter()的第二种语法为：

```
iter(callable, sentinel)
```

其callable参数必须被传入一个不需要任何参数的可调用对象，而sentinel参数则可以传入任意对象。此时iter()返回的迭代器将以这样的方式运行：重复调用callable，并将其返回值与sentinel比较，仅当两者相等时才抛出StopIteration异常。下面用一个例子说明iter()的这种用法：

```
#本脚本定义了一个代表计数器的类。
class Counter:
    def __init__(self):
        self.n = 0

    #该属性实现计数，并在每次计数时执行指定的操作。
    def tik(self):
        self.n += 1
        return self.n

    #该属性重置计数器。
    def reset(self):
        self.n = 0
```

请将这段代码保存为Counter.py，然后通过如下命令行和语句验证：

```
$ python3 -i Counter.py

>>> c = Counter()
>>> iterator1 = iter(c.tik, 5)
>>> for n in iterator1:
...     print(n)
...
1
2
3
4
>>> iterator2 = iter(c.tik, 8)
>>> for n in iterator2:
...     print(n)
...
6
7
>>> c.reset()
>>> for n in iter(c.tik, 4):
...     print(n)
...
1
2
3
>>>
```

最后，内置函数reversed()与iter()第一种语法的功能正好相反，基于可迭代对象产生一个方向相反的迭代器，其语法为：

```
reversed(iterable)
```

reversed()与iter()的区别在于如下两点：

1. 如果通过iterable参数传入的对象具有__reversed__魔术属性，则调用__reversed__并以其返回值作为自身的返回值。

2. 如果通过iterable参数传入的对象具有__getitem__魔术属性，且支持从0开始计数的索引，则生成一个类似MyIterator的迭代器，该迭代器的内部状态沿着len(iterable)，……，1，0变化。

显然，__reversed__的迭代顺序应该与__iter__正好相反。下面的例子是对iterate6.py的改写，使得MyIterable同时实现了__iter__和__reversed__：

```
class MyIterable:
    def __init__(self, seq):
        self.data = seq

    class EvenIterator():
        def __init__(self, data):
            self.data = data
            self.length = len(data)
            self.index = 0

        def __iter__(self):
            return self

        def __next__(self):
            if self.index >= self.length:
                raise StopIteration()
            ele = self.data[self.index]
            self.index += 2
            return ele

    #方向相反的迭代器的类型。
    class ReverseEvenIterator():
        def __init__(self, data):
            self.data = data
            self.length = len(data)
            self.index = (self.length-1)//2*2

        def __iter__(self):
            return self

        def __next__(self):
            if self.index < 0:
                raise StopIteration()
            ele = self.data[self.index]
            self.index -= 2
            return ele

    def __iter__(self):
        return self.EvenIterator(self.data)

    #返回方向相反的迭代器。
    def __reversed__(self):
        return self.ReverseEvenIterator(self.data)
```

请将这段代码保存为iterate7.py，然后通过如下命令行和语句验证：

```
$ python3 -i iterate7.py

>>> obj = MyIterable([1, 2, 3, 4, 5, 6])
>>> for n in iter(obj):
```

```
...     print(n)
...
1
3
5
>>> for n in reversed(obj):
...     print(n)
...
5
3
1
>>>
```

对于具有__getitem__魔术属性的可迭代对象，reversed()会非常智能地颠倒迭代过程中隐式生成的索引序列的顺序。请看下面的例子：

```
>>> for s in iter('abcd'):
...     print(s)
...
a
b
c
d
>>> for s in reversed('abcd'):
...     print(s)
...
d
c
b
a
>>>
```

在本节的最后介绍两个创建迭代器的内置函数。如果要在迭代序列的过程中同时取得其索引和元素，则需要使用内置函数enumerate()，其语法为：

```
enumerate(iterable, start=0)
```

其中iterable参数应被传入一个序列（或一个实现了__getitem__魔术属性且支持从0开始计数的索引的自定义类型的对象）；start参数则是一个索引修正值。enumerate()会返回一个迭代器，对其进行迭代将从start指定的索引开始依次得到若干二元组，其中第一个元素等于索引加上start，第二个元素为序列中的对应元素。下面的例子说明了该内置函数的作用：

```
>>> for index, element in enumerate('abc'):
...     print(str(index) + ': ' + str(element))
...
0: a
1: b
2: c
>>> for index, element in enumerate('abc', -3):
...     print(str(index) + ': ' + str(element))
...
```

```
-3: a
-2: b
-1: c
>>>
```

有时候，我们想要同时迭代多个可迭代对象，并用通过迭代这些可迭代对象依次获得的元素构成一个元组。此时需要使用内置函数zip()，其语法为：

```
zip(*iterable, strict=False)
```

