第3章 Python序列

序列是程序设计中经常用到的数据存储方式，几乎每一种程序设计语言都提供了类似的数据结构，简单地说，序列是一块用来存放多个值的连续内存空间。一般而言，在实际开发中同一个序列中的元素通常是相关的。Python 提供的序列类型可以说是所有程序设计语言类似数据结构中最灵活的，也是功能最强大的。

3.1 列表与列表推导式

Python 中常用的序列结构有列表、元组、字典、字符串、集合等。所有序列类型都可以进行某些特定的操作。这些操作包括：索引(indexing)、分片(sliceling)、加(adding)、乘(multiplying)以及检查某个元素是否属于序列的成员(成员资格)。除此之外，Python 还有计算序列长度、找出最大元素和最小元素的内置函数。

列表是 Python 的内置可变列表，是包含若干元素的有序连续内存空间。在形式上，列表的所有元素放在一对方括号“[”和“]”中，相邻元素之间使用逗号分隔开。当列表增加或删除元素时，列表对象自动进行内存的扩展或收缩，从而保证元素之间没有缝隙。Python 列表内存的自动管理可以大幅度减少程序员的负担，但列表的这个特点会涉及列表中大量元素的移动，效率较低，并且对于某些操作可能会导致意外的错误结果。因此，除非确实有需求，否则尽量从列表尾部进行元素的增加或删除操作，这会大幅度提高列表处理速度。

列表(List)是一组有序存储的数据，例如，饭店点餐的菜单就是一种列表。列表具有如下特性：

(1) 与变量一样，每个列表都有一个唯一标识它的名称。

(2) 一个列表的元素应具有相同的数据类型。

(3) 每个列表元素都有索引和值两个属性，索引是一个从 0 开始的整数，用户标识元素在列表中的位置，值就是对应位置的元素的值。

同一个列表中元素的类型可以不相同，可以同时包含整数、实数、字符串等基本类型，也可以是列表、元组、字典以及其他自定义类型的对象。例如

```
[1, 2, 3, 4, 5]
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
['spam', 1.0, 6, [10, 20]]
[['Tom', 10, 3], ['Mary', 8, 1]]
```

都是合法的列表对象。

对于 Python 序列而言，有很多方法是通用的，而不同类型的序列又有一些特有的方法。列表对象常用方法如表 3-1 所示，假设表中的示例基于 s=[1,3,2]。除此之外，Python 的很多内置函数和命令也可以对列表和其他序列对象进行操作，后面将逐步进行介绍。

表 3-1　列表对象常用方法

方　法	说　明	示　例
s. append(x)	将元素 x 添加至列表尾部	s. append('a')＃s=[1,3,2,'a'] s. append([1,2])＃s=[1,3,2,[1,2]]
s. extend(t)	将列表 t 附加至列表 s 尾部	s. extend([4])＃s=[1,3,2,4] s. extend(['ab'])＃s=[1,3,2,'a','b']
s. insert(i,x)	在列表指定位置 i 处添加元素 x	s. insert(1,4)＃s=[1,4,3,2] s. insert(8,5)＃s=[1,3,2,5]
s. remove(x)	在列表中删除首次出现的指定元素，若对象不存在，将导致 ValueError	s. remove(1)＃s=[3,2] s. remove(0)＃ValueError: list. remove(x): x not in list
s. pop([i])	删除并返回列表对象指定位置的元素，默认为最后一个元素	s. pop()＃输出 2。S=[1,3] s. pop(0)＃输出 1。S=[3,2]
s. index(x)	返回第一个值为 x 的元素的下标，若不存在值为 x 的元素，则抛出异常	s. index(1)＃输出 0 s. index(5)＃ValueError: 5 is not in list
s. count(x)	返回指定元素 x 在列表中的出现次数	s. count(1)＃输出 1 s. count(0)＃输出 0
s. reverse()	对列表元素进行原地翻转	s. reverse()＃s=[2,3,1]
s. sort()	对列表元素进行原地排序	s. sort()＃s=[1,2,3]

3.1.1　列表的创建与删除

1. 创建列表

列表采用方括号中用逗号分隔的项目来定义。其基本形式如下：

```
[x1, [x2, ..., xn] ]
```

如同其他类型的 Python 对象变量一样，使用赋值运算符“=”直接将一个列表赋值给变量即可创建列表对象，例如：

```
>>> a_list = ['a', 'b', 'c', 'd']
>>> a_list = []   #创建空列表
```

或者，也可以使用 list()函数将元组、range 对象、字符串或其他类型的可迭代对象类型的数据转换为列表。例如：

```
>>> a_list = list((3,5,7,9,11))
>>> a_list
[1, 3, 5, 7, 9]
>>> list('hello world')
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> x = list()    #创建空列表
```

例 3-1 创建列表对象。

```
>>> []
[]
>>> [1, 2, 3]
[1, 2, 3]
>>> list()
[]
>>> list((1, 2, 3))
[1, 2, 3]
>>> list(range(3))
[0, 1, 2]
>>> list('abc')
['a', 'b', 'c']
>>> list([1, 2, 3])
[1, 2, 3]
>>> a = ['x', 2]
>>> a
['x', 2]
```

上面的代码中用到了内置函数 range()，这是一个非常有用的函数，后面会多次用到，该函数语法为：

```
range([start, ] stop[, step])
```

内置函数 range()接收 3 个参数，第一个参数表示起始值(默认为 0)，第二个参数表示终止值(结果中不包括这个值)，第三个参数表示步长(默认为 1)，该函数在 Python 2.x 中返回一个包含若干整数的列表。另外，Python 2.x 还提供了一个内置函数 xrange()，语法与 range()函数一样，但是返回 xrange 可迭代对象，其特点为惰性求值，而不是像 range()函数一样返回列表。例如：

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> xrange(5)
xrange(5)
>>> list(xrange(5))
[0, 1, 2, 3, 4]
```

使用 Python 2.x 处理大数据或较大循环范围时，建议使用 xrange()函数来控制循环次数或处理范围，以获得更高的效率。

2. 删除列表

当列表不再使用时，使用 del 命令删除整个列表，如果列表对象所指向的值不再由其他对象指向，Python 将同时删除该值。

```
>>> del a_list
>>> a_list
NameError: name 'a_list' is not defined
```

正如上面的代码所展示的那样，删除列表对象 a_list 之后，该对象就不存在了，再次访问时将抛出异常 NameError 提示所访问的对象名不存在。

例 3-2 列表的创建与删除操作示例。

```
>>> s = [1, 2, 3, 4, 5, 6]
>>> s[1] = 'a'
>>> s
[1, 'a', 3, 4, 5, 6]
>>> s[2:3] = []
>>> s
[1, 'a', 5, 6]
>>> s[:1] = []
```

```
>>> s[2] = []
>>> s
[1, 'a', [], 4, 5, 6]
>>> del s[3]
>>> s
[1, 'a', [], 5, 6]
>>> s[:2]
[1, 'a']
>>> s
['a', 5, 6]
>>> s[:2] = 'b'
>>> s
['b', 6]
>>> del s[:1]
>>> s
[6]
```

3.1.2 列表元素的增加

列表元素的动态增加和删除是实际应用中经常遇到的操作，Python 列表提供了多种不同的方法来实现这一功能。

1. 运算符(+)

可以使用运算符(+)来实现将元素添加到列表中的功能。虽然这种用法在形式上比较简单也容易理解，但严格意义上来讲，这并不是真的为列表添加元素，而是创建一个新的列表，并将原列表中的元素和新元素依次复制到新列表的内存空间。由于涉及原列表元素的复制，该操作速度较慢，在涉及大量元素添加时不建议使用该方法。

