第11章. 容器

有很多内置类型可被视为一个容器，而标准库中的collections模块则定义了更多的容器类型。第10章讨论的字符串、字节串和字节数组分别是字符和字节的容器，内存视图则是抽象出的数组中元素的容器。但在第10章并没有讨论那些类型作为容器的特性，这将在本章和其他被视为容器的内置类型一起讨论（本书不详细讨论collections模块）。

11-1. 序列的典型例子

（教程：4.3、5.1、5.3）
（语言参考手册：3.2、6.2.3、6.2.5、6.15）
（标准库：内置函数、内置类型）

第3章已经说明，容器类型又被分为三类：序列、映射和集合类型。字符串和二进制序列都属于序列，此外属于序列的内置类型还包括元组、列表和范围。本节详细讨论后三种序列。

元组与字符串和字节串类似，是一种不可变的序列。内置类型tuple表示元组。

可以通过内置函数tuple()创建元组，其语法为：

```
class tuple([iterable])
```

其中iterable参数需要是一个可迭代对象（会在第12章解释），该对象会被迭代，依次得到的对象会成为新建元组的元素。如果省略了iterable参数，则得到一个空元组。下面是一些例子：

```
$ python3

>>> tuple()
()
>>> tuple('abc')
('a', 'b', 'c')
>>> tuple(b'1234')
(49, 50, 51, 52)
>>> tuple(bytearray(5))
(0, 0, 0, 0, 0)
>>> tuple((1, 'a', True))
(1, 'a', True)
>>> tuple([[1, 2], 9, 10])
([1, 2], 9, 10)
>>> tuple({0, None, NotImplemented})
(0, None, NotImplemented)
>>>
```

创建元组的另一种方式是使用“表达式列表（expression lists）”，即用逗号将若干子表达式连接在一起形成的表达式。下面是一些例子：

```
>>> 49, 50, 51, 52
(49, 50, 51, 52)
>>> 'a', 'b', 'c'
('a', 'b', 'c')
>>> 1, 'a', True
(1, 'a', True)
>>> 0, None, NotImplemented,
(0, None, NotImplemented)
>>> 1+2^2, 23.6/57.1, 9<<2
(1, 0.4133099824868652, 36)
>>> b'X',
(b'X',)
>>>
```

就像其他表达式一样，可以用圆括号将表达式列表括起来，不影响其结果（注意圆括号中什么也没有将导致创建一个空元组）：

```
>>> (0,)
(0,)
>>> (object(), None, type(tuple))
(<object object at 0x106ed8850>, None, <class 'type'>)
>>> ()
()
>>>
```

事实上，当一个元组被显示时，会自动采用圆括号括起来的表达式列表这种格式，被称为“括号格式（parenthesized forms）”。PEP 8推荐给表达式列表内的每个“,”后面都添加一个空格，但前面不要添加空格；另外，作为表达式列表最后一个字符的“,”的后面不要添加空格。

从上面的例子可以看出，元组与字符串和字节串的本质区别在于不限制元素为字符或字节，且允许元素分别属于不同的类。这意味着一个元组的元素可以是另一个元组，也就是说可以构造出类似其他编程语言（例如C）中的高维数组的结构，例如：

```
>>> ((1, 2, 3), (4, 5, 6), (7, 8, 9))
((1, 2, 3), (4, 5, 6), (7, 8, 9))
>>> ((('a'), ('b', 'c')), ('d', 'e'), ('f',))
((('a', ('b', 'c')), ('d', 'e'), ('f',))
>>>
```

列表就是元组的可变版本，它们之间的关系类似于字节数组和字节串之间的关系。内置类型list表示列表。

可以通过内置函数list()创建列表，其语法为：

```
class list([iterable])
```

同样，iterable参数需要是一个可迭代对象，而产生列表的方式与产生元组相同。如果省略了iterable参数，则得到一个空列表。下面的例子改写自创建元组的例子：

```
>>> list()
[]
>>> list('abc')
['a', 'b', 'c']
>>> list(b'1234')
[49, 50, 51, 52]
>>> list(bytearray(5))
[0, 0, 0, 0, 0]
>>> list((1, 'a', True))
[1, 'a', True]
>>> list([(1, 2), 9, 10])
[(1, 2), 9, 10]
>>> list({0, None, NotImplemented})
[0, None, NotImplemented]
>>>
```

我们也可以通过表达式列表创建列表，只需要用方括号将表达式列表括起来。下面的例子同样改写自创建元组的例子：

```
>>> [49, 50, 51, 52]
[49, 50, 51, 52]
>>> ['a', 'b', 'c']
['a', 'b', 'c']
>>> [1, 'a', True]
[1, 'a', True]
>>> [0, None, NotImplemented,]
[0, None, NotImplemented]
>>> [1+2^2, 23.6/57.1, 9<<2]
[1, 0.4133099824868652, 36]
>>> [b'X',]
[b'X',]
>>> [0,]
[0]
>>> [object(), None, type(tuple)]
[<object object at 0x106ed8850>, None, <class 'type'>]
>>> []
[]
>>>
```

注意与圆括号不同，方括号不被用于算术运算和位运算的表达式，因此其内没有“,”也不会产生混淆，例如“[0,]”也可以写作“[0]”。当一个列表被显示时，会自动采用方括号括起来的表达式列表这种格式，被称为“列表显示（list displays）”。

同样，列表也不限制元素为字符或字节，且允许元素分别属于不同的类，包括另一个列表。下面是用列表构造出的类似高维数组的结构的例子：

```
>>> [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> [['a'], ['b', 'c']], ['d', 'e'], ['f',]]
[['a'], ['b', 'c']], ['d', 'e'], ['f']]
>>>
```

范围像元组一样也是一种不可变序列。

它与元组的区别是：

- 所有元素都必须是整数或具有__index__魔术属性的对象。
- 元素是排好序的。
- 元素间的相互间隔固定，被称为“步长（step）”。

换句话说，范围指的是一个有限长的整数等差数列。这使得存储范围时只需要储存三个整数，即等差数列的首元素、尾元素和步长，访问其他元素时都是通过等差数列的计算公式推算出来的。因此范围是计算机科学中以时间换取空间的例子。

范围只能通过内置函数range()创建，该函数的第一种语法为：

```
class range(start, stop, step=1)
```

也就是从start开始（包括），到stop为止（不包括），步长为step的一个等差数列，例如：

```
>>> r = range(1, 10)
>>> list(r)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> r = range(1, 10, 3)
>>> list(r)
[1, 4, 7]
>>> r = range(2, -5, -2)
>>> list(r)
```

```
    [2, 0, -2, -4]
    >>>
```

注意step参数不能是0，因为步长为0会得到无限长的数列；step参数为正整数时，等差数列是升序的；step参数为负整数时，等差数列是降序的。

range()的第二种语法为：

```
class range(stop)
```

这其实是第一种语法的简写，隐含start=0和step=1，例如：

```
>>> tuple(range(5))
(0, 1, 2, 3, 4)
>>> tuple(range(-1))
()
>>>
```

范围没有直观的显示方法，会被显示为一次对range()的调用，例如：

```
>>> range(1, 10)
range(1, 10)
>>> range(10)
range(0, 10)
>>> range(1, 10, 2)
range(1, 10, 2)
>>>
```

最后值得一提，collections模块定义了“具名元组（named tuple）”和deque（双端队列）这两种序列类型，前者是元组的扩展，后者是列表的扩展。

11-2. 序列的操作

（教程：3.1.2、3.1.3、5.1.1、5.1.2、5.8）
（语言参考手册：3.2、6.3.2、6.3.3、6.10.1、6.10.2）
（标准库：内置函数、内置类型）

第10章已经说明字符串、字节串和字节数组支持通过“+”、“*”、“+=”和“*=”实现拼接。事实上拼接也适用于元组和列表，但不适用于范围。请看下面的例子：

```
>>> t = (0, 1) + (2,)
>>> t
(0, 1, 2)
>>> l = ['a'] * 3
```

```
>>> l
['a', 'a', 'a']
>>> t *= 2
>>> t
(0, 1, 2, 0, 1, 2)
>>> l += [None]
>>> l
['a', 'a', 'a', None]
>>>
```

进行拼接时元组和列表不能混合，就像字符串不能与字节串混合一样。（能够在拼接中混合的只有字节串和字节数组，而最终得到字节数组。）

容器类型对象之所以被称为“容器”，是因为它们容纳了其他一些对象作为元素。而一个容器的长度就是其元素的个数，这可以通过内置函数len()取得，其语法为：

```
len(collection)
```

其中collection参数需被传入一个求值结果为容器的表达式。这会导致collection.__len__被调用。所有容器类型都实现了__len__魔术属性。下面是取得序列长度的例子：

```
>>> len('早上好')
3
>>> len(bytearray('早上好', 'utf8'))
9
>>> len((0, 1, 2))
3
>>> len([-13, [None, (2.5, 0)]])
2
>>> len(range(10))
10
>>>
```

注意这些例子说明了如下三点：

➤ 字符串的长度等于其字符数，并不一定等于其字节数，取决于字符编码方式；而字节串/字节数组的长度总是等于其字节数，与字符编码方式无关。
➤ 元组和/或列表相互嵌套形成复杂结构时，其长度只计算最外层元组/列表容纳的对象个数，不需要递归地计算内层元组/列表容纳的对象个数。
➤ 范围虽然只储存了等差数列的两个端点，但其长度等于等差数列的项数。