可以通过iterable参数传入任意多个可迭代对象，而zip()返回的迭代器每次迭代时都会分别调用这些可迭代对象的__next__，并按照它们被传入的顺序将取得的对象组成一个元组。注意当strict参数被传入False时，允许被传入的可迭代对象具有不同的长度，而zip()返回的迭代器的可迭代次数等于所有这些长度中最小者；而当strict参数被传入True时，所有可迭代对象都必须具有相同的长度，否则会抛出ValueError。下面的例子说明了zip()的作用：

```
>>> for t in zip('abc', [0, 1, 2, 3], {False, True, None}):
...     print(t)
...
('a', 0, False)
('b', 1, True)
('c', 2, None)
>>>
>>> for t in zip('abc', [0, 1, 2, 3], {False, True, None}, strict=True):
...     print(t)
...
('a', 0, False)
('b', 1, True)
('c', 2, None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: zip() argument 2 is longer than argument 1
>>>
>>> for t in zip('abcd', [0, 1, 2, 3], {False, True, None, NotImplemented},
strict=True):
...     print(t)
...
('a', 0, False)
('b', 1, True)
('c', 2, None)
('d', 3, NotImplemented)
>>>
```

在大部分涉及迭代的场合，基于某个可迭代对象产生的迭代器都可以代替该可迭代对象。然而在类定义语句中，如果__slots__引用一个迭代器而非可迭代对象，那么该迭代器会被迭代，得到的每个字符串都会被视为一个标识符，并被创建数据描述器，但当类定义语句执行完之后该迭代器也已经迭代完成，此后__slots__将引用一个无法迭代的迭代器。这通常不是我们想要的，所以一般__slots__都会直接引用可迭代对象。

最后值得一提的是，标准库中的itertools模块提供了更多自定义迭代器的工具，但对它的详细讨论超出了本书的范围。

## 12-5. 生成器

（教程：9.9、9.10）
（语言参考手册：3.2、6.2.8、6.2.9、7.6、7.7）
（标准库：内置类型、内置异常）

从上一节的例子iterate6.py和iterate7.py可以看出，以本书目前为止介绍的语法自定义可迭代类是相当麻烦的：由于迭代器必须能够记录内部状态，所以需要先定义迭代器类，通过实例属性描述内部状态；而可迭代类至少要实现__iter__并让其返回某个迭代器，还可能要实现__reversed__并让其返回另一个迭代器，这些迭代器属于不同的迭代器类，而这些类通常在可迭代类的类体内部定义。这使得定义可迭代类的代码变得臃肿。

“生成器（generators）”是为了更方便地创建迭代器而被引入的技术。生成器就是具有表12-1列出属性的对象。由于同时具有__iter__和__next__，所以每个生成器都支持迭代器协议，可以被当成迭代器使用。然而生成器的功能更多，比迭代器更灵活。

表12-1. 生成器的属性

| 属性 | 说明 |
| --- | --- |
| **generator.__iter__()** | 用于支持iter()，总是返回其自身。 |
| **generator.__next__()** | 用于支持next()和for语句，以实现标准的迭代。 |
| **generator.send()** | 手工启动生成器，并传入一个值。 |
| **generator.throw()** | 手工启动生成器，并抛出一个异常。 |
| **generator.close()** | 手工关闭生成器。 |
| **generator.gi_code** | 引用创建该生成器的生成器函数的函数体对应的代码对象。 |
| **generator.gi_frame** | 引用执行生成器函数的函数体的过程中创建的帧对象。 |
| **generator.gi_yieldfrom** | 引用一个迭代器，用于迭代“yield from”语句指定的对象。 |
| **generator.gi_running** | 引用一个布尔值，表明生成器是否在运行。 |
| **generator.gi_suspended** | 引用一个布尔值，表明生成器是否被挂起。 |

生成器可以通过“生成器函数（generator functions）”或“生成器表达式（generator expressions）”创建。下面先讨论生成器函数。

生成器函数就是函数体内包含yield表达式或yield语句的函数。由于yield表达式仅能在生成器函数的函数体中使用，所以在表2-2中未将其列出。yield表达式的语法有两种，第一种为：

```
(yield expr[, ...])
```

其含义是对yield关键字后面的表达式列表中的每个表达式求值，然后将得到的对象打包为一个元组作为该yield表达式的求值结果；但如果yield关键字后面只有一个表达式（后面也没有逗号），则yield表达式的求值结果即该表达式的求值结果，不产生元组。

yield表达式的第二种语法为：

```
(yield from expr)
```

其中的expr求值结果必须是一个可迭代对象。该语法的含义是迭代对expr求值获得的可迭代对象，依次以得到的对象作为该yield表达式的求值结果。注意这是除了for语句和解包赋值之外，第三种将导致自动迭代一个对象的语法。