```
>>> s = [3, 4, 5]
>>> s = s + [7]
>>> s
[3, 4, 5, 7]
```

2. append()方法

使用列表对象的 append()方法，原地修改列表，是真正意义上的在列表尾部添加元素，速度较快，也是推荐使用的方法。

```
>>> s.append(9)
>>> s
[3, 4, 5, 7, 9]
```

3. extend()方法

使用列表对象的 extend()方法可以将另一个迭代对象的所有元素添加至该列表对象尾部。

```
>>> s.extend([11, 13])
>>> s
[3, 4, 5, 7, 9, 11, 13]
```

4. insert()方法

使用列表对象的 insert()方法将元素添加至列表的指定位置。

```
>>> s = [3, 4, 5]
```

```
>>> s.insert(2, 6)
>>> s
[3, 4, 6, 5, 3, 4, 6, 5]
```

列表的 insert()方法可以在列表的任意位置插入元素，但由于列表的自动内存管理功能，insert()方法会涉及插入位置之后所有元素的移动，这会影响处理速度，类似的还有后面介绍的 remove()方法以及使用 pop()函数弹出列表非尾部元素和使用 del 命令删除列表非尾部元素的情况。因此，除非必要，应尽量避免在列表中间位置插入和删除元素的操作，而是优先考虑使用前面介绍的 append()方法。

5. 乘法运算符(*)

使用乘法来扩展列表对象，将列表与整数相乘，生成一个新列表，新列表是原列表中元素的重复。

```
>>> s = [3, 5, 7]
>>> t = s * 3
>>> t
[3, 5, 7, 3, 5, 7, 3, 5, 7]
```

该操作实际上是创建了一个新的列表，而不是真的扩展了原列表，该操作同样适用于字符串和元组，并具有相同的特点。

需要注意的是，当使用 * 运算符将包含列表的列表进行重复并创建新列表时，并不创建元素的复制，而是创建已有对象的引用。因此，当修改其中一个值时，相应的引用也会被修改，例如下面的代码：

```
>>> x = [[None] * 2] * 3
>>> x
[[None, None], [None, None], [None, None]]
>>> x[0][0] = 1
>>> x
[1, None], [1, None], [1, None]]
>>> x = [[1, 2, 3]] * 3
>>> x[0][0] = 10
>>> x
[[10, 2, 3], [10, 2, 3], [10, 2, 3]]
```

例 3-3 列表元素的增加操作示例。

```
>>> s = [1, 2, 3, 4, 5]
>>> t = ['a', 'b', 'c']
>>> s + t
[1, 2, 3, 4, 5, 'a', 'b', 'c']
>>> s.append(6)
>>> s
[1, 2, 3, 4, 5, 6]
>>> s.extend([8, 9])
>>> s
[1, 2, 3, 4, 5, 6, 8, 9]
>>> s.insert(6, 7)
>>> s
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> t * 2
['a', 'b', 'c', 'a', 'b', 'c']
```

3.1.3 列表元素的删除

1. del 命令

使用 del 命令删除列表中的指定位置上的元素。前面已经提到过，del 命令也可以直接删除整个列表，此处不再赘述。

```
>>> s = [3, 5, 7, 9, 11]
>>> del s[1]
>>> s
[3, 7, 9, 11]
```

2. pop()方法

使用列表的 pop()方法删除并返回指定(默认为最后一个)位置上的元素，如果给定的索引超过了列表的范围，则抛出异常。

```
>>> t = [3, 5, 7, 9, 11]
>>> t.pop()
11
>>> t.pop(1)
5
>>> t
[3, 7, 9]
```

3. remove()方法

使用列表对象的 remove()方法删除首次出现的指定元素，如果列表中不存在要删除的元素，则抛出异常。

```
>>> x = [3, 5, 7, 9, 7, 11]
>>> x.remove(7)
>>> x
[3, 5, 9, 7, 11]
```

例 3-4 列表元素的删除操作示例。

```
>>> lst = [1,2,4,5,6]
>>> del lst[2]
>>> lst
[1, 2, 5]
>>> lst.remove(5)
[1, 2,]
>>> lst.pop()
2
>>> lst
[1,]
```

3.1.4 列表元素访问与计数

可以使用下标直接访问列表中的元素。如果指定下标不存在，则抛出异常提示下标越界，例如：

```
>>> s = [3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
```

```
>>> s[3]
6
>>> s[3] = 5.5
>>> s
[3, 4, 5, 5.5, 7, 9, 11, 13, 15, 17]
>>> s[15]
IndexError: list index out of range
```

使用列表对象的 index()方法可以获取指定元素首次出现的下标，语法为 str.index(str,start,stop)，其中 start 和 stop 用来指定搜索范围，start 默认为 0，stop 默认为列表长度。若列表对象中不存在指定元素，则抛出异常提示列表中不存在该值，例如：

```
>>> s
[3, 4, 5, 5.5, 7, 9, 11, 13, 15, 17]
>>> s.index(7)
4
>>> s.index(100)
ValueError: 100 is not in list
```

如果需要知道指定元素在列表中出现的次数，可以使用列表对象的 count()方法进行统计，例如：

```
>>> s
[3, 4, 5, 5.5, 7, 9, 11, 13, 15, 17]
>>> s.count(7)
1
>>> s.count(0)
0
```

该方法也可用于元组、字符串以及 range 对象，例如：

```
>>> range(10).count(3)
1
>>> (3, 3, 4, 4).count(3)
2
>>> 'abcdefgabc'.count('abc')
2
```

3.1.5 成员资格判断

如果需要判断列表中是否存在指定的值，可以使用前面介绍的 count()方法，如果存在，则返回大于 0 的数；如果返回 0，则表示不存在。或者，使用更加简洁的 in 关键字来判断一个值是否存在于列表中，返回结果为 True 或 False。

例 3-5 成员资格判断操作示例。

```
>>> s = [1, 2, 3]
>>> s
[1, 2, 3]
>>> 3 in s
True
True
>>> [5] in t
False
>>> s1 = [3, 5, 7, 9, 11]
>>> s2 = ['a', 'b', 'c', 'd']
```

```
>>> 18 in s
False
>>> t = [[1], [2], [3]]
>>> 3 in t
False
>>> 3 not in t
True
>>> [3] in t
True
>>> (3, 'a') in zip(s1, s2)
True
>>> for a, b in zip(s1, s2):
        print (a, b)
3 a
5 b
7 c
9 d
```

关键字 in 和 not in 也可以用于其他可迭代对象，包括元组、字典、range 对象、字符串、集合等，常用在循环语句中对序列或其他可迭代对象中的元素进行遍历。使用这种方法来遍历序列或迭代对象，可以减少代码的输入量、简化程序员的工作，并且大幅度提高程序的可读性，建议读者熟练掌握和运用。

3.1.6 切片操作

切片是 Python 序列的重要操作之一，适用于列表、元组、字符串、range 对象等类型。切片使用两个冒号分隔的 3 个数字来完成：第一个数字表示切片开始位置（默认为 0），第二个数字表示切片截止（但不包含）位置（默认为列表长度），第三个数字表示切片的步长（默认为 1），当步长省略时可以顺便省略最后一个冒号。可以使用切片来截取列表中的任何部分，得到一个新列表，也可以通过切片来修改和删除列表中的部分元素，甚至可以通过切片操作为列表对象增加元素。

与使用下标访问列表元素的方法不同，切片操作不会因为下标越界而抛出异常，而是简单地在列表尾部截断或者返回一个空列表，代码具有更强的健壮性。

例 3-6 列表的切片操作示例。