序列支持通过“抽取（subscription）”操作提取出其内元素，而这是通过如下语法实现的：

```
seq[index]
```

其中seq是一个求值结果为序列的表达式，而index则是一个求值结果为整数（或其他具有__index__的对象）的表达式。这会导致seq.__getitem__(index)被调用，而所有序列类型都实现了__getitem__魔术属性。PEP 8建议索引和方括号之间不要插入空格。

要想成功抽取一个序列的元素必须给出有效的索引。序列s的索引的有效范围是-len(s) ~ len(s)-1：当索引为非负整数时，将从序列的第一个元素向后查找，而第一个元素的索引为0；当索引为负整数时，将从序列的最后一个元素向前查找，而最后一个元素的索引为-1。如果给出的索引超出了有效范围，则会抛出IndexError异常。

下面是对序列执行抽取操作的一些例子：

```
>>> "abc"[0]
'a'
>>> "abc"[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> b'x9@p0'[4]
48
>>> bytearray([0, 1, 2, 3, 4, 5])[-6]
0
>>> (1, 2, 3)[0]
1
>>> [NotImplemented][0]
NotImplemented
>>> range(10)[0]
0
>>> range(10)[9]
9
>>> l = [(1, 1), (2, 4), (3, 9), (4, 16)]
>>> l[3]
(4, 16)
>>> l[3][0]
4
>>> l[3][1]
16
>>> t = ((('a',), 'b'), 'c', range(2))
>>> t[0]
(('a',), 'b')
>>> t[0][0]
('a',)
>>> t[0][0][0]
'a'
>>> t[2][0]
0
>>> t[2][1]
1
>>>
```

上面的例子还说明了很重要的一点：元组和/或列表相互嵌套形成复杂结构时，可以通过连续的多次抽取操作取得任意层的元素。这与在其他编程语言（例如C）中访问高维数组的方法是相同的。

抽取本质上是一种读操作，而可变序列还支持对其内元素进行写操作，即设置和删除。而这是从抽取操作的语法扩展出来的，即：

```
seq[index] = value
del seq[index]
```

设置操作的语法将导致seq.__setitem__(index, value)被调用，而删除操作的语法将导致seq.__delitem__(index)被调用。所有可变序列都实现了这两个魔术属性。下面是一些例子：

```
>>> ba = bytearray([0, 1, 2])
>>> ba[0] = 3
>>> ba
bytearray(b'\x03\x01\x02')
>>> del ba[1]
>>> ba
bytearray(b'\x03\x02')
>>> l = [0, 1, 2]
>>> l[0] = 3
>>> l
[3, 1, 2]
>>> del l[1]
>>> l
[3, 2]
>>>
```

序列的索引隐含一种顺序关系（索引的有效范围是一个步长为1的等差数列），这导致可以对序列执行一种其特有的操作——“切片（slicing）”。

切片是通过“切片对象（slice objects）”实现的，该类对象可通过内置函数slice()创建，其第一种语法为：

```
class slice(start, stop, step=1)
```

第二种语法为：

```
class slice(stop)
```

可以看出，slice()的语法与range()完全相同，所以切片对象本质上也是一个等差数列。然而切片对象代表的不是序列中的元素，而是序列的索引，或者更准确的说是序列索引的一个掩码。

切片本质上是抽取的变体，即以一个切片对象作为参数扉用一个序列的__getitem__魔术属性，抽取出所有索引在该切片对象中的元素以形成一个子序列。下面是切片的一些例子：

```
>>> sl = slice(1, 10, 2)
>>> 'abcde'.__getitem__(sl)
'bd'
>>> b'789'.__getitem__(sl)
b'8'
>>> (None, NotImplemented, Ellipsis).__getitem__(sl)
(NotImplemented,)
>>> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13].__getitem__(sl)
[2, 4, 6, 8, 10]
>>> range(10).__getitem__(sl)
range(1, 10, 2)
>>> range(0, -20, -3).__getitem__(sl)
range(-3, -21, -6)
>>>
```

切片对象本身具有下列非函数属性：

- start：引用通过start参数传入的整数。如果start参数被省略则引用None。
- stop：引用通过stop参数传入的整数。
- step：引用通过step参数传入的整数。如果step参数被省略则引用None。

显然，一个切片对象的功能可以通过这三个属性完全确定。此外，切片对象还提供了函数属性indices()，其语法为：

```
slice.indices(length)
```

调用该函数属性将返回一个三元组，可以视为调用slice()时给出的实际参数列表，其含义是将该切片对象应用于长度为length的序列时等价的最短切片对象。下面的例子说明了这些属性的作用：

```
>>> sl = slice(1, 7, 2)
>>> sl.start
1
>>> sl.stop
7
>>> sl.step
2
>>> sl.indices(1)
(1, 1, 2)
```

```
>>> [1].__getitem__(sl)
[]
>>> sl.indices(2)
(1, 2, 2)
>>> [1, 2].__getitem__(sl)
[2]
>>> sl.indices(3)
(1, 3, 2)
>>> [1, 2, 3].__getitem__(sl)
[2]
>>> sl.indices(4)
(1, 4, 2)
>>> [1, 2, 3, 4].__getitem__(sl)
[2, 4]
>>>
```

在执行切片操作时并非必须显式创建切片对象。Python支持将抽取操作的语法扩展为：

```
seq[[start]:[stop][:[step]]]
```

注意除了最外层的方括号外，其余方括号都表示其内的部分是可省略的。该语法会自动调用slice()创建一个切片对象，然后以它为参数调用seq.__getitem__()。被省略的部分会按照如下规则取默认值：

- start：默认取值0。
- stop：默认取值序列长度。
- step：默认取1。

也就是说，seq[:]和seq[::]都将得到原序列的一个拷贝。此外，由于会先创建一个切片对象，所以start和stop不受索引的有效范围限制。下面是一些例子：

```
>>> s = 'abcdefgh'
>>> s[:]
'abcdefgh'
>>> s[::]
'abcdefgh'
>>> s[:3]
'abc'
>>> s[:11]
'abcdefgh'
>>> s[2:6]
'cdef'
>>> s[2:6:2]
'ce'
>>> s[:6:2]
'ace'
>>>
>>> b = b'@#$%&'
>>> b[:]
b'@#$%&'
```

```
>>> b[:1]
b'@'
>>> b[1:]
b'#$%&
>>> b[::2]
b'@$&'
>>>
>>> t = (1, 2, 3, 4, 5)
>>> t[::]
(1, 2, 3, 4, 5)
>>> t[:2]
(1, 2)
>>> t[1:2]
(2,)
>>> t[0:3:2]
(1, 3)
>>>
>>> r = range(10)
>>> r[:]
range(0, 10)
>>> r[:5]
range(0, 5)
>>> r[3:9]
range(3, 9)
>>> r[3:9:3]
range(3, 9, 3)
>>>
```

注意这种形式的切片有个特点，即seq[:i]和seq[i:]恰好将原序列基于第i个元素分为两部分，该元素本身属于后半部分。PEP 8建议切片中的“:”或者两侧都不插入空格，或者两侧都插入一个空格，但“::”中间总是不插入空格。

易知，对于可变序列来说，Python支持将抽取操作的语法扩展为：

```
seq[[start]:[stop][:[step]]] = iterable
del seq[[start]:[stop][:[step]]]
```

上面一行表示通过迭代一个可迭代对象获得一个对象序列，然后用这些对象替换掉可变序列中的指定切片。下面一行则表示删除可变序列中的指定切片。下面是一些例子：

```
>>> l = [1, 2, 3, 4, 5]
>>> ba = bytearray(l)
>>> l[2:4] = 'abcde'
>>> l
[1, 2, 'a', 'b', 'c', 'd', 'e', 5]
>>> ba[::2] = b'xyz'
>>> ba
bytearray(b'x\x02y\x04z')
>>> del l[0:2]
>>> l
['a', 'b', 'c', 'd', 'e', 5]
>>> del ba[1:4:2]
>>> ba
bytearray(b'xyz')
```

```
>>> l[len(l)-2:] = []
>>> l
['a', 'b', 'c', 'd']
>>> ba[:] = b''
>>> ba
bytearray(b'')
>>>
```

容器类型的本质决定了它们必须支持成员检测，也就是in和not in比较运算，这在第3章已经提到了。

所有容器类型都实现了魔术属性__contains__，“element in collection”等价于调用collection.__contains__(element)，而“element not in collection”等价于表达式“not collection.__contains__(element)”。易知，__contains__返回一个布尔值，True表示指定对象是指定容器的元素。

对于序列来说，__contains__魔术属性的规则是这样的：

➤ 对于字符串、字节串和字节数组，__contains__返回True意味着element是collection的子串/子字节串/子字节数组。注意空串/空字节串/空字节数组是任何字符串/字节串/字节数组的元素。
➤ 对于其他类型的序列来说，__contains__返回True意味着存在一个index，使得表达式“element is collection[index] or element == collection[index]”的值是True。这意味着数字比较时所遵循的数学规则在成员检测运算中依然有效。

下面是一些例子：

```
>>> 'ab' in 'abc'
True
>>> 'ac' in 'abc'
False
>>> b'' not in b'xyz'
False
>>> b'x' in bytearray(b'xyz')
True
>>> 1 in (1.0, 2.0)
True
>>> False in [0, None, NotImplemented]
True
>>> 99 in range(0, 100, 2)
False
>>> (1,) not in [(1,), (2, 3)]
False
>>> (1,) not in [((1,), 2), 3]
True
>>>
```