就像其他的表达式一样，yield表达式可以被嵌入其他表达式内部，也可以作为赋值语句的右值（“=”右侧的部分）。而所谓的yield语句，其实就是单独一个yield表达式形成的语句，此时可以省略圆括号。

生成器函数的特殊之处在于，在完成对其函数体内某个yield表达式的求值之后，函数体的执行会被挂起，并将求值得到的对象返回给调用者。生成器函数的函数体并非在它自身被调用时执行，而是在它创建的生成器对象的__next__、send()或throw()属性被调用时才会被执行，这是生成器函数与普通函数的最大区别。为了便于说明，下面先给出一个特别简单的生成器函数作为例子：

```
#一个简单的生成器函数。
def gen1():
    print("A")
    yield 1
    print("B")
    yield 2
    print("C")
    yield 3
    print("D")
    return 0
```

请将这段代码保存为generator1.py，然后执行如下命令行和语句：

```
$ python3 -i generator1.py

>>> g = gen1()
>>> type(g)
<class 'generator'>
>>> value = next(g)
A
>>> value
1
>>> value = next(g)
B
>>> value
2
```

```
>>> value = next(g)
C
>>> value
3
>>> value = next(g)
D
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration: 0
>>>
```

上面的例子证明了调用生成器函数会创建一个生成器，而其函数体是在生成器被迭代的过程中被执行的。next()使得生成器g的__next__被调用，第一次调用导致函数体的如下语句被执行：

```
        print("A")
        yield 1
```

第二次调用导致函数体的如下语句被执行：

```
        print("B")
        yield 2
```

第三次调用导致函数体的如下语句被执行：

```
        print("C")
        yield 3
```

可见迭代生成器时会从函数体的开头开始执行，第n次迭代将执行到第n个yield表达式为止，该yield表达式会被求值，得到的对象成为本次调用__next__的返回值，然后函数体的执行被挂起。而第四次调用导致函数体的如下语句被执行：

```
        print("D")
        return 0
```

从上面例子可以看出，通过return语句返回的对象并不会成为本次调用__next__的返回值，因为当生成器函数的函数体执行完成时会自动抛出StopIteration异常，而该函数体的返回值会被该异常的value属性引用。对于上面的例子来说，“return 0”其实等价于：

```
        raise StopIteration(0)
```

for语句不会使用StopIteration异常的value属性引用的对象，但当我们通过next()手工迭代一个生成器，并通过try语句截获StopIteration异常以自行处理时，可以使用该对象。一个典型的应用场景是生成器函数的函数体被执行完成的情况有多种，每种情况对应不同的返回值，可以通过StopIteration异常的value属性判断该生成器是在什么情况下执行完成的。

为了支持挂起以及恢复运行，生成器必须记录相应生成器函数的函数体，以及该函数体被挂起时相应帧对象的状态。此外，如果导致函数体被挂起的yield表达式采用了第二种语法，那么生成器还必须记录下关联到相关可迭代对象的迭代器的内部状态。生成器的这些功能是通过实例属性gi_code、gi_frame和gi_yieldfrom实现的。此外，生成器还通过__name__和__qualname__特殊属性记录了相关生成器函数的函数名和限定名，通过__doc__特殊属性记录了相关生成器函数的文档字符串。换句话说，一个生成器函数可以产生任意多个生成器，而每个生成器都记录了该生成器函数本身的一切信息，区别仅在于帧对象的状态。

生成器被创建后是被挂起的，但此时gi_code属性和gi_frame属性已经引用了相应代码对象和帧对象，只不过该帧对象并不会被放入函数栈中（其f_back属性永远引用生成器被创建时被执行的帧对象），记录的状态相当于生成器函数刚完成了实际参数到形式参数的赋值，但尚未执行函数体的第一条语句。此后生成器需要通过调用__next__、send()或throw()来启动，进入运行状态，执行函数体到下一个yield表达式被求值或函数返回，重新被挂起。

上述过程会导致gi_frame引用的帧对象不断变化。gi_yieldfrom属性默认引用None，仅当使用第二种语法的yield表达式被求值后，该属性才会引用关联到相应可迭代对象的迭代器。而当StopIteration异常被抛出后，gi_frame和gi_yieldfrom都将引用None，但gi_code依然引用原来的代码对象。特别的，如果调用gi_frame引用的帧对象的clear()属性，则会导致该生成器一直运行到StopIteration异常被抛出。

综上所述，生成器在大部分情况下都是被挂起的，其gi_running属性引用False；仅当生成器短暂处于运行状态期间，其gi_running属性才会引用True。由于在生成器函数的函数体内无法访问到它所创建的生成器，而对于单独一个线程来说，访问某个生成器的gi_running属性一定会得到False，所以生成器的gi_running属性仅在多线程环境下才有意义。

