第 3 单元　容　　器

通常，容器（container）是指用来包装或装载物品的器件（如箱、罐、坛）。在计算机程序中，容器是用来存储和组织对象的对象，可以在容器中存储基本类型或任何类类型的对象。

根据容器的存储特点、操作方式，容器可以分为不同的类型。如表 1.1 所示，Python 提供了三大类内置的容器：序列（sequence）、字典（dictionary）和集合（set）。表 3.1 为它们的粗略比较。

表 3.1　序列、字典和集合的粗略比较

容器名称		边界符	元素形式	类型标识符	可变性	元素分隔符	元素互异性	元素有序性
序列	列表	[…]	对象	list	可变	,	无	位置有序
	元组	(…)	对象	tuple	不可变			
	字符串	'…'/"…"/'''…'''/"""…"""	字符	str	不可变	无		
字典		{…}	键-值对	dict	键不可变 值可变	,(元素间) :(键-值间)	键唯一	无
集合		{…}	对象	set	可变	,	有	
				frozenset	不可变			

3.1　序 列 容 器

Python 有元组（tuple）、列表（list）和字符串（string）3 种内置序列容器。

3.1.1　序列对象的构建

列表、元组和字符串都可以用如下两种方式构建对象：使用字面量直接构建和用构造函数构建。

1. 使用字面量构建

不管是元组，还是列表或字符串，它们创建对象的方法相似且非常简单，只要用边界符将序列元素括起来，就成为序列对象。当一个序列有多个元素时，元素之间应使用合法的分隔符分隔。

代码 3-1　创建合法的字符串对象。

```
>>> 'abcd1234'                    #创建一个普通字符串对象
'abcd1234'
```

```
>>> """abcdefghijk
lmnop"krs"w'123'"""                    #含有单撇号和双撇号的多行字符串,用三撇号做边界符
'abcdefghijk\nlmnop"krs"w\'123\''
>>>  ''                                #一个空字符串对象
''
```

说明：

（1）Python 字符串是由单个字符为元素组成的序列。这些单个字符是键盘上可以打出的任何字符，也可以是转义字符。

（2）三撇号字符串与单撇号字符串和双撇号字符串的区别在于，它可以使多行的、可以直接包含换行，也可以直接包含单撇号和双撇号。

代码 3-2 创建合法的列表对象。

```
>>> ['ABCDE','Hello',"ok",'''Python''',123]     #创建一个列表对象
['ABCDE', 'Hello', 'ok', 'Python', 123]
```

代码 3-3 创建合法的元组对象。

```
>>> ('ABCDE','Hello',"ok",'''Python''',123)     #创建一个元组对象
 ('ABCDE', 'Hello', 'ok', 'Python', 123)
>>> (123,)                                     #要创建只有一个元素的元组，最后要加一个分隔符
123
>>> 1,2,3                                      #圆括号可以省略
(123,)
```

非但如此，还可以用变量指向这些用符号常量构建的序列对象。

代码 3-4 用变量指向序列对象。

```
>>> str1 =  'abcd1234'                                    #一个普通字符串对象,并用变量 str1 指向它
>>> lIst1 = ['ABCDE','Hello',"ok",'''Python''',123] #创建一个列表对象，并用变量 list1 指向它
>>> tup1 = ('ABCDE','Hello',"ok",'''Python''',123)  #创建一个元组对象,并用变量 tup1 指向它
```

2. 用构造方法构建序列对象

用构造方法构建序列对象的语法如下。

```
list(iterable)          #用可枚举对象 iterable 构造一个列表对象
tuple(iterable)         #用可枚举对象 iterable 构造一个元组对象
str(字符串字面量)        #用可枚举对象 iterable 构造一个字符串对象
```