```
>>> s = [3, 5, 7, 9, 11]
>>> s[::]
[3, 5, 7, 9, 11]
>>> s[::2]
[3, 7, 11]
>>> s[1::2]
[5, 9]
>>> s[3::]
[9, 11]
>>> s[3: 6]
[9, 11]
>>> s[3: 6:1]
[9, 11]
>>> s[0:100:1]
[3, 5, 7, 9, 11]
>>> s[100:]
[]
```

可以使用切片操作来快速实现很多目的，例如原地修改列表内容，列表元素的增、删、改、查以及元素替换等操作都可以通过切片来实现，并且不影响列表对象内存地址。

```
>>> s = [3, 5, 7]
>>> s[len(s):]
[]
>>> s[len(s):] = [9]
>>> s
[3, 5, 7, 9]
>>> s[:3] = [1, 2, 3]
>>> s
[1, 2, 3, 9]
>>> s[:3] = []
>>> s
[9]
>>> s = list(range(10))
>>> s
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> s[::2] = [0] * (len(s)/2)
>>> s
[0, 1, 0, 3, 0, 5, 0, 7, 0, 9]
```

也可以结合使用 del 命令与切片操作来删除列表中的部分元素。

```
>>> s = [3, 5, 7, 9, 11]
>>> del s[:3]
>>> s
[9, 11]
```

切片返回的是列表元素的浅复制，与列表对象的直接复制并不一样。

例 3-7 列表元素的浅复制与直接复制操作示例。

```
>>> s1 = [3, 5, 7]
>>> s2 = s1    #s1 和 s2 指向同一块内存
>>> s2
[3, 5, 7]
>>> s2[1] = 8
>>> s1
[3, 8, 7]
>>> s1 == s2
True
>>> s1 is s2
True
>>> s1 = [3, 5, 7]
>>> s2 = s1[::]       #浅复制
>>> s1 == s2
True
>>> s1 is s2
False
>>> s2[1] = 8
>>> s2
[3, 8, 7]
>>> s1
[3, 5, 7]
>>> s1 == s2
False
>>> s1 is s2
False
```

3.1.7 列表排序

在实际应用中，经常需要对列表元素进行排序。

1. sort()方法

sort()方法用于在原位置对列表进行排序。在“原位置排序”意味着改变原来的列表，从而让其中的元素能按一定的顺序排列，而不是简单地返回一个已排序的列表副本，该方法支持多种不同的排序方式。

```
>>> s = [2, 4, 6, 1, 3, 5]
>>> s.sort()
>>> s
[1, 2, 3, 4, 5, 6]
```

当用户需要一个排好序的列表副本，同时又保留原有列表不变的时候，正确方法是，首先把 s 的副本复制给 t，然后对 t 进行排序，例如：

```
>>> s = [2, 4, 6, 1, 3, 5]
>>> t = s[ : ]
>>> t.sort()
>>> s
[1, 2, 3, 4, 5, 6]
>>> t
[2, 4, 6, 1, 3, 5]
```

再次调用 a[:]得到的是包含了 s 所有元素的切片，这是一种很高效的复制整个列表的

方法。如果只是简单地把 s 赋值给 t 是没用的，因为这样做就让 s 和 t 都指向同一个列表了。

```
>>> t = a
>>> t.sort()
>>> s
[1, 2, 3, 4, 5, 6]
>>> t
[1, 2, 3, 4, 5, 6]
```

2. sorted()

也可以使用内置函数 sorted()对列表进行排序，与列表对象的 sort()方法不同，内置函数 sorted()返回新列表，并不对原列表进行任何修改。

```
>>> s = [2, 4, 6, 1, 3, 5]
>>> t = sorted(s)
>>> s
s = [2, 4, 6, 1, 3, 5]
>>> t
[1, 2, 3, 4, 5, 6]
```

这个函数实际上可以用于任何序列，却总是返回一个列表：

```
>>> sorted('Python')
['P', 'h', ',n', 'o', 't', 'y']
```

在某些应用中可能需要将列表元素进行逆序排列，也就是所有元素位置翻转，第一个元素与最后一个元素交换位置，第二个元素与倒数第二个元素交换位置，以此类推。Python 提供了内置函数 reverse()支持对列表元素进行逆序排列，与列表对象的 reverse()方法不同，内置函数 reversed()不对原列表做任何修改，而是返回一个逆序排列后的迭代对象，例如：

```
>>> s = [3, 4, 5, 2, 1]
>>> t = reversed(s)
>>> t
<listreverseiterator at 0xa46af28>
>>> list(t)
[1, 2, 5, 4, 3]
```

例 3-8 列表元素的排序操作示例。

```
>>> s1 = [2, 4, 6, 1, 3, 5]
>>> s1.sort()
>>> s1
[1, 2, 3, 4, 5, 6]
>>> s1.reverse()
>>> s1
>>> s2 = [2, 4, 6, 1, 3, 5]
>>> s = reversed(s2)
>>> list(s)
[5, 3, 1, 6, 4, 2]
[5, 3, 1, 6, 4, 2]
>>> s1.sort(reverse = True)
>>> s2 = [2, 4, 6, 1, 3, 5]

>>> sorted(s2)
[1, 2, 3, 4, 5, 6]
>>> list(s2)
[6, 5, 4, 3, 2, 1]
>>> list(s2)
[6, 5, 4, 3, 2, 1]
```

3.1.8 列表推导式

使用列表推导式，可以简单、高效地处理一个可迭代对象，并生成结果列表。列表推导式的形式如下：

```
[expr for i1 in 序列 1 ... for iN in 序列 N]                    #迭代序列中的所有内容，并计算生成列表
[expr for i1 in 序列 1 ... for iN in 序列 N if cond_expr] #按条件迭代，并计算生成列表
```

表达式 expr 使用每次迭代内容 i_1... i_N，计算生成一个列表。如果指定了条件表达式 cond_expr，则只有满足条件的元素参与迭代。例如：

```
>>> s = [x * x for x in range(10)]
```

相当于

```
>>> s = []
>>> for x in range(10):
    s.append(x * x)
```

接下来再通过几个示例来进一步展示列表推导式的强大功能。

(1) 使用列表推导式实现嵌套列表的平铺。

```
>>> vec = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

(2) 过滤不符合条件的元素。

在列表推导式中可以使用 if 子句来筛选，只在结果列表中保留符合条件的元素。例如，下面的代码用于从当前列表中选择符合条件的元素组成新的列表：

```
>>> s = [-1, -4, 6, 7.5, -2.3, 9, -11]
>>> [x for x in s if x>0]
[6, 7.5, 9]
```

(3) 在列表推导式中使用多个循环，实现多序列元素的任意组合，并且可以结合条件语句过滤特定元素。

```
>>> [(x, y) for x in range(3) for y in range(3)]
[(0, 0), (0,1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
>>> [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

(4) 使用列表推导式实现矩阵逆转。

```
>>> matrix = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

例 3-9 列表推导式操作示例。

```
>>> [i ** 2 for i in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 91]
```

```
>>> [i for i in range(10) if i % 2 == 0]
[0, 2, 4, 6, 8]
>>> [(x, y, x * y) for x in range(1, 4) for y in range(1, 4) if x >= y]
[(1, 1, 1), (2, 1, 2), (2, 2, 4), (3, 1, 3), (3, 2, 6), (3, 3, 9)]
```

3.1.9 案例精选

案例 3-1 计算列表 X=[1,5,3,4,2,7,6]中所有元素的平均值和中位数。

中位数是指 s 中所有数按照升序或降序排列后，处于最中间位置的数据值。如果元素数量是奇数，则序列 X 的最中间位置是一个数据，可以表示为 $X_{\frac{n}{2}}$，如果元素数量是偶数，序列 X 中位数是最中间两个位置数据的平均值，即$(X_{\frac{n}{2}-1}+X_{\frac{n}{2}})/2$。在本案例中，X=[1,5,3,4,2,7,6]的中位数是 4。