此外，序列类型重写了魔术属性__lt__、__le__、__eq__、__ne__、__gt__和__ge__，使得它们相互间可以按照如下规则进行比较：

➤ 进行==或!=比较时，首先判断它们的类型是否相同，然后判断它们的长度是否相同，最后再比较对应位置的元素是否相等。仅当类型相同、长度相同、所有对应位置的元素都相等时才判断两个序列相等，其他情况都判断两者不等。
➤ 仅当它们类型相同且不是范围时才能进行<、>、<=和>=比较，此时按照字典排序算法进行比较。

下面是一些例子（字符串、字节串和字节数组的比较在第10章已经说明）：

```
>>> (1, 2) == [1, 2]
False
>>> (1, 2) == (1, 2, 3)
False
>>> (1, 2) == (2, 3)
False
>>> (1, 2) < (1, 2, 3)
True
>>> (1, 2) < (-1, -2, -3)
False
>>> ['a', ['b', 'c']] >= ['a', ['a'], 'b']
True
>>>
```

最后，所有序列都具有表11-1列出的属性。

表11-1. 序列的通用属性

属性	说明
seq.count()	返回指定元素在序列中的出现次数。
seq.index()	返回指定元素在序列指定区域内首次出现的位置，正向搜索。

seq.count()的语法为：

```
seq.count(x)
```

它返回一个非负整数，为对象x在序列seq中出现的次数。注意在第10章讨论了str.count()、bytes.count()和bytearray.count()，它们是seq.count()的扩展。下面是关于seq.count()的一个例子：

```
>>> (1, 0, 1, 1, 0, 1, 0, 1).count(1)
5
>>> (1, 0, 1, 1, 0, 1, 0, 1).count(0)
3
>>> (1, 0, 1, 1, 0, 1, 0, 1).count(1.0)
5
>>> (1, 0, 1, 1, 0, 1, 0, 1).count(0j)
3
>>> (1, 0, 1, 1, 0, 1, 0, 1).count('1')
0
>>> (1, 0, 1, 1, 0, 1, 0, 1).count('0')
0
>>>
```

seq.index()的语法为：

```
seq.index(x, start=0, end=None)
```

按照索引从start（包括）到end（不包括）的顺序搜索x，如果找到则返回x的索引，没有找到则抛出ValueError异常。第10章讨论的str.index()、bytes.index()和bytearray.index()同样是seq.index()的扩展。下面是关于seq.index()的一个例子：

```
>>> (1, 0, 1, 1, 0, 1, 0, 1).index(0)
1
>>> (1, 0, 1, 1, 0, 1, 0, 1).index(0, 2)
4
>>> (1, 0, 1, 1, 0, 1, 0, 1).index(0, 5, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: tuple.index(x): x not in tuple
>>>
```

而所有可变序列则都具有表11-2列出的属性。

表11-2. 可变序列的通用属性

属性	说明
seq.append()	将指定的一个元素添加到序列的末尾。
seq.extend()	将指定的多个元素添加到序列的末尾。
seq.insert()	将指定的一个元素插入到序列的指定位置。
seq.remove()	从序列中删除指定元素，但仅删除它第一次出现的位置。
seq.clear()	清空序列。
seq.pop()	从序列的指定位置删除一个元素，并将其返回。
seq.copy()	返回序列的一个拷贝。

属性	说明
seq.reverse()	逆转序列中元素的顺序。

seq.append()和seq.extend()用于给可变序列追加元素，其语法为：

```
seq.append(x)
seq.extend(iterable)
```

seq.append()会将通过x参数传入的单个元素追加到序列末尾；seq.extend()则会迭代通过iterable参数传入的可迭代对象，并依次将得到的元素追加到序列末尾。下面是一个例子：

```
>>> l = []
>>> l.append('a')
>>> l
['a']
>>> l.append('b')
>>> l
['a', 'b']
>>> l.extend('cd')
>>> l
['a', 'b', 'c', 'd']
>>>
```

seq.insert()用于将新元素插入到可变序列的指定位置，其语法为：

```
seq.insert(index, x)
```

其功能是使x成为seq的第index个元素，而seq原来的索引大于等于index的元素都将各自的索引增加1。注意这等价于“seq[index:index] = [x]”，也就是说替换一个空的切片等同于插入操作。下面是一个例子：

```
>>> ba = bytearray([1, 2, 3, 4])
>>> ba
bytearray(b'\x01\x02\x03\x04')
>>> ba.insert(1, 5)
>>> ba
bytearray(b'\x01\x05\x02\x03\x04')
>>>
```

seq.remove()和seq.clear()用于删除可变序列中的元素，其语法为：

```
seq.remove(x)
seq.clear()
```

seq.remove()的行为是从索引0开始按升序搜索，如果找到了某个元素与x之间进行==比较时得到True则删除该元素，否则什么也不做。seq.clear()则删除可变序列中的所有元素。下面是一些例子：

```
>>> l = [1, 0, 0, 1, 1, 1, 0]
>>> l.remove(1)
>>> l
[0, 0, 1, 1, 1, 0]
>>> l.remove(1.0)
>>> l
[0, 0, 1, 1, 0]
>>> l.remove(0j)
>>> l
[0, 1, 1, 0]
>>> l.clear()
>>> l
[]
>>>
```

seq.pop()是专门为通过可变序列实现栈和队列而引入的，其语法为：

```
seq.pop(index=-1)
```

其功能是从可变序列中删除第index个元素，并将该元素返回。为了实现栈或队列，我们需要联合使用seq.append()和seq.pop()。栈的特点是“先进后出”，即最先插入的元素最后弹出，因此应调用“seq.pop(-1)”（-1可以省略）。下面是一个例子：

```
>>> stack = []
>>> stack.append('a')
>>> stack
['a']
>>> stack.append('b')
>>> stack
['a', 'b']
>>> stack.append('c')
>>> stack
['a', 'b', 'c']
>>> stack.pop()
'c'
>>> stack
['a', 'b']
>>> stack.pop(-1)
'b'
>>> stack
['a']
>>>
```

队列的特点是“先进先出”，即最先插入的元素最先弹出，因此应调用“seq.pop(0)”。下面是一个例子：

```
>>> queue = []
>>> queue.append('a')
>>> queue
['a']
>>> queue.append('b')
>>> queue
['a', 'b']
>>> queue.append('c')
>>> queue
['a', 'b', 'c']
>>> queue.pop(0)
'a'
>>> queue
['b', 'c']
>>> queue.pop(0)
'b'
>>> queue
['c']
>>>
```

当然seq.pop()也并非只能从可变序列的开头和末尾弹出元素，而是可以按随机的顺序弹出元素，例如：

```
>>> inventory = bytearray()
>>> inventory.extend([1, 2, 3, 4, 5, 6])
>>> inventory
bytearray(b'\x01\x02\x03\x04\x05\x06')
>>> inventory.pop(2)
3
>>> inventory
bytearray(b'\x01\x02\x04\x05\x06')
>>> inventory.pop(1)
2
>>> inventory
bytearray(b'\x01\x04\x05\x06')
>>>
```

但需要强调的是，seq.pop()从可变序列的末尾取得元素是速度最快的，而从其他位置取得元素的速度要慢很多，因此它最适合用来实现栈。如果要实现队列，乃至双端队列，推荐使用collections模块定义的deque类型。调用一个空序列的pop()会抛出IndexError异常。

seq.copy()就等价于“seq[:]”或“seq[::]”，返回原序列的一个拷贝。这里需要强调的是，不论调用seq.copy()，还是使用“seq[:]”或“seq[::]”，得到的都是“浅拷贝”，也就是说如果原序列的元素中有属于容器类型的，则拷贝序列中的相应元素引用同一容器对象，而不会递归地拷贝该容器对象。下面的例子说明了这点：

```
>>> old = ['abc', (0.5, -0.5), range(10)]
>>> new = old.copy()
>>> new is old
False
>>> new[0] is old[0]
True
>>> new[1] is old[1]
True
>>> new[2] is old[2]
True
>>>
```

seq.reverse()的作用是将可变序列中元素的排列顺序颠倒。注意它不会返回原序列的拷贝，而是改变原序列自身。下面是一个例子：

```
>>> ba = bytearray([1, 2, 3])
>>> ba
bytearray(b'\x01\x02\x03')
>>> ba.reverse()
>>> ba
bytearray(b'\x03\x02\x01')
>>> ba.reverse()
>>> ba
bytearray(b'\x01\x02\x03')
>>>
```

特别的，列表除了支持可变序列的通用属性之外，还提供了函数属性sort()以实现排序，其语法为：

```
list.sort(*, key=None, reverse=False)
```

其中key参数需被传入一个如下格式的回调函数：

```
def func(ele):
    ... any code goes here ...
```

其中ele会被传入列表中的元素，而返回的对象则被作为对该元素排序时比较用的键。如果ele被传入None，则以该元素本身作为键。当reverse参数被传入False时，会用<来比较两个键，最后的结果为升序；当reverse参数被传入True时，会用>来比较两个键，最后的结果为降序。这意味着，键的类型必须实现了__lt__和__gt__魔术属性（例如数字、字符串）。与seq.reverse()类似，list.sort()不会返回一个新的列表，而会直接改变原列表中元素的排列顺序。下面是一个例子：

```
>>> l = ['apple', 'Banana', 'PEAR', 'oRange', 'potato']
>>> l.sort()
```

```
>>> l
['Banana', 'PEAR', 'apple', 'oRange', 'potato']
>>> l.sort(reverse=True)
>>> l
['potato', 'oRange', 'apple', 'PEAR', 'Banana']
>>> l.sort(key=str.lower)
>>> l
['apple', 'Banana', 'oRange', 'PEAR', 'potato']
>>> l.sort(key=str.swapcase)
>>> l
['apple', 'oRange', 'potato', 'Banana', 'PEAR']
>>> def f(ele):
...     return ele[1:]
...
>>> l.sort(key=f)
>>> l
['PEAR', 'oRange', 'Banana', 'potato', 'apple']
>>>
```