当参数缺省时，构建的是一个空序列对象。

代码 3-5 用构造方法构建序列对象示例。

```
>>> list1 = list()                          #构造一个空列表对象
>>> list1
[]
>>> list2 = list("I like Python,2017")      #用字符串（可枚举）对象构造一个空列表对象
>>> list2
```

```
['I',' ','l','i','k','e',' ','P','y','t','h','o','n',',','2','0','1','7']
>>> list3 = list(range(1,20,3))            #用 range()函数返回的递增整数序列构造一个列表对象
>>> list3
[1, 4, 7, 10, 13, 16, 19]
>>>
>>> t1 = tuple('a','b','c',1,2,3)          #错误，不能用多个参数
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    ti = tuple('a','b','c',1,2,3)
TypeError: tuple() takes at most 1 argument (6 given)
>>> t1=tuple(['a','b','c',1,2,3])          #正确
>>> t1
('a', 'b', 'c', 1, 2, 3)
>>> t2 = tuple(range(1,20,3))              #用 range()函数返回的递增整数序列构造一个元组对象
>>> t2
(1, 4, 7, 10, 13, 16, 19)
>>> type(list1)                            #测试类型
<class 'list'>
>>> list2 = list('Hello,Python')           #将字符串用函数 list()转换为列表
>>> list2
['H', 'e', 'l', 'l', 'o', ',', 'P', 'y', 't', 'h', 'o', 'n']
```

3.1.2 序列通用操作

不管是哪种序列对象，都可以使用下列操作符进行操作。

1. 序列对象的元素索引与切片

序列的基本特征是有序，即序列中的元素包含了一个从左到右的顺序，这个顺序用元素在序列中的位置偏移量表示。这个位置偏移量，也称为序列号或下标（index，索引），可以分为如图 3.1 所示的正向和反向两个体系。

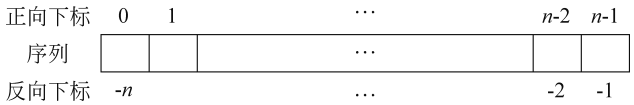

图 3.1　序列的正向索引与反向索引

注意：正向下标最左端为 0，向右按 1 递增；反向下标最右端为 1，向左按-1 递减。

使用索引/切片操作符（[ ]）可以对序列进行索引/切片操作。

1）索引

索引（index）是快捷获取信息的手段。在序列容器中，索引一个元素的操作由索引操作符（[]，也称为下标操作符）和下标进行。

代码 3-6　序列索引举例。

```
>>> list1 = ['ABCDE','Hello',"ok",'''Python''',123]
>>> list1[3]
```

```
'Python'
>>> s1 = 'abcd1234'
>>> s1[-3]
'2'
>>> t1 = ('ABCDE','Hello',"ok",'''Python''',123)
>>> t1[-5]
'ABCDE'
```

2）切片

切片操作的格式如下。

```
序列对象变量 [ 起始下标 : 终止下标 : 步长 ]
```

说明：

（1）切片就是在序列中划定一个区间[起始下标 : 终止下标)，并按步长选取元素，但不包括终止下标指示的元素。

（2）步长的默认值为 1，即不指定步长，就是获取指定区间中的每个元素，但不包括终止下标指示的元素。

（3）起始下标和终止下标缺省或表示为 None，分别默认为起点和终点。

（4）起始在左、终止在右时，步长应为正；起始在右、终止在左时，步长应为负。否则切片为空。

代码 3-7 序列切片举例。

```
>>> list1 = ['ABCDE','Hello',"ok",'''Python''',123]
>>> list1[:]                          #起始、终止、步长都缺省
['ABCDE', 'Hello', 'ok', 'Python', 123]
>>> list1[None:]                      #起始为 None，其他缺省
['ABCDE', 'Hello', 'ok', 'Python', 123]
>>> list1[::2]                        #起始、终止缺省，步长为 2
['ABCDE', 'ok', 123]
>>> list1[1:3]                        #步长缺省，起始、终止分别为 1、3
['Hello', 'ok']
>>> list1[-5:-2]                      #反向索引：起始在左，步长为正
['ABCDE', 'Hello', 'ok']
>>> list1[2:2]                        #起始与终止相同，取空
[]
>>> list1[2:3]
['ok']
>>> s1 = "ABCDEFGHIJK123"
>>> s1[-2:-10:2]                      #反向索引：起始在右，步长为正，将得空序列
''
>>> s1[-2:-11:-2]                     #反向索引：起始在右，步长为负
'2KIGE'
>>> s1[11:2:-2]                       #正向索引：起始在右，步长为负
'1JHFD'
```