本案例代码如下：

```
def mean(numbers):
    sum = 0
    for num in numbers:
        sum += num
    avg = sum / len(numbers)
    return avg

def median(numbers):
    sorted(numbers)
    length = len(numbers)
    if length % 2 == 0:
        med = (numbers[length//2 - 1] + numbers[length//2])/2
    else:
        med = (numbers[length//2])
    return med

X = [1, 5, 3, 4, 2, 7, 6]
print "序列 X 的平均值:",mean(X)
print "序列 X 的中位数:",median(X)
```

案例 3-1 运行结果如下：

```
序列 X 的平均值: 4
序列 X 的中位数: 4
```

案例 3-2 以正确的宽度在居中的“盒子”内打印一个句子。

```
sentence = raw_input("Please input a sentence: ")
screen_width = 80
text_width = len(sentence)
box_width = text_width + 6
left_margin = (screen_width - box_width)/2

print '' * left_margin + "+" + '-' * (box_width - 2) + '+'
print '' * left_margin + "|" + '' *  (box_width - 2)    + '|'
print '' * left_margin + "|" + '' * 2 + sentence + '' * 2 + '|'
print '' * left_margin + "|" + '' *  (box_width - 2)   + '|'
print '' * left_margin + "+" + '-' * (box_width - 2) + '+'
```

案例 3-2 运行结果如下：

```
Please input a sentence: NICE TO MEET YOU!.
```

案例 3-3 分片示例。

对 http://www.something.com 形式的 URL 进行分割：

```
url = raw_input('Please enter the URL: ')
domain = url[11:-4]
print "Domain name: " + domain
```

案例 3-3 运行结果如下：

```
Please enter the URL: http://www.baidu.com
Domain name: baidu
```

案例 3-4 序列成员资格示例。

查看用户输入的用户名和密码是否存在于数据库(本例中是一个列表)中的程序。如果用户名/密码这一数值对存在于数据库中，那么就在屏幕上打印'Access granted'。

```
#检查用户名和密码
database = [
    ['Lucy', '1234'],
    ['Sam', '1324'],
    ['John', '5678'],
    ['Smith', '1010']
]
username = raw_input('User name: ')
pin = raw_input('PIN code: ')
if [username, pin] in database: print 'Access granted'
```

案例 3-4 运行结果如下：

```
User name: Lucy
IN code: 1234
PAccess granted
```

3.2 元组与生成器推导式

元组与列表类似，也是一种序列，但与列表不同的是，元组属于不可变序列，不能修改。元组一旦创建，用任何方法都不可以修改其元素的值，也无法为元组增加或删除元素，如果确实需要修改，只能再创建一个新的元组。

3.2.1 元组的创建与删除

元组的定义形式和列表很相似，区别在于定义元组时所有元素放在一对圆括号“()”中，而不是方括号。圆括号可以省略：如果用逗号分隔了一些值，那么就自动创建了元组。

```
(x1, [x2, ..., xn])
```

或者

```
x1, [x2, ..., xn]
```

其中，x_1，x_2，...，x_n 为任意对象。注意：如果元组中只有一个项目时，后面的逗号不能省略，这是因为 Python 解释器把(x_1)解释为 x_1，例如(1)解释为整数 1，(1,)解释为元组。

元组也可以通过创建 tuple 对象来创建。其基本形式为：

```
tuple()           #创建一个空元组
tuple(iterable)   #创建一个元组,包含的项目可为枚举对象 iterable 中的元素
```

例 3-10 创建元组对象示例。

```
>>> 1, 2, 3
(1, 2, 3)
>>> (1, 2, 3)
(1, 2, 3)
>>> ()     #空元组
()
>>> 1
1
>>> 1,
(1, )
>>> (1)
1
>>> 'a', 'b', 'c'
('a', 'b', 'c')
>>> 'a',
('a', )
>>> tuple()
()
>>> tuple(range(3))
(0, 1, 2)
>>> tuple('abc')
('a', 'b', 'c')
>>> tuple([1, 2, 3])
(1, 2, 3)
```

tuple()函数的功能与 list 函数基本上是一样的：以一个序列作为参数并把它转换为元组。如果参数就是元组，那么该参数就会被原样返回。使用“=”将一个元组复制给变量，就可以创建一个元组变量。

```
>>> a_tuple = ('a', )
>>> b_tuple = ('a', 'b', 'c')
>>> c_tuple = ()
```

如同使用 list()函数将序列转换为列表一样，也可以使用 tuple()函数将其他类型序列转换为元组。

```
>>> print tuple('abcdefg')
('a', 'b', 'c', 'd', 'e', 'f', 'g')
>>> s = [1, 2, 3, 4]
>>> tuple(s)
(1, 2, 3, 4)
```

对于元组而言，只能使用 del 命令删除整个元组对象，而不能只删除元组中的部分元素，因为元组属于不可变序列。

3.2.2 元组的基本操作

元组其实并不复杂，除了创建元组和访问元组元素之外，支持索引访问、切片操作、连接操作、重复操作、成员资格操作、比较运算符操作，以及求元组长度、最大值、最小值等。

例 3-11 元组的基本操作示例。

```
>>> s1 = (1, 2, 1)
>>> s2 = ('a', 'x', 'y', 'z')
>>> len(s1)
3
>>> len(s2)
4
>>> max(s1)
2
>>> min(s2)
'a'
>>> sum(s1)
4
>>> sum(s2)
Traceback (most recent call last):
  File
"< ipython - input - 77 - bb5a6cd66c38 >",
line 1, in < module >
    sum(s2)
TypeError: unsupported operand type(s)
for + : 'int' and 'str'
>>> s1[0:2]
(1, 2)
```

元组的分片还是元组，就像列表的分片还是列表一样。

3.2.3 元组与列表的区别

列表属于可变序列，可以随意地修改列表中的元素值以及增加和删除列表元素，而元组属于不可变序列，元组中的数据一旦定义就不允许通过任何方式更改。因此，元组没有提供append()、extend()和 insert()等方法，无法向元组中添加元素；同样，元组也没有 remove()和pop()方法，也不支持对元组元素进行 del 操作，不能从元组中删除元素，只能使用 del 命令删除整个元组。元组也支持切片操作，但是只能通过切片来访问元组中的元素，而不支持使用切片来修改元组中元素的值，也不支持使用切片操作来为元组增加或删除元素。

元组的访问和处理速度比列表更快，如果定义了一系列常量值，主要用途仅是对它们进行遍历或其他类似用途，而不需要对其元素进行任何修改，那么一般建议使用元组而非列表。可以认为元组对不需要修改的数据进行了“写保护”，从内在实现上不允许修改其元素值，从而使得代码更加安全。

另外，作为不可变序列，与整数、字符串一样，元组可用作字典的键，而列表则永远都不能当作字典键使用，因为列表不是不可变的。

最后，虽然元组属于不可变列表，其元素的值是不可改变的，但是如果元组中包含序列，情况就略有不同，例如：

```
>>> s = ([1, 2], 3)
>>> s[0][0] = 5
>>> s
([5, 2], 3)
>>> s[0].append(8)
>>> s
([5, 2, 8], 3)
>>> s[0] = s[0] + [10]
Traceback (most recent call last):
  File "< ipython - input - 81 - 746f13e9d0fd >", line 1, in < module >
    s[0] = s[0] + [10]
TypeError: 'tuple' object does not support item assignment
>>> s
([5, 2, 8], 3)
```

3.2.4 生成器推导式

从形式上看，生成器推导式与列表推导式非常接近，只是生成器推导式使用圆括号而不是列表推导式所使用的方括号。与列表推导式不同的是，生成器推导式的结果是一个生成器对象而不是列表，也不是元组。使用生成器对象的元素时，可以根据需要将其转化为列表或元组，也可以使用生成器对象的 next()方法进行遍历，或者直接将其作为迭代器对象来使用。但是无论用哪种方法访问其元素，当所有元素访问结束后，如果需要重新访问其中的元素，必须重新创建该生成器对象。本章涉及的新函数如表 3-2 所示。