注意该例子用到了两个技巧：

1. 虽然在第10章中介绍str.lower()和str.swapcase()时认为它们没有参数，但实际上作为str类的实例属性它们都默认具有self形式参数，因此符合list.sort()中key参数所需回调函数的要求。
2. 自定义函数f()通过字符串的切片操作取得字符串去掉首字母后剩余的部分。

11-3. 映射的典型例子

（教程：5.5）
（语言参考手册：3.3.1、6.2.7）
（标准库：内置函数、内置类型、sys）

在数学上，映射指的是自变量到因变量的一种对应关系，因此序列也符合这一定义，因为它们将索引映射到了序列中的元素。然而在Python中，映射指的是键到值的对应关系，换句话说，映射类型的对象是键值对的容器。一方面，映射中的键必须具有全局唯一性。另一方面，有时候多个对象在逻辑上应对应同一个键，例如0、0.0和0j。为了同时满足这两方面的要求，映射在内部其实存储的是键的哈希值。这意味着作为键的对象必须能被计算哈希值，也就是“可哈希（hashable）”的。

属于某一类型的对象是可哈希的，当且仅当该类实现了__hash__魔术属性。所有不可变的内置类型的实例都是可哈希的，包括：None、NotImplemented、Ellipsis、所有函数相关对象、以type为元类的类、模块、数字相关对象、字符串、字节串、内存视图以及不直接或间接包含可变类型对象的不可变容器。

如果自定义的类实现了__hash__魔术属性，那么该类的实例也是可哈希的。但自定义的__hash__必须保证同时满足如下要求：

➤ __hash__的返回值应是一个储存所需空间不超过机器字长的整数（即32位或64位）。如果该整数超出了机器字长，则会被截断。
➤ 实现__hash__的类本身或其某个基类必须重写__eq__，且使得如果该类的两个实例用==比较得到True，则它们的哈希值必然相同。

此外，如果该类的实例是可变的，则不能实现__hash__，因为可哈希对象必须是不可变的。因此如果一个类的某个直接基类的实例是可哈希的，而该类自身的实例是可变的，则该类必须重写__hash__并使其引用None。

当访问映射类型对象内的元素时，会自动调用相应键的__hash__。此外，内置函数hash()用于取得一个可哈希对象的哈希值，其语法为：

```
hash(object)
```

这等价于调用object.__hash__()。下面我们做一个实验。首先执行如下命令行和语句：

```
$ python3

>>> hash(True)
1
>>> hash(False)
0
>>> hash(1)
1
>>> hash(0)
0
>>> hash(0.0)
0
>>> hash(0j)
0
>>> hash(106)
106
>>> hash(-34)
-34
>>> hash(4.5)
1152921504606846980
>>> hash(0.9-1.15j)
1037629354349962941
>>> hash('a')
-1502467095553133195
>>> hash(b'a')
-1502467095553133195
>>> hash(memoryview(b'a'))
-1502467095553133195
>>> hash(None)
282019434
>>> hash(NotImplemented)
282019528
>>> hash(Ellipsis)
282020014
>>> hash((1, 2, 3))
529344067295497451
```

```
>>> hash((None, 'a'))
143411235152346124
>>> ^D
```

在重新启动Python解释器后，再次执行上述命令行和语句：

```
$ python3

>>> hash(True)
1
>>> hash(False)
0
>>> hash(1)
1
>>> hash(0)
0
>>> hash(0.0)
0
>>> hash(0j)
0
>>> hash(106)
106
>>> hash(-34)
-34
>>> hash(4.5)
1152921504606846980
>>> hash(0.9-1.15j)
1037629354349962941
>>> hash('a')
6071232610954342405
>>> hash(b'a')
6071232610954342405
>>> hash(memoryview(b'a'))
6071232610954342405
>>> hash(None)
284766314
>>> hash(NotImplemented)
284766408
>>> hash(Ellipsis)
284766894
>>> hash((1, 2, 3))
529344067295497451
>>> hash((None, 'a'))
3118654958176140947
>>>
```

从该实验结果可以得出在默认配置下计算对象哈希值的下述规律：

➤ 数字（包括布尔值）的哈希值不会因Python解释器的重启而改变。
➤ 字符串、字节串、内存视图、以及数字之外的非容器对象的哈希值会因Python解释器的重启而改变。
➤ 字符串、字节串和内存视图之外的容器对象的哈希值是否会因Python解释器的重启而改变，取决于它是否（递归地）包含字符串、字节串、内存视图、或数字之外的非容器对象。

➤ 在Python解释器启动后，字符串、字节串和内存视图的哈希值只与它们所占用内存中的每个位的取值有关，只要这些位的取值相同，不论是字符串、字节串还是内存视图，也不论字符串采用了什么编码方式，得到的哈希值都是相同的。

导致上述现象的原因是：Python解释器在启动时会生成一个0~4294967295范围内的随机整数，当计算字符串、字节串、内存视图、以及数字之外的非容器对象的哈希值时以该随机整数作为盐值，而对于其他容器则基于它们的每个元素的哈希值计算其本身的哈希值。

有些时候，我们需要确保一个非数字对象的哈希值不因Python解释器重启而改变（例如在多个Python解释器进程之间共享哈希值），在这种情况下可设置PYTHONHASHSEED环境变量。PYTHONHASHSEED环境变量用于指定计算字符串、字节串和内存视图的哈希值时使用的盐值。如果该环境变量未设置，或被设置为“random”，则采用随机盐值。如果将其设置为一个0~4294967295范围内的整数，则总是使用该整数为盐值。请通过如下命令行和语句验证（假设你在使用Unix或类Unix操作系统）：

```
$ export PYTHONHASHSEED=1
$ python3

>>> hash('a')
8432517439229126278
>>> hash(b'a')
8432517439229126278
>>> hash(memoryview(b'a'))
8432517439229126278
>>> ^D

$ python3

>>> hash('a')
8432517439229126278
>>> hash(b'a')
8432517439229126278
>>> hash(memoryview(b'a'))
8432517439229126278
>>> ^D

$ export PYTHONHASHSEED=random
$ python3
>>> hash('a')
-4167473033543037746
>>> hash(b'a')
-4167473033543037746
>>> hash(memoryview(b'a'))
-4167473033543037746
>>>
```

事实上，对象的哈希值随机变化有助于提高Python脚本的安全性。然而数字的哈希值默认就是固定的，这是因为必须保证两个任意类型（包括Decimal和Fraction）的数字x和y只要满足x == y，就必然有hash(x) == hash(y)。因此Python解释器使用了一个不需要盐值的算法来计算数字的哈希值，该算法需要对一个很大的素数取余。

计算字符串、字节串和内存视图的哈希值时，Python解释器使用的是一个需要盐值的算法。事实上，计算它们的哈希值时只考虑底层缓冲区的状态。因此同一缓冲区采用不同字符编码方式时可能对应不同字符串，但哈希值是固定不变的，等于相应的字节串的哈希值，也等于绑定到该缓冲区的内存视图的哈希值。

Python解释器计算哈希值的相关参数可以通过表11-3列出的sys属性来查询。

表11-3. 哈希算法相关sys属性

属性	说明
sys.hash_info	一个具名元组，包含哈希算法相关参数，即如下七个元素： width：哈希值的长度（单位是位）。 modulus：用于取余的大素数。 inf：+∞对应的哈希值。 imag：复数中的j转化为的整数。 algorithm：处理字符串、字节串和内存视图使用的哈希算法名称。 seed_bits：种子密钥的长度（单位是位）。

请通过如下语句验证：

```
      >>> import sys
      >>> sys.hash_info
      sys.hash_info(width=64, modulus=2305843009213693951, inf=314159, nan=0,
imag=1000003, algorithm='siphash24', hash_bits=64, seed_bits=128, cutoff=0)
      >>>
```

在明确了什么是对象的哈希值后，让我们回到映射类型本身。第3章已经提到，只有一种属于映射的内置类型——字典。此外，collections模块定义的ChainMap是另一种映射类型，Counter、OrderedDict和defaultdict是字典的子类，然而对它们的讨论超出了本书的范围。下面将集中讨论字典。

内置类型dict表示字典。可以通过内置函数dict()创建字典，其语法为：

```
    class dict([iterable, ]**kwargs)
```

其中iterable参数是一个可迭代对象，而kwargs则是若干个（可以是0个）基于关键字对应的实际参数。该语法概括了多种情况，下面依次讨论。

当iterable参数和kwargs参数都被省略时，将创建一个空字典，例如：

```
>>> dict()
{}
>>>
```

当省略了iterable参数，而kwargs是1个以上的基于关键字对应的实际参数时，每个实际参数将被转化为一个键值对，键即关键字转化成的字符串，值则是实际参数本身。下面是一些例子：

```
>>> dict(x=1.0, y=-2.5)
{'x': 1.0, 'y': -2.5}
>>> dict(name='Bob', age=29, sex='male')
{'name': 'Bob', 'age': 29, 'sex': 'male'}
>>> dict(a0=1, a1=1, a2=2, a3=3, a4=5)
{'a0': 1, 'a1': 1, 'a2': 2, 'a3': 3, 'a4': 5}
>>> dict(_start=0, _stop=None)
{'_start': 0, '_stop': None}
>>>
```