另一方面，gi_suspended属性则大体上与gi_running属性的取值相反，但存在一点很重要的区别：当生成器刚被创建，__next__属性尚未被调用时，gi_suspended属性引用的是False而非True。故该属性可被用于判断一个生成器是否已经生成了至少一个对象。

下面用一个例子说明生成器的上述属性的作用：

```
#一个具有参数且使用yield语句的第二种语法的生成器函数。
def gen2(n):
    """This generator generates sequence 0, 1, ..., n-1."""
    yield from range(n)
```

请将这段代码保存为generator2.py，然后执行如下命令行和语句：

```
$ python3 -i generator2.py

>>> g1 = gen2(3)
```

```
>>> g2 = gen2(4)
>>> g1.__name__
'gen2'
>>> g2.__name__
'gen2'
>>> g1.__qualname__
'gen2'
>>> g2.__qualname__
'gen2'
>>> g1.gi_code
<code object gen2 at 0x10b3c5370, file "/Users/wwy/generator2.py", line 3>
>>> g2.gi_code
<code object gen2 at 0x10b3c5370, file "/Users/wwy/generator2.py", line 3>
>>> g1.gi_frame
<frame at 0x10b3ec440, file '/Users/wwy/generator2.py', line 3, code gen2>
>>> g2.gi_frame
<frame at 0x10b307300, file '/Users/wwy/generator2.py', line 3, code gen2>
>>> g1.gi_yieldfrom
>>> g2.gi_yieldfrom
>>> next(g1)
0
>>> next(g2)
0
>>> g1.gi_yieldfrom
<range_iterator object at 0x10b3b3180>
>>> g2.gi_yieldfrom
<range_iterator object at 0x10b3d42a0>
>>> g1.gi_suspended
False
>>> g2.gi_suspended
False
>>>
>>> next(g1)
1
>>> next(g2)
1
>>> g1.gi_suspended
True
>>> g2.gi_suspended
True
>>> next(g1)
2
>>> next(g2)
2
>>> next(g1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> next(g2)
3
>>> g1.gi_code
<code object gen2 at 0x10b3c5370, file "/Users/wwy/generator2.py", line 3>
>>> g1.gi_frame
>>> g1.gi_yieldfrom
>>> g2.gi_code
<code object gen2 at 0x10b3c5370, file "/Users/wwy/generator2.py", line 3>
>>> g2.gi_frame
<frame at 0x10b307300, file '/Users/wwy/generator2.py', line 5, code gen2>
>>> g2.gi_yieldfrom
<range_iterator object at 0x10b3d42a0>
>>> next(g2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> g2.gi_code
```

```
    <code object gen2 at 0x10b3c5370, file "/Users/wwy/generator2.py", line 3>
    >>> g2.gi_frame
    >>> g2.gi_yieldfrom
    >>>
```

从这个例子可以看出，同一生成器函数产生的不同生成器共享代码对象，而帧对象和迭代器是相互独立的，运行时互不干扰。

上面的两个例子都在通过next()手工迭代生成器。下面给出一个例子，说明如何通过生成器函数实现自定义可迭代类，以及用for语句自动迭代生成器：

```
class MyIterable:
    def __init__(self, seq):
        self.data = seq

    def __iter__(self):
        length = len(self.data)
        i = 0
        while i < length:
            yield self.data[i]
            i += 2

    def __reversed__(self):
        i = (len(self.data)-1)//2*2
        while i >= 0:
            yield self.data[i]
            i -= 2
```

请将这段代码保存为generator3.py，然后执行如下命令行和语句：

```
$ python3 -i generator3.py

>>> obj = MyIterable([1, 2, 3, 4, 5, 6])
>>> for n in iter(obj):
...     print(n)
...
1
3
5
>>> for n in reversed(obj):
...     print(n)
...
5
3
1
>>>
```

比较generator3.py和iterate7.py，你就会发现两者实现了同样的功能，但通过将__iter__和__reversed__实现为生成器函数，generator3.py要精简得多。

至此我们已经阐明如果将生成器当成迭代器来使用。下面将讨论生成器额外具有的功能。事实上，这些功能是为了用生成器实现协程而引用的，但现在已经有了专门的协程语法，所以没必要再这样使用生成器。协程会在第14章详细讨论。

首先，我们可以通过调用生成器的send()属性来手工迭代一个生成器，其语法为：

```
generator.send(value)
```

使用该属性与使用next()的区别在于，通过value参数传入的对象将作为导致上一次挂起的yield表达式的求值结果以参与之后的运算，而使用next()时前一次yield表达式的求值结果永远是None。注意当用send()属性启动一个生成器时，由于不存在导致上一次挂起的yield表达式，所以value参数必须被传入None。下面的例子说明了send()属性的用法：

```
#一个会使用yield表达式的求值结果的生成器函数。
def gen3(start, upper_limit):
    n = start
    while n < upper_limit:
        print(n)
        n *= (yield n)
```