表 3-2 本章涉及的新函数

函 数	描 述
cmp(x,y)	比较两个值
len(seq)	返回序列的长度
list(seq)	把序列转换成列表
max(args)	返回序列或者参数集合中的最大值
min(args)	返回序列或者参数集合中的最小值
reversed(seq)	对序列进行反向迭代
sorted(seq)	返回已排序的包含 seq 所有元素的列表
tuple(seq)	把序列转换成元组

例 3-12 生成器推导式操作示例。

```
>>> s = [(i + 2) ** 2 for i in range(10)]
>>> s
[4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
>>> tuple(s)
(4, 9, 16, 25, 36, 49, 64, 81, 100, 121)
>>> list(s)
[4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
>>> t = ((i + 2) ** 2 for i in range(10))
>>> t
<generator object <genexpr> at 0x000000000BB51168>
>>> list(t)
[4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
>>> s.next()
4
>>> s.next()
9
>>> s.next()
16
>>> for i in [(i + 2) ** 2 for i in range(10)]:
    print i,
4 9 16 25 36 49 64 81 100 121
```

3.3 字典

字典是“键-值对”的无序可变序列。字典中的每个元素由“键”和“值”(又称数据项)两部分组成,“键”是关键字,“值”是与关键字有关的数据。定义字典时,“键”与“值”之间用冒号(:)分隔,相邻元素之间用逗号(,)分隔,所有的元素放在一对大括号“{}”中。空字典(不包括任何项)由一对大括号组成,即{}。其基本形式如下:

```
{键 1:值 1,[键 2:值 2, ..., 键 n:值 n]}
```

例如:phonebook={'Alice':'1234','Beth':'1902','Camille':'5678'}

在上例中,名字是键,电话号码是值。

“键”必须是不可变对象,“键”在字典中必须是唯一的,“值”可以是不可变对象或可变对象。字典中的“键”可以是 Python 中任意不可变数据,例如整数、实数、复数、字符串、元组等,但不能使用列表、集合、字典作为字典的“键”,因为这些类型的对象是可变的。另外,字典中的“键”不允许重复,而“值”是可以重复的。

3.3.1 字典创建与删除

字典的创建有如下几下几种方法。

1. 直接键入

```
>>> d1 = {}
>>> d2 = {'A':65, 'B':66, 'C':67,'A':"Hello"}
>>> d3 = {'A':65, 'B':66, 'C':67, 1:100, 1.0:200}
>>> d1
{}
>>> d2
{'A': 'Hello', 'B': 66, 'C': 67}
>>> d3
{1: 200, 'A': 65, 'B': 66, 'C': 67}
```

在字典 d2 中,写入了两个键都是'A'的元素,系统将用最后一个键值对取代前一个,以保证键的唯一性。在字典 d3 中,写入了整数 1 和浮点数 1.0 两个键,由于系统把 1 和 1.0 当作是同一个键,所以,系统用最后一个键值对代替了前一个。

2. 用 dict()函数创建字典

可以用 dict()函数,通过其他映射(比如其他字典)或者(键,值)这样的序列对建立字典。

```
>>> items = [('name', 'Lucy'), ('age', 20)]
>>> d = dict(items)
>>> d
{'age': 20, 'name': 'Lucy'}
>>> d['name']
'Lucy'
```

dict()函数也可以通过关键字参数来创建字典，如下例所示：

```
>>> d = dict(name = 'Lucy', age = 20)
>>> d
{'age': 20, 'name': 'Lucy'}
```

还可以以给定内容为“键”，创建“值”为空的字典：

```
>>> d1 = dict.fromkeys(['name', 'age', 'sex'])
>>> d2
{'age': None, 'name': None, 'sex': None,}
```

当不再需要某个字典时，可以使用 del 命令删除整个字典。

3.3.2 访问字典的键和值

与列表和元组类似，可以使用下标的方式来访问字典中的元素，但不同的是字典的下标是字典的“键”，而列表和元组访问时下标必须是整数值。使用下标的方式访问字典“值”时，若指定的“键”不存在则抛出异常。

```
>>> d = {'name': 'Michelle', 'sex': 'female', 'age': 36}
>>> d['name']
'Michelle'
>>> d['tel']
Traceback (most recent call last):
  File "< ipython - input - 21 - 0cc1c7833f98 >", line 1, in < module >
    d['tel']
KeyError: 'tel'
```

比较安全的字典元素访问方式是字典对象的 get()方法。使用字典对象的 get()方法可以获取指定“键”对应的“值”，并且可以在指定“键”不存在的时候返回指定值，如果不指定，则默认返回 None。

```
>>> print d.get('tel')
None
>>> d['tel'] = d.get('tel', [])
>>> d['tel'].append(12345678)
>>> d
{'age': 36, 'name': 'Michelle', 'sex': 'female', 'tel': [12345678]}
```

使用字典对象的 items()方法可以返回字典的“键-值对”列表，使用字典对象的 keys()方法可以返回字典的“键”列表，使用字典对象的 values()方法可以返回字典的“值”列表。

```
>>> for item in d.items():
    print item
('age', 36)
('tel', [12345678])
('name', 'Michelle')
('sex', 'female')
>>> for key in d.keys():
        print key
```

```
age
tel
name
sex
>>> for key,value in d.items():
        print key,value
age 36
tel [12345678]
name Michelle
sex female
```

3.3.3 字典元素的添加与修改

当以指定“键”为下标为字典元素赋值时，若该“键”存在，则表示修改该“键”的值；若不存在，则表示添加一个新的“键-值对”，也就是添加一个新元素。

```
>>> d['age'] = 26
>>> d
{'age': 26, 'name': 'Michelle', 'sex': 'female', 'tel': [12345678]}
>>> d['address'] = 'Dalian'
>>> d
{'address': 'Dalian',
 'age': 26,
 'name': 'Michelle',
 'sex': 'female',
 'tel': [12345678]}
```

使用字典对象的 update()方法将另一个字典的“键-值对”一次性全部添加到当前字典对象，如果两个字典中存在相同的“键”，则以另一个字典的“值”为准对当前的字典进行更新。

```
>>> d.update({'emal':'123@#.mail.cn','address':'Beijing'})
>>> d
{'address': 'Beijing',
 'age': 26,
 'emal': '123@#.mail.cn',
 'name': 'Michelle',
 'sex': 'female',
 'tel': [12345678]}
```

当需要删除字典元素时，可以使用 del 命令删除字典中指定“键”对应的元素，或者使用字典对象的 clear()方法来删除字典中的所有元素，还可以使用字典对象的 pop()方法删除并返回指定“键”的元素，后面章节将详细介绍。

3.3.4 字典可用的函数与方法

1. 标准类型的内置函数

标准类型的内置函数：

type()函数：返回字典的类型。

str()函数：返回字典的字符串表示形式。

```
>>> d = {'name': 'Michelle', 'sex': 'female', 'age': 36}
>>> type(d)
dict
>>> str(d)
"{'age': 36, 'name': 'Michelle', 'sex': 'female'}"
```

2. 字典类型专用函数

字典类型专用函数有：

dict()函数：用来创建字典。不指出函数的参数，创建空字典。

len()函数：返回键值对的数目。这种专用函数也可以用在序列、集合对象上。

hash()函数：用来判断一个对象是否可以作为字典的键。可以的话，hash()函数返回一个整数值(哈希值)；不可以的话，会抛出异常。如果两个对象有相同的值，那么它们的返回值相同，而且用它们作为字典的键时，只取其值作为键，只有一个键值对。例如：

```
>>> a = b = 'A'                    #a、b同值
>>> hash(a)                        #返回相同哈希值
1913471194
>>> hash(b)                        #返回相同哈希值
1913471194
>>> d = {a:1,b:2}
>>> d                              #只有一个键,取后一个值
{'100': 2}
>>> hash(d)                        #字典d不可哈希,不能再作为键了
Traceback (most recent call last):
  File "<ipython-input-48-a37dc9dc2032>", line 1, in <module>
    hash(d)
TypeError: unhashable type: 'dict'
```