注意以这种方式指定的键值对时，键只能是符合标识符格式要求的字符串。

当iterable参数是一个可迭代对象，而kwargs参数被省略时，新建字典的键值对将完全由该可迭代对象确定。该可迭代对象可以有两种情况。第一种情况是它并非一个映射，但迭代它得到的对象是包含两个元素的序列，该序列的第一个元素将成为键，第二个元素将成为值，例如：

```
>>> d1 = dict([(0, 'zero'), (1, 'one')])
>>> d1
{0: 'zero', 1: 'one'}
>>>
```

第二种情况是该可迭代对象就是一个映射，此时新建字典将与该映射具有相同的键值对，特别的当该映射就是一个字典时，新建字典将是该字典的一个拷贝，例如：

```
>>> d2 = dict(d1)
>>> d2
{0: 'zero', 1: 'one'}
>>> d2 is d1
False
>>>
```

以这种方式指定键值对时，键可以是任意可哈希对象。（但一般而言，我们会选择数字、字符串和字节串作为字典的键，因为它们可以通过字面值方便地给出。）

当iterable参数是一个可迭代对象，而kwargs是1个以上基于关键字对应的实际参数时，可以看成前面两种情况的组合：首先根据可迭代对象创建一个字典，然后将通过基于关键字对

应的实际参数指定的键值对添加到该字典中，如果发生键冲突则后者覆盖前者。下面是一个例子：

```
>>> dict((['x', 0], ['y', 1]), y=2, z=3)
{'x': 0, 'y': 2, 'z': 3}
>>>
```

从上面的例子可以看出，字典被显示时会采用一个大括号结构，大括号内用“,”分隔名值对，而名值对之间则用“:”分隔。该格式被称为“字典显示（dictionary displays）”，它同样可以用于创建字典。下面是一个例子：

```
>>> d3 = {(1, 0): "East", (0, 1): "North", (-1, 0): "West", (0, -1):
"South"}
>>> d3
{(1, 0): 'East', (0, 1): 'North', (-1, 0): 'West', (0, -1): 'South'}
>>>
```

注意该例子中的字典以元组为键，由于这些元组只包含整数，所以是可哈希的。PEP 8推荐字典显示中的“:”前面不要添加空格，后面则添加一个空格。

最后，从上面的讨论可知，除非映射中的键是数字或仅包含数字的容器，否则计算键的哈希值代价较高。为了提高映射中元素的访问速度，所有Python解释器都使用了“字符串驻留（string interning）”技术。

该技术依赖于在内部维护的一个用C实现的数据结构，可以动态增减字符串，并快速查找一个字符串是否在该数据结构中。我们把该数据结构称为“驻留字符串表（interned string table）”。如果一个映射的所有键都在驻留字符串表中，而访问它的元素时使用的键也在驻留字符串表中，就可以直接比较这两个字符串的地址，不需要计算它们的哈希值，能极大提高这些元素的访问速度。

可以通过表11-4列出的sys属性将一个字符串动态插入驻留字符串表，其语法为：

```
sys.intern(string)
```

其中string参数需被传入一个字符串。需要强调的是，该函数会原封不动地返回被传入的字符串，而我们应用某个变量引用它。这是因为驻留字符串表使用的是弱引用（将在第15章讨论），无法增加其内字符串的引用数，而如果不将动态插入的字符串赋值给某个变量，则它会因为引用数为0而被垃圾回收机制销毁。

表11-4. 驻留字符串表相关sys属性

属性	说明
`sys.intern()`	将指定的字符串插入驻留字符串表，然后返回该字符串。

Python脚本中出现的所有标识符和字符串都会被自动插入驻留字符串表，而这些标识符本质上是某个变量字典的键，引用数不为0。这使得访问标识符本质上是比较字符串的地址，不涉及哈希值的计算，所以速度很快。在实际应用中，使用映射的一个小技巧是要么只使用字符串作为键，通过字符串驻留技术避免计算哈希值；要么只使用数字作为键，以较快地计算哈希值；不要使用其他类型的对象作为键，也不要混用字符串和数字。

11-4. 映射的操作

（语言参考手册：6.3.2、6.10.1、6.10.2）
（标准库：内置函数、内置类型、types）

映射并不支持拼接，但字典支持合并，其语法为：

```
dict1 | dict2
```

其中dict1和dict2必须都是字典，合并得到的字典取两者键值对的并集，当发生键冲突时以dict2为准，例如：

```
>>> {"a": 1, "b": 2, "c": 3} | {"d": 4, "e": 5}
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
>>> {b'xyz': (1.0, 0), None: None} | {None: NotImplemented}
{b'xyz': (1.0, 0), None: NotImplemented}
>>>
```

显然，从“|”可以扩展出“|=”，例如：

```
>>> d = {0: 'abc'}
>>> d |= {1: 'def'}
>>> d
{0: 'abc', 1: 'def'}
>>> d |= {2: 'ghi', 1:'xyz'}
>>> d
{0: 'abc', 1: 'xyz', 2: 'ghi'}
>>>
```

作为容器类型的一元，映射同样支持通过内置函数len()取得其长度，即其包含的键值对的个数。下面是一些例子：

```
>>> len(dict(a=1, b=2, c=3, d=4))
4
>>> len({(0, 1): 1, (3, 4): 5})
2
>>> len(dict())
0
>>>
```

映射同样支持抽取操作，但并不会根据索引抽取出元素（键值对），而是根据键抽取出相应的值。这是通过如下语法实现的：

```
mapping[key]
```

其中mapping是一个求值结果为映射的表达式，而key则是一个求值结果为可哈希对象的表达式。这会导致mapping.__getitem__(key)被调用。但与序列的__getitem__不同，映射的__getitem__会首先计算通过key参数传入的对象的哈希值，然后根据该哈希值取得相应的值，这导致了访问映射中元素的速度较慢（除非利用了前面提到的字符串驻留技术提速）。PEP 8建议键和方括号之间不要插入空格。

由于起作用的是键的哈希值，所以使用具有相同哈希值的键将抽取同一个值，例如：

```
>>> d = {0: 'zero'}
>>> d[0]
'zero'
>>> d[0.0]
'zero'
>>> d[0j]
'zero'
>>>
```

如果对映射执行抽取操作时指定的键不是可哈希的，则会抛出TypeError异常，例如：

```
>>> d = {0: 'a', 1: 'b'}
>>> d[bytearray(b'a')]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'bytearray'
>>>
```

而如果指定的键是可哈希的，但不匹配映射中的任何键值对，则会抛出KeyError异常，例如：

```
>>> d = {0: 'a', 1: 'b'}
>>> d[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 2
>>>
```

特别的，如果一个映射类型实现了__missing__魔术属性，则当指定的键不匹配任何键值对时会自动调用__missing__，该键会通过__missing__的key参数传入，而__missing__的返回值会被作为抽取出的值。字典不具有__missing__，因此下面的例子自定义了一个映射类型来说明__missing__的作用：

```
import collections

#该类继承了collections模块定义的UserDict类，使得自定义映射类型变得简单。
class MyDict(collections.UserDict):
    def __init__(self, initialdata):
        self.data = collections.UserDict(initialdata)

    def __missing__(self, key):
        return "Unmatched key: " + str(key)
```

请将上述代码保存为MyDict.py，然后通过如下命令行和语句验证：

```
$ python3 -i MyDict.py

>>> md = MyDict([(0, 'a'), (1, 'b')])
>>> md[0]
'a'
>>> md[1]
'b'
>>> md[2]
'Unmatched key: 2'
>>> md[3]
'Unmatched key: 3'
>>>
```

第3章已经说明字典是可变的，因此字典支持通过如下语法实现的设置和删除：

```
dict[key] = value
del dict[key]
```

同样，设置操作的语法将导致dict.__setitem__(key, value)被调用，而删除操作的语法将导致dict.__delitem__(key)被调用。但与序列不同，对字典执行设置操作时，如果通过key指定的键在通过dict指定的字典中没有键值对匹配，那么会将键值对key:value添加到dict中，

而非抛出KeyError异常。而对字典执行删除操作时，如果通过key指定的键在通过dict指定的字典中没有键值对匹配，则会抛出KeyError异常。下面是一个例子：

```
>>> d = {}
>>> d['H'] = (1, 'hydrogen')
>>> d
{'H': (1, 'hydrogen')}
>>> d['He'] = (2, 'nitrogen')
>>> d
{'H': (1, 'hydrogen'), 'He': (2, 'nitrogen')}
>>> d['He'] = (2, 'helium')
>>> d
{'H': (1, 'hydrogen'), 'He': (2, 'helium')}
>>> del d['He']
>>> d
{'H': (1, 'hydrogen')}
>>> del d['N']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'N'
>>>
```

由于键可以是任何可哈希对象，一个映射中的键的类型可能各不相同，更谈不上存在隐含的顺序关系，所以映射类型不支持切片。不过值得一提，CPython在实现字典时会按照键值对被添加的顺序储存它们，所以还是有一个内在顺序，但该顺序没有任何逻辑上的意义。