请将这段代码保存为generator4.py，然后执行如下命令行和语句：

```
$ python3 -i generator4.py

>>> try:
...     g = gen3(2, 100000)
...     n = g.send(None)
...     while True:
...         n = g.send(n)
... except StopIteration:
...     print("Bigger than upper limit!")
...
2
4
16
256
65536
Bigger than upper limit!
>>>
```

在这个例子中，从生成器g被启动开始，每次yield表达式返回的n都会在下一次被传入，以进行“n *= n”的运算，这样就得到了这样一个数组：$a_n = a_{n-1} * a_{n-1}$。而生成器函数gen3()的start参数用来设置$a_0$，upper_limit参数用来限制数组的最大项。

生成器函数的函数体也可能抛出异常，且可以使用try语句或with语句进行异常处理。特别的，如果函数体没有执行完成就抛出StopIteration异常，则它会自动被截获，并被转换成一个RuntimeError异常，后者通过__cause__属性引用前者。而生成器的throw()属性的功能是手工触发异常，其语法为：

```
generator.throw(expression)
```

其中expression参数需被传入一个求值结果为某异常的表达式。这等价于生成器进入运行状态后先额外执行如下语句：

```
raise expression
```

下面的例子说明了throw()属性的用法：

```
import random

#一个进行了异常处理的生成器函数:
def gen4():
    try:
        while True:
            n = random.random()
            yield n
    except RuntimeError:
        return 0
```

请将这段代码保存为generator5.py，然后执行如下命令行和语句：

```
$ python3 -i generator5.py

>>> g1 = gen4()
>>> g1.send(None)
0.082583468557756
>>> g1.send(None)
0.47413976189036566
>>> g1.send(None)
0.9368945841268356
>>> g1.throw(RuntimeError())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration: 0
>>>
>>> g2 = gen4()
>>> g2.send(None)
0.27862692077145557
>>> g2.send(None)
0.9783414971083237
>>> g2.throw(StopIteration(100))
Traceback (most recent call last):
  File "/Users/wwy/generator5.py", line 8, in gen4
    yield n
    ^^^^^^^
StopIteration: 100

The above exception was the direct cause of the following exception:
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: generator raised StopIteration
>>>
```

上面例子中定义的生成器函数gen4()创建的生成器本质上是一个随机序列生成器，通过调用其send()属性可以无限生成[0.0, 1.0)内的随机数，仅当通过调用其throw()属性抛出了某个异常后，该生成器才会停止运行。调用g1.throw()时抛出的是RuntimeError异常，这会被生成器函数的函数体中的try语句截获并处理，最后通过return语句返回0，进而被转换成value属性引用0的StopIteration异常。而调用g2.throw()时抛出的是StopIteration异常，其value属性引用100，该异常不会被生成器函数的函数体中的try语句截获，但在抛出到函数体之外时会自动被转换为RuntimeError异常，且后者通过__cause__属性引用前者。

然而，通过抛出异常的方式让生成器停止运行是比较笨拙的，更好的方式是直接调用生成器的close()属性，其语法为

```
generator.close()
```

这会导致生成器进入运行后抛出GeneratorExit异常。第8章已经说明，GeneratorExit是一种较特殊的异常类型，处理它的默认流程是让生成器完成对自身的清理，效果与生成器函数的函数体执行完成相同，但不会抛出StopIteration异常。事实上，生成器具有__del__属性，在其被销毁时会自动调用其close()属性。下面是对generator5.py的改写：

```
import random

def gen5():
    try:
        while True:
            n = random.random()
            yield n
    except Exception:
        print("Stop generates random numbers.")
        return 0
```

请将这段代码保存为generator6.py，然后执行如下命令行和语句：

```
$ python3 -i generator6.py

>>> g = gen5()
>>> g.send(None)
0.5198079249390933
>>> g.send(None)
0.39014987808352775
>>> g.throw(RuntimeError())
Stop generates random numbers.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration: 0
```

```
>>> g = gen5()
>>> g.send(None)
0.13122585008589738
>>> g.send(None)
0.1048590071682568
>>> g.close()
>>>
```

从上面的例子可以看出，通过close()属性关闭生成器更干净利落。事实上，throw()属性更多的时候是为了抛出某种自定义异常，以结合try语句给生成器增加额外的功能。

我们对通过生成器函数产生生成器的讨论到此为止。接下来讨论生成器表达式。前面已经提到了，用圆括号将推导式括起来就构成了生成器表达式，对该表达式求值的结果是一个生成器。请看下面的例子：

```
>>> g = ((x, y) for x in range(2) for y in (x, 2))
>>> type(g)
<class 'generator'>
>>> g.__name__
'<genexpr>'
>>> g.__qualname__
'<genexpr>'
>>> for coordinate in g:
...     print(coordinate)
...
(0, 0)
(0, 2)
(1, 1)
(1, 2)
>>>
```