3. 字典类型专用方法

Python提供了大量的字典类型专用方法，各方法及功能如表3-3所示。

表3-3 字典类型专用方法及功能

方　法	功　能
clear()	删除字典中的所有元素
copy()	返回字典的一个浅复制副本
fromkeys(<序列>,val)	创建并返回一个新字典、以<序列>中元素做键，val指定值，不指定val，默认为None
get(<键>,d)	<键>在字典中，返回<键>对应的值，<键>不在字典中，返回d或没有返回
items()	返回一个包含字典中键值对元组的列表
keys()	返回一个包含字典中键的列表
pop(<键>[,d])	<键>在字典中，删除并返回<键>对应的键值对。<键>不在字典中，有d时，返回d；无d时，抛出异常

续表

方　法	功　能
popitem()	删除并返回一个键值对的元组，字典空时，返回异常
setdefault(＜键＞,d)	对在字典中的键，返回对应的值，参数 d 设置无效；不在字典中的键，设置键和值，返回设置的值，d 默认为 None
update(＜字典＞)	把字典的键值对添加到方法绑定的字典
values()	返回一个包含字典中值的列表

下面介绍表 3-3 中的方法。

(1) clear()方法

clear()方法用于清除字典中所有的项。这是个原地操作，没有返回值。

```
>>> d = {}
>>> d['name'] = 'Lucy'
>>> d['age'] = 20
>>> d
{'age': 20, 'name': 'Lucy'}
>>> returned_value = d.clear()
>>> d
{}
>>> print returned_value
None
```

考虑以下两种情况。

第 1 种情况：

```
>>> x = {}
>>> y = x
>>> x['key'] = 'value'
>>> y
{'key': 'value'}
>>> x = {}
>>> y
{'key': 'value'}
```

第 2 种情况：

```
>>> x = {}
>>> y = x
>>> x['key'] = 'value'
>>> y
{'key': 'value'}
>>> x.clear()
>>> y
{}
```

两种情况中，x 和 y 最初对应同一个字典，情况 1 中，通过将 x 关联到一个新的空字典来“清空”它，这对 y 一点影响也没有，它仍然是关联到原先的字典。如果真的想清空原始字典中所有的元素，必须使用 clear()方法。正如在情况 2 中所看到的，y 随后也被清空。

(2) copy()方法

copy()方法返回一个具有相同键值对的新字典(这个方法实现的是浅复制(shallow copy)，因为值本身就是相同的，而不是副本)。

```
>>> x = {'username': 'admin', 'machines': ['foo', 'bar', 'baz']}
>>> y = x.copy()
>>> y['username '] = 'mlh'
>>> y['machines'].remove('bar')
>>> y
{'username': 'mlh', 'machines': ['foo', 'baz']}
```

```
>>> x
{'username': 'admin', 'machines': ['foo', 'baz']}
```

可以看到，当在副本中替换值的时候，原始字典不受影响，但是，如果修改了某个值（原地修改，而不是替换），原始的字典也会改变，因为同样的值也存储在原始字典中（就像上面例子中的“machines”列表一样）。

避免这个问题的一种方法就是使用深复制（deep copy），复制其所包含的所有值。可以使用 copy 模块的 deepcopy()函数来完成操作。

```
>>> from copy import deepcopy
>>> d = { }
>>> d['name'] = ['Alfred', 'Bertrand']
>>> c = d.copy()
>>> dc = deepcopy(d)
>>> d['names'].append('Clive')
>>> c
{'names': ['Alfred', 'Bertrand', 'Clive']}
>>> dc
{'names': ['Alfred', 'Bertrand']}
```

(3) has_key()方法

has_key()方法可以检查字典中是否含有给出的键。表达式 d.has_key(k)相当于表达式 k in d。使用哪种方式很大程度上取决于用户的喜好。

```
>>> d = { }
>>> d.has_key('name')
False
>>> d['name'] = 'Eirc'
>>> d.has_key('name')
True
```

(4) items()方法和 iteritems()方法

items()方法将所有的字典元素以列表方式返回，这些列表项中的每一项都来自“键-值对”，但是元素在返回时没有特殊的顺序。

```
>>> d = {'name': 'Michelle', 'sex': 'female', 'age': 36}
>>> d.items()
[('age', 36), ('name', 'Michelle'), ('sex', 'female')]
```

iteritems()方法的作用大致相同，但是会返回一个迭代器对象而不是列表：

```
>>> d = {'name': 'Michelle', 'sex': 'female', 'age': 36}
>>> it = d.iteritems()
>>> it
<dictionary-itemiterator at 0xa4919a8>
```

(5) keys()方法和 iterkeys()方法

keys()方法：将字典中的键以列表形式返回。

iterkeys()方法：返回针对键的迭代器。

使用方法参考 items()方法和 iteritems()方法，此处不再赘述。

(6) values()方法和 itervalues()方法

values()方法：以列表形式返回字典中的值，与返回键的列表不同的是，返回值的列表中可以包含重复的元素。

itervalues()方法：返回值的迭代器。

```
>>> d = {}
>>> d[1] = 1
>>> d[2] = 2
>>> d[3] = 3
>>> d[4] = 1
>>> d.values()
[1, 2, 3, 1]
```

(7) pop()方法和 popitem()方法

pop()方法：当“键”在字典中时，删除并返回“键”对应的键值对，当“键”不在字典中，有参数 d 时，返回参数 d；无参数 d 时，抛出异常。

popitem()方法：从字典中删除一个键值对，并返回这个键值对的元组；字典为空时，返回异常，但是删除是随机的。

```
>>> d = {'name': 'Michelle', 'sex': 'female', 'age': 36}
>>> d.pop('sex')
'female'
>>> d
{'age': 36, 'name': 'Michelle'}
>>> it = d.popitem()
>>> list(it)
['age', 36]
>>> d
{'name': 'Michelle'}
```

(8) setdefault()方法

setdefault()方法：在某种程度上类似于 get()方法，就是能够获得与给定键相关联的值，除此之外，setdefalut()方法还能在字典中不含有给定键的情况下设定相应的键值。

```
>>> d = {}
>>> d.setdefault('name', 'N/A')
'N/A'
>>> d
{'name', 'N/A'}
>>> d['name'] = 'Michelle'
>>> d.setdefault('name', 'N/A')
'Michelle'
>>> d
{'name', 'Michelle'}
```

从上面的代码中可以看到，当键不存在的时候，setdefault()方法返回默认值并且相应地更新字典。如果键存在，那么就返回与其对应的值，但不改变字典。默认值是可选的，这点和 get()方法一样。如果不设定，会默认使用 None。

```
>>> d = {}
>>> print d.setdefault('name')
>>> d
{'name', None}
```

(9) copy()方法、get()方法和 update()方法

这 3 种方法功能意义明确,已在前面章节介绍过,此处不再赘述。

3.3.5 案例精选

案例 3-5 编写程序,使用人名作为键的字典,每个人用另一个字典来表示,其键'phone'和'city'分别表示他们的电话号码和城市。

本案例代码如下:

```
people = {
    'Alice': {
        'phone': '1234',
        'city': 'Sydney'
    }
    'Beth': {
        'phone': '1902',
        'city': 'London'
    }
    'Camille': {
        'phone': '5678',
        'city': 'Chicago'
    }
}
# 针对电话号码和城市使用的描述性标签,会在打印输出的时候用到
labels = {
    'phone': 'phone number',
    'city': 'city name'
}
name = raw_input('Name: ')
# 查找电话号码还是所在城市?使用正确的键:
# 使用正确的键:
if request  ==  'p': key = 'phone'
if request  ==  'a': key = 'city'
# 如果名字是字典中的有效键才打印信息:
if name in people: print "%S 'S %S IS %S. " % \
 (name, labels[key], people[name][key])
```