映射类型同样实现了魔术属性__contais__，以支持成员检测。但需要强调的是，对映射进行in和not in运算判断的是它包含的键值对是否能匹配指定的键，与值没有关系。下面是一个例子：

```
>>> d = {0: 1, 1.0: 2}
>>> 0 in d
True
>>> 0.0 in d
True
>>> 0j in d
True
>>> 1 in d
True
>>> 2 in d
False
>>>
```

特别的，我们可以将映射作为list()内置函数的参数，这样得到的列表将包含该映射的所有键（但顺序无法预知）。下面是一个例子：

```
>>> d = {137.8: 'a', None: 'b', (0, 10): -1}
>>> list(d)
[137.8, None, (0, 10)]
```

显然，映射类型只重写了__eq__和__ne__，没有重写__lt__、__le__、__gt__和__ge__。这意味着映射只支持进行==或!=比较，规则为：两个映射相等当且仅当它们具有相同的键值对。这其实将映射的比较转换成了键值对的比较。而在比较键值对时，首先基于键的哈希值确定键值对之间的对应关系，如果存在不对应的键值对则直接判断为不相等；然后再比较对应的键值对之间的值，只要有一组不相等就判断为不相等。下面是一个例子：

```
>>> d1 = {0: 'a', 1: 'b'}
>>> d2 = {0.0: b'a', 1.0: b'b'}
>>> d3 = {0j: bytearray(b'a'), 1+0j: bytearray(b'b')}
>>> d1 == d2
False
>>> d1 == d3
False
>>> d2 == d3
True
>>>
```

最后，表11-5列出了字典额外支持的操作。下面依次介绍。

表11-5. 字典的属性

属性	说明
dict.fromkeys()	一个类方法，创建具有指定的键且所有键都对应同一指定值的字典。
dict.update()	替换字典中的键值对。
dict.get()	抽取指定键对应的值，如果没有匹配的键值对则返回指定的默认值。
dict.setdefault()	抽取指定键对应的值，如果没有匹配的键值对则先添加该键到指定默认值的键值对，然后返回该默认值。
dict.pop()	从字典删除匹配指定键的键值对，并返回值。
dict.popitem()	按照先入后出的顺序从字典删除并返回一个键值对。
dict.clear()	清空字典。
dict.copy()	返回字典的拷贝。
dict.items()	返回一个由字典中的键值对构成的视图。
dict.keys()	返回一个由字典中的键构成的视图。
dict.values()	返回一个由字典中的值构成的视图。

dict.fromkeys()的语法为：

```
classmethod dict.fromkeys(iterable, value=None)
```

其中iterable参数可以被传入一个已经存在的映射，这样新建字典的键将与该映射相同；也可以被传入一个元素都是可哈希对象的非映射类型容器，这样新建字典的键将为该容器中的元素。value参数用于指定新建字典中的所有键所对应的值。下面是一些例子：

```
>>> dict.fromkeys({'u':25, 'v':-37}, 1+9j)
{'u': (1+9j), 'v': (1+9j)}
>>> dict.fromkeys([0, 1, 2])
{0: None, 1: None, 2: None}
>>>
```

dict.update()的语法为：

```
dict.update([iterable, ]**kwargs)
```

注意该属性的形式参数与内置函数dict()的完全相同。事实上，该属性等价于先基于传入的参数在内部创建一个字典dict0，然后执行“dict = dict | dict0”。易知，如果没有传入任何实际参数，则原字典保持不变。下面是一个例子：

```
>>> d = {'a': -1}
>>> d.update({'a': 1}, b=2, c=3)
>>> d
{'a': 1, 'b': 2, 'c': 3}
>>> d.update()
>>> d
{'a': 1, 'b': 2, 'c': 3}
>>>
```

dict.get()本质上也用于实现抽取操作，但即便没有__missing__魔术属性也不会抛出KeyError异常。它的语法为：

```
dict.get(key, default=None)
```

该属性的行为是：如果字典中存在与通过key参数传入的键匹配的键值对，则返回值；否则返回default。下面是一个例子：

```
>>> d = {}
>>> d.get('a', 1)
1
>>> d
{}
>>> d.get('a', 2)
2
>>> d
{}
>>> print(d.get(b'a'))
```

```
None
>>>
```

dict.setdefault()与dict.get()类似，其语法为：

```
dict.setdefault(key, default=None)
```

该属性与dict.get()只有一点区别：当通过key参数传入的键不匹配任何键值对时，先添加键值对key:default，然后再返回default。下面是一个例子（请与关于dict.get()得例子做对比）：

```
>>> d = {}
>>> d.setdefault('a', 1)
1
>>> d
{'a': 1}
>>> d.setdefault('a', 2)
1
>>> d
{'a': 1}
>>> print(d.setdefault(b'a'))
None
>>>
```

dict.pop()与seq.pop()的功能类似，但基于键而非索引弹出字典中的元素，且返回的是键值对中的值。dict.pop()的语法为：

```
dict.pop(key[, default])
```

请注意该属性的default参数并没有默认实参值，因此如果省略了default参数，而key参数指定的键在字典中没有匹配的键值对，则会抛出KeyError异常。下面是一个例子：

```
>>> d = {'x': 10.2, 'y': 36.8}
>>> d.pop('x', 0.0)
10.2
>>> d
{'y': 36.8}
>>> d.pop('x', 0.0)
0.0
>>> d
{'y': 36.8}
>>> d.pop('y')
36.8
>>> d
{}
>>> d.pop('y')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
KeyError: 'y'
>>>
```

字典并不适合实现栈和队列，dict.pop()也不能弹出字典中的元素，因为字典中的元素在逻辑上是键值对。然而dict.popitem()将字典当成了一个栈，按照键值对被添加到字典中的顺序反方向弹出键值对，直到字典为空时才会抛出KeyError异常。dict.popitem()总是返回一个二元组，其中第一个元素为键，第二个元素为值。调用一个空字典的popitem()会抛出KeyError异常。下面的例子说明了该属性的用法：

```
>>> d = dict(a='A', b='B', c='C')
>>> d
{'a': 'A', 'b': 'B', 'c': 'C'}
>>> d.popitem()
('c', 'C')
>>> d
{'a': 'A', 'b': 'B'}
>>> d.popitem()
('b', 'B')
>>> d
{'a': 'A'}
>>> d.popitem()
('a', 'A')
>>> d
{}
>>> d.popitem()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'popitem(): dictionary is empty'
>>> d['d'] = 'D'
>>> d
{'d': 'D'}
>>> d.popitem()
('d', 'D')
>>>
```

dict.clear()与seq.clear()的功能类似，会一次性删除字典中的所有键值对，使该字典变成一个空字典。下面是一个例子：

```
>>> d = dict.fromkeys(range(5))
>>> d
{0: None, 1: None, 2: None, 3: None, 4: None}
>>> d.clear()
>>> d
{}
>>>
```

dict.copy()与seq.copy()的功能类似，会返回原字典的一个浅拷贝。下面是一个例子：

```
>>> d1 = {'p1': (9, -16), 'p2': (-3, 0)}
>>> d2 = d1.copy()
>>> d2
{'p1': (9, -16), 'p2': (-3, 0)}
```

```
>>> d2 is d1
False
>>> d2['p1'] is d1['p1']
True
>>> d2['p2'] is d1['p2']
True
>>>
```

dict.items()、dict.keys()和dict.values()都返回绑定到原字典的视图，分别用于查看该字典的键值对、键和值。这些视图的类型分别是dict_items、dict_keys和dict_values，只支持如下操作：

- 直接显示该字典当前有哪些键值对、键和值。
- 通过len()取得视图中键值对、键和值的数量。
- 通过in或not in判断一个键值对、键或值是否在视图中。
- 通过迭代取得视图中的键值对、键或值。

需要强调的是，视图不支持任何本质是写入的操作，也不支持抽取。此外，视图本质上只是一种对绑定字典的受限访问方式，如果字典的内容发生变化，则该变化会实时反映到这些视图中。下面的例子说明了这些视图的基本用法（忽略了迭代）：

```
>>> d = {0: 'abc', b'x': 3+9j, (1, 2): [1, 2], None: NotImplemented}
>>> v_items = d.items()
>>> v_items
dict_items([(0, 'abc'), (b'x', (3+9j)), ((1, 2), [1, 2]), (None,
NotImplemented)])
>>> len(v_items)
4
>>> (b'x', (3+9j)) in v_items
True
>>> v_keys = d.keys()
>>> v_keys
dict_keys([0, b'x', (1, 2), None])
>>> len(v_keys)
4
>>> 0 not in v_keys
False
>>> v_values = d.values()
>>> v_values
dict_values(['abc', (3+9j), [1, 2], NotImplemented])
>>> NotImplemented in v_values
True
>>> del d[None]
>>> v_items
dict_items([(0, 'abc'), (b'x', (3+9j)), ((1, 2), [1, 2])])
>>> v_keys
dict_keys([0, b'x', (1, 2)])
>>> v_values
dict_values(['abc', (3+9j), [1, 2]])
>>>
```

dict_items、dict_keys和dict_values还具有一个共同的属性——mapping。该属性会引用一个MappingProxyType对象，该对象同样是绑定到原字典的视图，但额外支持抽取操作，因此被视为原字典的一个只读代理。下面的例子是上面例子的继续，说明了如何通过该只读代理访问原字典：

```
>>> v_items.mapping
mappingproxy({0: 'abc', b'x': (3+9j), (1, 2): [1, 2]})
>>> v_keys.mapping
mappingproxy({0: 'abc', b'x': (3+9j), (1, 2): [1, 2]})
>>> v_values.mapping
mappingproxy({0: 'abc', b'x': (3+9j), (1, 2): [1, 2]})
>>> proxy = v_keys.mapping
>>> proxy[0]
'abc'
>>> proxy.get(b'x', False)
(3+9j)
>>> proxy.get(b'y', False)
False
>>>
```