注意在生成器表达式中不能有yield表达式，这是因为推导式中除了第一个comp_for之外的所有comp_for都有自己的作用域。这些作用域在内部是基于函数作用域实现的，因此相当于隐式形成了函数调用的嵌套，推导式中的expr0相当于在最内层函数调用中被执行。而yield表达式的作用是将其求值结果返回给外层函数调用，并使得内层函数调用被挂起。不论它出现在expr0中还是某个comp_for中，都会使得相应的某层函数调用创建一个未被引用的生成器，进而导致错误。

## 12-6. 迭代的应用

（标准库：内置函数、functools、io）

可迭代对象是很重要的一类对象，支持很多独有操作。本节重点讨论通过内置函数实现的此类操作。

---

首先，任何可迭代对象都可以被视为数学意义上的一个集合，因此支持数理逻辑中的all和any运算。内置函数all()和any()实现了这两种扩展逻辑运算，其语法为：

```
all(iterable)
any(iterable)
```

all()的行为是：仅当通过iterable参数传入的可迭代对象中的所有元素逻辑值检测结果都为True时才返回True，否则返回False。any()的行为是：只要通过iterable参数传入的可迭代对象中有一个元素逻辑值检测结果为True就返回True，否则返回False。请看下面的例子：

```
>>> all([0, 1, 2])
False
>>> any([0, 1, 2])
True
>>> all('abc')
True
>>> any('abc')
True
>>> all({None, 0j, False})
False
>>> any({None, 0j, False})
False
>>>
>>> all({1: None, 2: True, 3: False})
True
>>> any({1: None, 2: True, 3: False})
True
>>> all({1: None, 2: True, 3: False}.values())
False
>>> any({1: None, 2: True, 3: False}.values())
True
>>> all({1: None, 2: True, 3: False}.items())
True
>>> any({1: None, 2: True, 3: False}.items())
True
>>> all({})
True
>>> any({})
False
>>>
```

该例子说明了对于长度为0的可迭代对象，all()返回True，any()返回False。

---

可以通过内置函数max()和min()分别取得可迭代对象的最大元素和最小元素，其语法为：

```
max(iterable, *, key=None[, default])
min(iterable, *, key=None[, default])
```

其中可选参数key与在list.sort()中含义相同，可传入仅接收一个参数的回调函数以生成用于比较的键，如果传入None则以元素本身进行比较。iterable参数被传入的可迭代对象必须满足基于其元素用key参数指定的方法生成的键相互间可以用<比较。可选的default参数可被传

入任何对象，当iterable参数被传入的可迭代对象长度为0时该对象会被作为默认值返回。如果省略了default参数，则会在上述情况下抛出ValueError异常。下面是一些例子：

```
>>> max([1, 2, 3])
3
>>> min([1, 2, 3])
1
>>> max('abc')
'c'
>>> min('abc')
'a'
>>> max({'apple', 'pear', 'banana'}, key=len)
'banana'
>>> min({'apple', 'pear', 'banana'}, key=len)
'pear'
>>> max(range(0), default=-1)
-1
>>> min(range(0), default=-1)
-1
>>>
```

需要说明的是，max()和min()还有第二种语法：

```
max(arg1, arg2, *args, key=None)
min(arg1, arg2, *args, key=None)
```

可以这样理解这种语法：通过参数arg1、arg2和args传入的表达式被在内部转换为一个元组，然后按照第一种语法进行处理。由于元组长度至少为2，所以不需要default参数。下面是用这种语法对之前例子的改写：

```
>>> max(1, 2, 3)
3
>>> min(1, 2, 3)
1
>>> max('a', 'b', 'c')
'c'
>>> min('a', 'b', 'c')
'a'
>>> max('apple', 'pear', 'banana', key=len)
'banana'
>>> min('apple', 'pear', 'banana', key=len)
'pear'
>>>
```

---

内置函数sorted()将list.sort()的功能扩展到了所有可迭代对象，其语法为：

```
sorted(iterable, /, *, key=None, reverse=False)
```