案例 3-5 运行结果如下:

```
Name: Beth
Phone number (p) or address (a)? p
Beth's phone number is 1902.
```

3.4 集合

集合是无序序列，且集合中元素不重复。在 Python 语言中，集合类型有两种：可变集合(set)和不可变集合(frozenset)。可变集合的元素是可以添加、删除的，而不可变集合的元素是不可以这样做的。可变集合是不可哈希的，不可以作为字典的键或其他集合的元素，而不可变集合是可哈希的，可以作为字典的键或其他集合的元素。

3.4.1 集合的定义

可变集合通过大括号中用逗号分隔的元素定义，基本形式如下：

```
{x1, [x2, ..., xn]}
```

其中，x_1，x_2，...，x_n 为任意可哈希对象。集合中的元素不可重复，且无序。

在 Python 中变量不需要提前声明其类型，直接将集合复制给变量即可创建一个集合对象。

```
>>> a = {1, 2}
>>> a.add(3)
>>> a
{1, 2, 3}
```

也可以使用 set()函数将列表、元组等其他可迭代对象转换为集合，如果原来的数据中存在重复元素，则在转换为集合的时候只保留一个。

```
>>> a_set = set(range(5,10))
>>> b_set = set([0, 1, 2, 3, 0, 1, 2, 3, 4])
>>> a_set
set([8, 9, 5, 6, 7])
>>> b_set
set([0, 1, 2, 3, 4])
>>> x = set()          #空集合
>>> x
set()
```

不可变集合通过创建 frozenset()来创建。

```
>>> s = frozenset('python')
>>> print type(s)
<type 'frozenset'>
>>> print s
frozenset(['h', 'o', 'n', 'p', 't', 'y'])
```

例 3-13 创建集合对象示例。

```
>>> {1, 2, 1}
{1, 2}
>>> {1, 'a', True}
{' ', ',', '1', '2', '3'}
>>> frozenset('1, 2, 3')
frozenset({' ', ',', '1', '2', '3'})
```

```
{1, 'a'}
>>> {1, 2, True}
{1, 2}
>>> set('1, 2, 3')
>>> set('Hello')
{'H', 'e', 'l', 'o'}
>>> {'a',[1, 2]}
TypeError: unhashable type: 'list'
```

3.4.2 集合的基本操作

1. 访问集合元素

访问集合中的元素是指检查元素是否是集合中的成员或通过遍历方法显示集合内的成员。由于集合本身是无序的,所以不能为集合创建索引或切片操作。

例 3-14 访问集合元素示例。

```
>>> s = set(['A', 'B', 'C', 'D'])
>>> 'A' in s
True
>>> 'a' not in s
True
>>> for i in s:
        print i
A
C
B
D
```

2. 集合的更新

集合的更新包括增加、修改、删除集合的元素等。可以使用操作符或集合的内置方法来实现集合的更新动作。

例 3-15 更新集合元素示例。

```
>>> s = set(['A', 'B', 'C', 'D'])
>>> s = s|set('Python')                    #使用操作符"|"
>>> s
{'A', 'B', 'C', 'D', 'P', 'h', 'n', 'o', 't', 'y'}
>>> s.add('ABC')                           #add()方法,增加集合元素
>>> s
{'A', 'ABC', 'B', 'C', 'D', 'P', 'h', 'n', 'o', 't', 'y'}
>>> s.remove('ABC')                        #remove()方法,删除集合元素
>>> s
{'A', 'B', 'C', 'D', 'P', 'h', 'n', 'o', 't', 'y'}
>>> s.update('ABCDEF')#修改集合元素        #update()方法,修改集合元素
>>> s
{'A', 'B', 'C', 'D', 'E', 'F', 'P', 'h', 'n', 'o', 't', 'y'}
>>> del s                                  #删除集合 s
>>> s
NameError: name 's' is not defined
```

注意:增加、修改、删除集合的元素只针对可变集合,对于不可变集合,实施这些操作将引发异常。例如:

```
>>> t = frozenset(['A', 'B', 'C', 'D'])
>>> t.add('E')
Traceback (most recent call last):
  File "<ipython-input-27-0311d11cb592>", line 1, in <module>
```

```
    t.add('E')
AttributeError: 'frozenset' object has no attribute 'add'
```

3.4.3 集合可用的操作符

集合类型操作符分为标准类型操作符、集合类型专用操作符、仅适用于可变集合的专用操作符。

1. 标准类型操作符

这个类型的操作符是标准类型操作符，共有 8 个操作符，用于集合与元素或集合与集合的关系判断上。

(1) 成员关系(in 和 not in)。已经在集合的基本操作中介绍过，此处不再赘述。

(2) 子集与超集。一个集合是另一个集合的子集表示：前者中元素都在后者中，且后者中有或没有元素不在前者的集合中。如果说严格子集，就是后者必须有元素不在前者中。

对于两个集合 s 和 t，如果 s 是 t 的子集(严格子集)，则 t 是 s 的超集(严格超集)。

有 4 个运算符用于子集(＜和＜＝)与超集(＞和＞＝)。

例 3-16 子集与超集示例。

```
>>> s = set(['A', 'B', 'C', 'D'])
>>> t = frozenset(['D', 'C', 'B', 'A'])
>>> s < t
False
>>> s > t
False
>>> s < = t
True
>>> s = set(['A', 'B', 'C', 'D', 'E'])
>>> s > = t
True
```

(3) 集合等价与不等价(＝＝和!＝)。集合的等价是指，不同类型或同类型的两个集合，一个集合的所有元素都在另一个集合中，反之亦然。或者说，一个集合是另一个集合的子集，若反过来也成立，则这两个集合等价。

集合的等价与不等价和元素在集合中的顺序无关，集合的特点之一就是无序的。

例 3-17 集合等价与不等价示例。

```
>>> s = set(['A', 'B', 'C', 'D'])
>>> t = frozenset(['D', 'C', 'B', 'A'])
>>> s == t
True
>>> s != t
False
```

2. 集合类型专用操作符

这类集合类型专用操作符实际上是针对集合与集合的运算会产生运算结果，共有 4 个操作符(|、&、-和^)。

对于这 4 个操作符，如果操作符两边的集合是同类型的，产生的集合依然是该类型的；但是当两边的集合类型不一致时，结果集合的类型与左操作对象一致。

(1) 并操作符(|)。并操作产生一个新集合，称为并集。新集合的元素是参与操作的两个集合的所有元素，即属于两个集合之一的成员，并操作有一个完成同样功能的方法 union()。

例 3-18 集合并操作示例。

```
>>> s = set(['A', 'B', 'C', 'D', 1])
>>> t = frozenset(['A', 'B', 'E','F', -1])
>>> s|t
{-1, 1, 'A', 'B', 'C', 'D', 'E', 'F'}
>>> t|s
frozenset({-1, 1, 'A', 'B', 'C', 'D', 'E', 'F'})
>>> s.union(t)
{-1, 1, 'A', 'B', 'C', 'D', 'E', 'F'}
>>> t.union(s)
frozenset({-1, 1, 'A', 'B', 'C', 'D', 'E', 'F'})
```