第12章讨论更多关于映射的只读代理的知识。

11-5. 集合类型的典型例子

（教程：5.4）
（语言参考手册：6.2.6）
（标准库：内置函数、内置类型）

序列反映了索引和元素之间的对应关系，映射反映了键和值之间的对应关系，而集合类型对象不反映任何对应关系——它们只是将若干可哈希对象聚拢在一起。第3章已经提到，内置的集合类型包括集合和凝固集合，这里需要补充说明集合是可变的，而凝固集合是不可变的。由于集合类型对象中的元素必须是可哈希的，所以凝固集合自身也是可哈希的。

内置类型frozenset表示凝固集合。可以通过内置函数frozenset()创建凝固集合，其语法为：

```
class frozenset([iterable])
```

其中iterable参数需要是一个只包含可哈希对象的可迭代对象，该对象会被迭代，依次得到的对象会成为新建凝固集合的元素。而对于具有相同哈希值的多个元素，将只保留其中第一个。如果省略了iterable参数，则得到一个空凝固集合。

下面是一些例子：

```
>>> frozenset([0, 2, 0, 4, 4, 8])
frozenset({0, 8, 2, 4})
>>> frozenset('abucus')
frozenset({'c', 'a', 'u', 'b', 's'})
>>> frozenset([frozenset({None}), frozenset({False, True})])
frozenset({frozenset({False, True}), frozenset({None})})
>>> frozenset()
frozenset()
>>>
```

这些例子还说明了两点：

- 凝固集合的元素可以是其他凝固集合。
- 凝固集合只能显示为对frozenset()的调用，没有更直观的显示方法。

内置类型set表示集合。可以通过内置函数set()创建集合，其语法为：

```
class set([iterable])
```

其中iterable参数需要是一个只包含可哈希对象的可迭代对象，该对象会被迭代，依次得到的对象会成为新建集合的元素。对于具有相同哈希值的多个元素，同样只保留其中第一个。如果省略了iterable参数，则得到一个空集合。

下面的例子改编自之前的例子：

```
>>> set([0, 2, 0, 4, 4, 8])
{0, 8, 2, 4}
>>> set('abucus')
{'c', 'a', 'u', 'b', 's'}
>>> set([frozenset({None}), frozenset({False, True})])
{frozenset({False, True}), frozenset({None})}
>>> set()
set()
>>>
```

关于这些例子需要说明如下两点：

- 集合的元素可以是其他凝固集合，但不能是其他集合。
- 除了空集合之外，集合都可以用大括号的方式显示，与映射的区别在于只包含用“,”分隔的可哈希对象列表，但每个对象后面不存在“:”和值。这被称为“集合显示（set display）”。但要注意，“{}”代表的是空字典，因此空集合只能显示为“set()”。

最后，集合显示也可被用于创建集合，例如：

```
>>> {'abc', None, 9.4-7j}
{'abc', (9.4-7j), None}
>>>
```

11-6. 集合类型的操作

（教程：5.4）
（语言参考手册：6.10.1、6.10.2）
（标准库：内置函数、内置类型）

由于集合类型既没有索引也没有键，所以不支持抽取和切片。此外，集合类型也不支持拼接和合并。但由于它们支持数学上的集合运算，所以可以认为并集运算覆盖了拼合和合并的功能。下面先讨论表11-6列出的集合类型对象的通用属性。

表11-6. 集合类型对象的通用属性

属性	说明
set.isdisjoint()	判断两个集合类型对象是否不包含相同的元素。
set.issubset()	判断一个集合类型对象是否是另一个集合类型对象的子集。
set.issuperset()	
set.union()	实现并集运算。
set.intersection()	实现交集运算。
set.difference()	实现差集运算。
set.symmetric_difference()	实现对称差集运算。
set.copy()	返回该集合类型对象的拷贝。

集合类型同样实现了__len__魔术属性，因此其实例支持通过内置函数len()取得其长度，即它所包含的元素个数。下面是一些例子：

```
>>> len({19.8, 'yeah', b'ouh\0', None})
4
>>> len(frozenset({(1, 2), (3, 4)}))
2
>>>
```

集合类型支持in和not in比较，以判断一个元素是否属于某个集合类型对象，这也是数学上集合的基本功能。下面是一些例子：

```
>>> 'a' in set('abc')
True
>>> 'xy' in frozenset('xyz')
False
>>>
```

除此之外，set.isdisjoint()被用于判断两个集合类型对象是否包含相同的元素，其语法为：

```
set.isdisjoint(other)
```

其中other参数需被传入另一个集合类型对象。如果set和other包含有相同的元素（也就是说它们的交集不为空集合），该属性就会返回False。下面是一些例子：

```
>>> {1, 2, 3}.isdisjoint({1.0, 5.0})
False
>>> {1, 2, 3}.isdisjoint(frozenset([4, 5, 6]))
True
>>> frozenset([(0, 1), (2, -1)]).isdisjoint(frozenset([0, 1, 2, -1]))
True
>>> frozenset('abc').isdisjoint(set('abc'))
False
>>>
```

与数字和字符串不同，数学中的集合并没有序关系。然而如果集合A包含了集合B的所有元素，那么B就是A的子集；如果集合B是集合A的子集，但集合A并不是集合B的子集，那么B就是A的真子集；如果集合A和集合B互为子集，则它们相等。

基于这种子集关系，集合类型重写了魔术属性__lt__、__le__、__eq__、__ne__、__gt__和__ge__，使得集合类型对象相互间可以按照如下规则进行比较：

➤ ==和!=比较：当且仅当集合A和集合B包含的元素一一对应时，A==B为True，A!=B为False。
➤ <=和>=比较：当且仅当集合B是集合A的子集时，B<=A和A>=B为True。注意B<=A和A>=B为False并不意味着B>A和A<B为True，因为A和B之间可以不存在子集关系。
➤ <和>比较：当且仅当集合B是集合A的真子集时，B<A和A>B返回True。注意B<A和A>B返回False并不意味着B>=A和A<=B返回True，因为A和B之间可以不存在子集关系。

下面是一些例子：

```
>>> {0, 1} == {0.0, 1.0}
True
>>> {0, 1} != {-1.0, 1.0}
True
>>> {0, 1} <= {0.0, 1.0, 2.0}
True
>>> {0, 1} <= {0.0, 1.0}
True
>>> {0, 1} >= {0j, 1.0+0j}
True
>>> {0, 1} >= {1.0+0j}
True
>>> {0, 1} < {0.0, 1.0, 2.0}
True
>>> {0, 1} < {0.0, 1.0}
False
>>> {0, 1} > {0j, 1.0+0j}
False
>>> {0, 1} > {1.0+0j}
True
>>> {0, 1} <= {-1.0, 1.0}
False
>>> {0, 1} > {-1.0, 1.0}
False
>>> {0, 1} >= {-1.0, 1.0}
False
>>> {0, 1} < {-1.0, 1.0}
False
>>>
>>> {0, 1} == frozenset([1, 0])
True
>>> frozenset([1, 0]) < frozenset(('a', 'b', 1, 0j))
True
>>>
```

除了上述6个比较运算符，还可以通过set.issubset()和set.issuperset()来判断子集关系，它们的语法为：

```
set.issubset(other)
set.issuperset(other)
```

当且仅当set是other的子集时，set.issubset()返回True；当且仅当other是set的子集时，set.issuperset()返回True。需要强调的是，other参数除了传入集合类型对象外，还可以传入其他类型的仅包含可哈希对象的可迭代对象，而后者会被自动迭代以在内部生成一个集合。下面是一些例子：

```
>>> {0, 1}.issubset([1.0, 0j, 2])
True
>>> frozenset('acd').issuperset(['a', 'd'])
True
>>>
```

依靠实现__or__，通过“|”连接若干个集合类型对象可以实现并集运算。若这些集合类型对象中存在哈希值相同的多个元素，则并集只保留其中第一个元素。并集的具体类型由表达式中第一个集合类型对象确定。

下面是一些例子：

```
>>> st1 = {0, 1} | frozenset('abc') | set([None, NotImplemented])
>>> st2 = frozenset('abc') | {0, 1} | set([None, NotImplemented])
>>> st1
{0, 1, 'c', NotImplemented, 'a', None, 'b'}
>>> st2
frozenset({0, 'c', 1, NotImplemented, 'a', None, 'b'})
>>> st1 == st2
True
>>> {0j, 2, 4.0} | {0, 3, 9+0j}
{0j, 2, 3, 4.0, (9+0j)}
>>>
```

而set.union()同样实现了并集运算，其语法为：

```
set.union(*other)
```

其中other参数可被传入任意多个仅包含可哈希对象的可迭代对象，当这些对象不属于集合类型时，会自动迭代它们以在内部生成相应的集合。而并集的具体类型与set的类型相同。

下面是一些例子：

```
>>> {'a', 'b'}.union('cde', 'fghi')
{'h', 'c', 'a', 'g', 'b', 'd', 'f', 'e', 'i'}
>>> frozenset([1, 2, 3]).union((4,), [5], range(6, 8))
frozenset({1, 2, 3, 4, 5, 6, 7})
>>>
```