它的行为模式与list.sort()完全相同，只不过iterable参数可以被传入任何可迭代对象，且该排序不会作用于原对象，而是返回一个排好序的列表。下面用集合改写第11章中关于list.sort()的例子：

```
>>> st ={'apple', 'Banana', 'PEAR', 'oRange', 'potato'}
>>> sorted(st)
['Banana', 'PEAR', 'apple', 'oRange', 'potato']
>>> max(range(0), default=None)
>>> st ={'apple', 'Banana', 'PEAR', 'oRange', 'potato'}
>>> sorted(st)
['Banana', 'PEAR', 'apple', 'oRange', 'potato']
>>> sorted(st, reverse=True)
['potato', 'oRange', 'apple', 'PEAR', 'Banana']
>>> sorted(st, key=str.lower)
['apple', 'Banana', 'oRange', 'PEAR', 'potato']
>>> sorted(st, key=str.swapcase)
['apple', 'oRange', 'potato', 'Banana', 'PEAR']
>>> def f(ele):
...     return ele[1:]
...
>>> sorted(st, key=f)
['PEAR', 'oRange', 'Banana', 'potato', 'apple']
>>>
```

---

对于仅包含数字的可迭代对象，可以通过内置函数sum()对这些数字求合，其语法为：

```
sum(iterable, /, start=0)
```

其行为是对iterable参数传入的可迭代对象包含的数字求合，并加上start参数传入的数字。下面是一些例子：

```
>>> sum(range(10))
45
>>> sum(range(10), start=1)
46
>>> sum({0.9, 2, 7.5-8j})
(10.4-8j)
>>> sum({0.9, 2, 7.5-8j}, start=2+0.4j)
(12.4-7.6j)
>>>
```

注意sum()的执行过程可以被这样抽象：假设可迭代对象包含的数字是$x_0, x_1, ..., x_n$，则首先计算$s_0 = start + x_0$，再计算$s_1 = s_0 + x_1$，……，依此类推，最后计算$s_n = s_{n-1} + x_n$。这是对可迭代对象进行“降维（reduce）”操作的一个例子。

但并非只有包含数字的可迭代对象可以降维，降唯也不一定要用加法，而可以通过任何二元运算或接受2个参数的函数实现。标准库中的functools模块提供了reduce()函数以实现任意降维操作，其语法为：

```
functools.reduce(callback, iterable[, initializer])
```

其中callback参数需被传入一个接受2个参数的回调函数。initializer参数可以被传入任意对象，如果未被省略，则当iterable参数被传入的可迭代对象非空时该对象扮演sum()中的start参数的角色，而在iterable参数被传入空可迭代对象时该对象作为默认值被返回。如果省略了initializer参数，则当iterable参数被传入空可迭代对象时会抛出TypeError异常，当被传入的可迭代对象仅包含一个元素时返回该元素。下面是一个例子：

```
>>> import functools
>>> functools.reduce(str.strip, ['abcdefg', 'a', 'b', 'c'])
'defg'
>>> functools.reduce(str.strip, ['abcdefg'])
'abcdefg'
>>> functools.reduce(str.strip, ['abcdefg', 'a', 'b', 'c'], 'acxyzbg')
'xyz'
>>> functools.reduce(str.strip, [], 'acxyzbg')
'acxyzbg'
>>> functools.reduce(str.strip, {'abcdefg', 'a', 'b', 'c'})
''
>>> functools.reduce(str.strip, {'abcdefg', 'a', 'b', 'c'}, 'acxyzbg')
'xyz'
>>>
```

该例子说明了降维的结果可能与元素的排列顺序有关（取决于callback参数），此时传入非序列的可迭代对象得到的结果通常不可预测。

---

内置函数filter()的功能是按照指定的规则筛选可迭代对象中的元素，其语法为：

```
filter(callback, iterable)
```

callback参数应被传入仅接受一个参数的回调函数，它会被传入可迭代对象中的元素，并根据某种规则返回True或False。而filter()会返回一个迭代器，对其进行迭代将得到由iterable参数传入的可迭代对象中使得该回调函数返回True的所有元素。特别的，如果callback参数被传入None，则筛选标准是对元素本身进行逻辑值检测，保留逻辑值检测结果为True的元素。下面是一些例子：

```
>>> iterator = filter(lambda x: len(x)>5, {'apple', 'pear', 'banana',
'cherry'})
>>> for s in iterator:
```

```
...     print(s)
...
cherry
banana
>>> iterator = filter(lambda n: n%3 == 0, range(10))
>>> for n in iterator:
...     print(n)
...
0
3
6
9
>>> iterator = filter(None, [0.0, 1.0, None, True, False])
>>> for obj in iterator:
...     print(obj)
...
1.0
True
>>>
```