(2) 交操作符(&)。交操作产生一个新集合,称为交集。新集合的元素是参与操作的两个集合的共同元素,同样有相应的等价方法 intersection()。

例 3-19 集合交操作示例。

```
>>> s = set(['A', 'B', 'C', 'D', 1])
>>> t = frozenset(['A', 'B', 'E','F', -1])
>>> s&t
{'A', 'B'}
>>> t&s
frozenset({'A', 'B'})
>>> s.intersection(t)
{'A', 'B'}
>>> t.intersection(s)
frozenset({'A', 'B'})
```

(3) 差补操作符(-)。假设参加操作的集合是 s 和 t,s 与 t 的差补是只属于 s 而不属于 t 的元素,反过来说,t 与 s 的差补是只属于 t 而不属于 s 的元素,这个运算符的等价方法为 difference()。

例 3-20 集合差补操作示例。

```
>>> s = set(['A', 'B', 'C', 'D', 1])
>>> t = frozenset(['A', 'B', 'E','F', -1])
>>> s-t
{1, 'C', 'D'}
>>> t-s
frozenset({-1, 'E', 'F'})
>>> s.difference(t)
{1, 'C', 'D'}
>>> t.difference(s)
frozenset({-1, 'E', 'F'})
```

(4) 对称差分操作符(^)。假设参加操作的集合是 s 和 t,s 与 t 进行对称差分操作的结果集是所有属于集合 s 和集合 t,并且不同时属于集合 s 和集合 t 的元素,这个运算符的等价方法为 symmetric_difference()。

例 3-21 集合对称差分操作示例。

```
>>> s = set(['A', 'B', 'C', 'D', 1])
```

```
>>> t = frozenset(['A', 'B', 'E','F', -1])
>>> t^s
frozenset({-1, 1, 'C', 'D', 'E', 'F'})
>>> s^t
{-1, 1, 'C', 'D', 'E', 'F'}
>>> s.symmetric_difference(t)
{-1, 1, 'C', 'D', 'E', 'F'}
>>> t.symmetric_difference(s)
frozenset({-1, 1, 'C', 'D', 'E', 'F'})
```

3. 4个复合操作符

4个复合操作符是上面产生新集合的4个操作符(|、&、-和^)分别与赋值符相结合构成的增量赋值操作符,它们是|=、&=、-=和^=。

如果有集合s和t,则:

s|=t等价于s=s|t;

s&=t等价于s=s&t;

s-=t等价于s=s-t;

s^=t等价于s=s^t。

例3-22 4个复合操作符示例。

```
>>> s = set(['A', 'B', 'C', 'D', 1])
>>> t = frozenset(['A', 'B', 'E','F', -1])
>>> t | = s
>>> t
frozenset({-1, 1, 'A', 'B', 'C', 'D', 'E', 'F'})
>>> t = frozenset(['A', 'B', 'E','F', -1])
>>> t & = s
>>> t
frozenset({'A', 'B'})
```

3.4.4 案例精选

案例3-6 学校举办运动会,计算机学院有5人参加长跑比赛a=['Lucy','Adward','John','Tom','Aaron'],有6人参加跳远比赛b=['John','Ben','Aaron','Carl','Ellis','bonnie'],有3人参加铅球比赛c=['Aaron','Ellis','Alice'],同时参加长跑和跳远的是哪些同学,同时参加三项比赛的是哪些同学?计算机学院参加运动会的同学有哪些?

```
>>> a = ['Lucy', 'Adward', 'John', 'Tom', 'Aaron']
>>> b = ['John', 'Ben', 'Aaron','Carl', 'Ellis','bonnie']
>>> c = ['Aaron', 'Ellis', 'Alice']
>>> ab = set(a) & set(b)
>>> abc = set(a) & set(b) & set(c)
>>> total = set(a) | set(b) | set(c)
>>> print "同时参加长跑和跳远的同学是:", ab
同时参加长跑和跳远的同学是: set(['Aaron', 'John'])
>>> print "同时参加三项比赛的同学是:", abc
```

```
同时参加三项比赛的同学是: set(['Aaron'])
>>> print "计算机学院参加运动会的同学是:", total
计算机学院参加运动会的同学是: set(['Ellis', 'Adward', 'Lucy', 'bonnie', 'Aaron', 'Carl', 'Tom',
'Ben', 'John', 'Alice'])
```

习题 3

一、单选题

1. Python 语句 print type([1,2,3,4])的运行结果是(　　)。

A. <type 'list'>　　B. <type 'tuple'>　　C. <type 'set'>　　D. <type 'dict'>

2. Python 语句 print type((1,2,3,4))的运行结果是(　　)。

A. <type 'list'>　　B. <type 'tuple'>　　C. <type 'set'>　　D. <type 'dict'>

3. Python 语句 print type({1,2,3,4})的运行结果是(　　)。

A. <type 'list'>　　B. <type 'tuple'>　　C. <type 'set'>　　D. <type 'dict'>

4. Python 语句 nums = set([1,2,2,3,3,3,4]); print len(nums)的运行结果是(　　)。

A. 1　　B. 2　　C. 3　　D. 4

5. Python 语句 s='Hello'; print s[1: 3]的运行结果是(　　)。

A. Hel　　B. He　　C. ell　　D. el

6. Python 语句 s1 = [1,2,3]; s2 = s1; s1[1] = 0; print s2 的运行结果是(　　)。

A. [1,2,3]　　B. [0,2,3]　　C. [1,0,3]　　D. 以上都不对

7. Python 语句 s={'a',1,'b',2}; print s['b']的运行结果是(　　)。

A. 语法错　　B. 'b'　　C. 1　　D. 2

8. Python 语句 s=[a,b,c]; s * 3 的运行结果是(　　)。

A. [a,a,a,b,b,b,c,c,c]　　B. [a,b,c,a,b,c,a,b,c]

C. [[a,a,a],[b,b,b],[c,c,c]]　　D. [[a,b,c],[a,b,c],[a,b,c]]

9. Python 语句 s1=[a,b,c,d]; s2=[e,f]; print(len(s1+s2))的运行结果是(　　)。

A. 4　　B. 5　　C. 6　　D. 7

10. Python 语句 d={1: 'x',2: 'y',3: 'z'}; del d[1]; del d[2]; d[1]='A'; print len(d)的运行结果是(　　)。

A. 0　　B. 1　　C. 2　　D. 3

11. Python 语句 fruits = ['apple','banana','peach']; fruits[-1][-1]的运行结果是(　　)。

A. 'e'　　B. 'a'　　C. 'h'　　D. 'p'

二、多项选择题

1. 关于 a or b 的描述正确的是(　　)。

A. 如果 a=True,b=True,则 a or b 等于 True

B. 如果 a=True,b=False,则 a or b 等于 True

C. 如果 a=True,b=False,则 a or b 等于 False

D. 如果 a=False,b=False,则 a or b 等于 False

2. 以下能够创建字典的 Python 语句是(　　)。

A. dict1＝{}　　B. dict2＝{1：3}

C. dict3＝dict([1,2],[3,4])　　D. dict4＝dict(([1,2],[3,4]))

三、判断题

1. 字典的键不是唯一的,即一个字典中可以出现两个以上的名字相同的键。(　　)

2. 对于可变集合,可以增加、修改、删除集合的元素。(　　)

3. 集合与集合的运算有并操作(|)、交操作(&)、差补操作(-)和对象差分操作(^),这些操作要求是同类型的集合。(　　)

4. 针对序列对象,可以使用索引和切片操作,当索引值出界时,将抛出异常。(　　)

5. 列表、字典、集合属于可变序列,元组、字符串属于不可变序列。(　　)

四、上机实践

员工信息包括:员工编号(ID),姓名(Name),职务(Title),电话(Phone),试编写程序,能够完成以下功能:

(1) 添加新员工信息。

(2) 列表打印所有员工信息。

(3) 输入一个员工编号,输出该员工所有信息。