依靠实现__and__，通过“&”连接若干个集合类型对象可以实现交集运算。同样，若这些集合类型对象中存在哈希值相同的多个对象，则交集只保留其中第一个对象。交集的具体类型也是由表达式中第一个集合类型对象确定。

下面是一些例子：

```
>>> st1 = set('abcde') & frozenset('apple')
>>> st2 = frozenset('apple') & set('abcde')
>>> st1
{'a', 'e'}
```

```
>>> st2
frozenset({'a', 'e'})
>>> st1 == st2
True
>>> {0, 1, 2, 3, 4, 5, 6} & {0, 2, 4, 6} & set(range(0, 7, 3))
{0, 6}
>>>
```

而set.intersection()同样实现了交集运算，其语法为：

```
set.intersection(*other)
```

其中other参数的含义与在set.union()中相同。而交集的具体类型同样与set的类型相同。

下面是一些例子：

```
>>> {'a', 'b', 'c'}.intersection('apple')
{'a'}
>>> frozenset(b'0011').intersection(b'1001', b'1100')
frozenset({48, 49})
>>>
```

依靠实现__sub__，通过“-”连接若干个集合类型对象可以实现差集运算。但与并集和交集不同，差集运算并不满足交换律，即A - B并不等于B - A。而A - B1 - B2 - ...等价于A - (B1 | B2 | ...)。同样，差集的具体类型也是由表达式中第一个集合类型对象确定的。

下面是一些例子：

```
>>> st1 = set(range(10)) - frozenset(range(0, 100, 2)) - frozenset(range(0,
5, 3))
>>> st2 = frozenset(range(10)) - set(range(0, 100, 2)) - frozenset(range(0,
5, 3))
>>> st1
{1, 5, 9, 7}
>>> st2
frozenset({1, 5, 9, 7})
>>> st1 == st2
True
>>> set('abcde') - {'d', 'b', 'u', 'o'}
{'c', 'e', 'a'}
>>>
```

而set.difference()同样实现了差集运算，其语法为：

```
set.difference(*other)
```

其中other参数的含义也与在set.union()中相同。而差集的具体类型同样与set的类型相同。

下面是一些例子：

```
>>> {'a', 'b', 'c'}.difference('apple')
{'b', 'c'}
>>> frozenset(b'0011').difference(b'1001', b'1100')
frozenset()
>>>
```

依靠实现__xor__，通过“^”将两个集合类型对象连接起来可以实现对称差集运算。这里要强调的是对称差集总是只涉及两个集合类型对象，用“^”将多个集合类型对象连接起来只能看成多次对称差集运算。对称差集的具体类型由第一个集合类型对象确定。

下面是一些例子：

```
>>> st1 = set('abcd') ^ frozenset('cdef')
>>> st2 = frozenset('abcd') ^ set('cdef')
>>> st1
{'a', 'b', 'f', 'e'}
>>> st2
frozenset({'a', 'b', 'f', 'e'})
>>> st1 == st2
True
>>> {0, 1} ^ {-1, 0}
{1, -1}
>>>
```

而set.symmetric_difference()同样实现了对称差集运算，其语法为：

```
set.symmetric_difference(other)
```

其中other参数需被传入一个仅包含可哈希对象的可迭代对象，同样如果它不属于集合类型则会自动被迭代以在内部创建相应集合。对称差集的具体类型同样与set的类型相同。

下面是一些例子：

```
>>> set('abcd').symmetric_difference('cdef')
{'a', 'b', 'f', 'e'}
>>> frozenset('abcd').symmetric_difference('cdef')
```

```
    frozenset({'a', 'b', 'f', 'e'})
    >>> {0, 1}.symmetric_difference([-1, 0])
    {1, -1}
    >>>
```

最后，与seq.copy()和dict.copy()类似，set.copy()返回一个集合类型对象的浅拷贝。

下面是一个例子：

```
    >>> a = (1, 2)
    >>> b = (3, 4)
    >>> st1 = {a, b}
    >>> st2 = st1.copy()
    >>> st2
    {(1, 2), (3, 4)}
    >>> for ele in st2:
    ...     print(ele is a)
    ...     print(ele is b)
    ...
    True
    False
    False
    True
```

注意该例子通过迭代验证了st2是st1的浅拷贝，而迭代会在第12章详细讨论。

至此我们讨论完了所有集合类型通用的操作，下面讨论仅能对可变集合类型施加的操作。由于集合类型不支持抽取和切片，所以对可变集合类型对象的写入操作只能通过表11-7列出的属性实现。

表11-7. 可变集合类型对象的通用属性

属性	说明
set.add()	向可变集合添加元素。
set.remove()	从可变集合删除元素，元素必须存在。
set.discard()	从可变集合删除元素，元素可以不存在。
set.pop()	从可变集合随机删除一个元素，并将该元素返回。
set.clear()	清空可变集合。
set.update()	以并集的方式更新可变集合中的元素。
set.intersection_update()	以交集的方式更新可变集合中的元素。
set.difference_update()	以差集的方式更新可变集合中的元素。
set.symmetric_difference_update()	以对称差集的方式更新可变集合中的元素。

set.add()的语法为：

```
set.add(ele)
```

它将通过ele参数传入的可哈希对象添加到set中。如果set中已经存在与传入ele参数的对象的哈希值相同的元素，则什么也不做。下面是一个例子：

```
>>> st = set()
>>> st
set()
>>> st.add(0)
>>> st
{0}
>>> st.add(1)
>>> st
{0, 1}
>>> st.add(0.0)
>>> st
{0, 1}
>>>
```

set.remove()和set.discard()都用于删除可变集合中的指定元素，其语法为：

```
set.remove(ele)
set.dicard(ele)
```

两者的区别仅当通过ele参数传入的对象在可变集合中不存在时才会体现：set.remove()会抛出KeyError异常，而set.discard()则什么也不做。下面是一个例子：

```
>>> st = {1, 2, 3, 4, 5}
>>> st.remove(2)
>>> st
{1, 3, 4, 5}
>>> st.discard(5)
>>> st
{1, 3, 4}
>>> st.remove(6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 6
>>> st
{1, 3, 4}
>>> st.discard(6)
>>> st
{1, 3, 4}
>>>
```

set.pop()与seq.pop()和dict.pop()类似。但要强调的是，Python官方手册没有对弹出一个可变集合的元素的顺序进行任何规定，因此不同Python解释器可以有不同的实现方法。这意味着无法联合使用set.add()和set.pop()实现栈或队列。调用一个空可变集合的pop()会抛出KeyError异常。下面是一个例子：

```
>>> st = set(range(5))
>>> st
{0, 1, 2, 3, 4}
>>> st.pop()
0
>>> st.pop()
1
>>> st.pop()
2
>>> st.pop()
3
>>> st.pop()
4
>>>
```

set.clear()与seq.clear()和dict.clear()的功能相同。下面是一个例子：

```
>>> st = {1, 2, 3, 4}
>>> st.clear()
>>> st
set()
>>>
```

而剩下的四种操作是从并集、交集、差集和对称差集派生出来的，即用相应集合运算的结果更新原可变集合。下面依次讨论。

依靠实现__ior__，从“|”派生出的“|=”用并集的结果更新一个可变集合。下面的例子说明了其用法：

```
>>> st = {0}
>>> st |= {1} | {2} | {3, 4}
>>> st
{0, 1, 2, 3, 4}
>>>
```

而set.update()则是从set.union()派生出来的，其语法为：

```
set.update(*other)
```

下面是一个例子：

```
>>> st = {'a', 'b'}
>>> st.update('cd', 'efg')
>>> st
{'g', 'b', 'f', 'c', 'd', 'e', 'a'}
>>>
```

依靠实现__iand__，从“&”派生出的“&=”用交集的结果更新一个可变集合，下面的例子说明了其用法：

```
>>> st = set(range(10))
>>> st &= frozenset(range(0, 20, 2)) & set(range(0, 10, 3))
>>> st
{0, 6}
>>>
```

而set.intersection_update()则是从set.intersection()派生出来的，其语法为：

```
set.intersection_update(*other)
```

下面是一个例子：

```
>>> st = set('abcdefg')
>>> st.intersection_update('aceg','hela')
>>> st
{'a', 'e'}
>>>
```

依靠实现__isub__，从“-”派生出的“-=”用差集的结果更新一个可变集合，下面的例子说明了其用法：

```
>>> st = set((3.5, 6.0, 7.3))
>>> st -= {1.0, 3.5} | {6.0, 7.0}
>>> st
{7.3}
>>>
```

注意在“-=”右边的表达式本身是一个并集而非差集，这是因为“-=”会先完成其右侧表达式的计算，再与其左侧的可变集合完成差集和更新。

set.difference_update()则是从set.difference()派生出来的，其语法为：

```
set.difference_update(*other)
```

下面是一个例子：

```
>>> st = set(b'uvwxyz')
>>> st.difference_update(b'uv', b'y', b'z')
>>> st
{119, 120}
>>>
```

依靠实现__ixor__，从“^”派生出的“^=”用对称差集的结果更新一个可变集合，下面的例子说明了其用法：

```
>>> st = set('abc')
>>> st ^= frozenset('bcd')
>>> st
{'a', 'd'}
>>>
```

而set.symmetric_difference_update()则是从set.symmetric_difference()派生出来的，其语法为：

```
set.symmetric_difference_update(other)
```

下面是一个例子：

```
>>> st = {0, 1}
>>> st.symmetric_difference_update({-1, 0})
>>> st
{1, -1}
>>>
```