---

内置函数map()的功能是将若干可迭代对象中的元素的组合映射到另一个元素，其语法为：

```
map(callback, *iterable)
```

callback参数被传入的回调函数的形式参数个数必须正好等于iterable参数被传入的可迭代对象的个数，且它不能小于2。该回调函数会被多次调用，每次调用时都会依次迭代传入的可迭代对象，并以取得的元素作为参数，而其返回值即映射到的对象。这一过程会持续到至少一个可迭代对象抛出StopIteration异常为止。下面是一些例子：

```
>>> def f(*args):
...     return ''.join(args)
...
>>> iterator = map(f, 'abc', 'def', 'ghi', 'xyz')
>>> for s in iterator:
...     print(s)
...
adgx
behy
cfiz
>>>
>>> iterator = map(lambda x: x**3, range(3))
>>> for n in iterator:
...     print(n)
...
0
1
8
>>>
>>> iterator = map(lambda x, y: [x, y], range(4), 'abcdefgh')
>>> for item in iterator:
...     print(item)
...
[0, 'a']
```

```
[1, 'b']
[2, 'c']
[3, 'd']
>>>
```

---

最后，在第7章已经提到了，IOBase支持迭代器协议，所以任何文件对象都是可迭代对象。假设file是个文件对象，则每迭代一次file等价于调用一次file.readlines(1)。请还原第7章中创建的test1.txt和test2.txt，然后通过如下命令行和语句验证这一技巧：

```
$ python3

>>> import sys
>>> fd = open('test1.txt')
>>> for line in fd:
...     sys.stdout.writelines(line)
...
Python is fun!

Insert a line is not easy...

Life is short, you need Python.
>>> with open('test2.txt') as fd:
...     for line in fd:
...         sys.stdout.writelines(line)
...
你好

こんにちは

Hello

Bonjour

GutenTag

привет

Ciao
>>>
```

## 12-7. 映射的只读代理

（标准库：types）

第3章和第5章提到过，当通过元类创建类时，类对象的__dict__最终会引用变量字典的一个只读代理，事实上该只读代理是一个MappingProxyType对象。第11章则介绍过，通过dict.items()、dict.keys()和dict.values()获得的视图都具有mapping属性，该属性同样引用一个MappingProxyType对象，被视为原字典的一个只读代理。

MappingProxyType是types模块定义的一个类，其实例化语法为：

```
class types.MappingProxyType(mapping)
```

其中mapping参数是一个映射，而新创建的MappingProxyType实例将成为该映射的只读代理。

顾名思义，只读代理过滤掉了映射本身支持的写操作，仅剩下读操作。具体来说，我们可以通过映射的只读代理执行下列9种操作：

- 通过“|”实现映射的合并。
- 通过方括号实现该映射的抽取和切片。
- 通过in和not in判断指定的键是否在该映射中。
- 通过len()取得该映射中键值对的数量。
- 通过iter()取得一个迭代器，进而正序遍历该映射的所有键。
- 通过reversed()取得一个迭代器，进而逆序遍历该映射的所有键。
- 支持get()属性。
- 支持copy()属性。
- 支持items()、keys()和values()是属性。

下面通过一个例子来说明如何使用映射的只读代理：

```
$ python3

>>> import types
>>> d1 = {'a': 0, 'b': 1}
>>> d2 = {'b': 2, 'c': 3}
>>> proxy1 = types.MappingProxyType(d1)
>>> proxy2 = types.MappingProxyType(d2)
>>> proxy1
mappingproxy({'a': 0, 'b': 1})
>>> proxy2
mappingproxy({'b': 2, 'c': 3})
>>> proxy1 | proxy2
{'a': 0, 'b': 2, 'c': 3}
>>> proxy2 | proxy1
{'b': 1, 'c': 3, 'a': 0}
>>> 'a' in proxy1
True
>>> 'b' not in proxy2
False
>>> proxy1['b']
1
>>> proxy2['b']
2
>>> len(proxy1)
2
>>> for k in proxy1:
...     print(k)
...
a
b
```

```
>>> for k in reversed(proxy2):
...     print(f"{k}: {proxy2[k]}")
...
c: 3
b: 2
>>> proxy1.get("d", 4)
4
>>> proxy2.copy()
{'b': 2, 'c': 3}
>>> proxy1.items()
dict_items([('a', 0), ('b', 1)])
>>> proxy1.keys()
dict_keys(['a', 'b'])
>>> proxy1.values()
dict_values([0, 1])
>>> proxy1.items().mapping
mappingproxy({'a': 0, 'b': 1})
>>> proxy1.items().mapping == proxy1
True
>>> proxy1.items().mapping is proxy1
False
>>>
```

一个映射的只读代理与它的视图具有如下共通性：只读代理也只是一种对映射的受限访问方式，如果映射的内容发生变化，则该变化会实时反映到只读代理中。请继续上面的例子，通过如下语句验证：

```
>>> d1['b'] = -1
>>> proxy1
mappingproxy({'a': 0, 'b': -1})
>>> d2['e'] = -5
>>> proxy2
mappingproxy({'b': 2, 'c': 3, 'e': -5})
>>> del d2['b']
>>> proxy2
mappingproxy({'c': 3, 'e': -5})
>>>
```