2. 序列对象连接与重复操作

Python 用连接操作符（+）和重复操作符（*）进行序列的连接和重复操作，格式如下。

```
序列 1 + 序列 2
```

```
序列 * 重复次数
```

代码 3-8 序列连接与重复举例。

```
>>> list1 = ['ABCDE','Hello',"ok",'''Python''',123]
>>> list2 = ['xyz',567]
>>> list1 + list2
['ABCDE', 'Hello', 'ok', 'Python', 123, 'xyz', 567]
>>> list2 * 3
['xyz', 567, 'xyz', 567, 'xyz', 567]
>>> s1 = "ABCDEFGHIJK123"
>>> s2 = 'abcdfg'
>>> s1 + s2
'ABCDEFGHIJK123abcdfg'
>>> s2 * 3
'abcdfgabcdfgabcdfg'
```

3. 序列对象判定操作

序列对象的判定操作包括如下 5 类，它们均得到 bool 值：True 或 False。

（1）对象值比较操作符：>、>=、<、<=、==和!=。

（2）对象身份判定操作符：is 和 is not。

（3）成员属于判定操作符：in 和 not in。

（4）布尔操作符：not、and 和 or。

（5）判定序列对象的元素是否全部或部分为 True 的内置函数：all()和 any()。

代码 3-9 对序列进行判定操作举例。

```
>>> list1 = ['ABCDE','Hello',"ok",'''Python''',123]; list2 = ['xyz',567]
>>> list1 == list2
False
>>> list1 != list2
True
>>> list1 > list2
False
>>> list1 < list2
True
>>> 'ABCDE' in list1
True
>>> ['xyz',567] is list2
False
>>> list3 = ['xyz',567]; list3 == list2
True
```

```
>>> list3 is list2
False
>>> t1 = (1,2,3); t2 = (1,2,3); t3 = (1,2,0)
>>> t1 == t2
True
>>> t1 is t2
False
>>> t1 = t2; t1 is t2
True
>>> all(t3)
False
>>> any(t3)
True
```

说明：

（1）相等比较（==）与是否比较（is）的不同在于，相等比较的是值，是否比较的是 id（地址）。

（2）序列对象不是按照值存储的，即值相等，不一定是同一对象。

（3）字符串之间的比较，是按正向下标，从 0 开始，以对应字符的码值（如 ASCII 码值）作为依据进行的，直到对应字符不同，或所有字符都相同，才能决定大小，或是否相等。

4. 获取序列对象的长度、最大值、最小值与和

下面 4 个 Python 内置函数，用于获取序列的有关数据。

len(obj)：返回对象 obj 的元素个数。

max(s)：返回序列 s 的最大值（仅限字符串或数值序列）。

min(s)：返回序列 s 的最小值（仅限字符串或数值序列）。

sum(s)：返回序列 s 的元素之和（仅限数值序列）。

代码 3-10 对序列求元素个数、最大元素、最小元素与和。

```
>>> list1 = ['ABCDE','Hello',"ok",'''Python''',123]; s1 = 'qwertyuiop'; t1 = (1.23,3.1416,1.414)
>>> len(list1)
5
>>> max(list1)                                    #错误，非数值序列、非字符串求最大值
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    max(list1)
TypeError: '>' not supported between instances of 'int' and 'str'
>>> max(s1)
'y'
>>> sum(s1)                                       #错误，非数值序列求和
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    sum(s1)
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> sum(t1)
5.7856
```

5. 序列元素排序

可以用内置函数 sorted()返回一个序列元素排序后的列表。该函数的格式如下。

```
sorted(序列对象[, key = 排序属性][, reverse = False/True])
```

说明：

（1）排序的前提是元素间可以相互比较，用术语 iterable（可迭代）表示。若一个序列中有不可相互比较的元素，就不可排序。

（2）一个序列中的元素对象可以有许多属性，要用 key 指定按照哪个属性排序。例如对字符串可以指定 str.lower。通常，对于字符串元素以及数值型元素对象，key 项可以缺省，默认按照数值排序。对于字符串对象，按照编码值（如 ASCII 码值）排序。

（3）sorted()函数默认按照升序排序，但可以用 reverse 的取值为 True/ False，决定是否反转。

（4）sorted()返回一个列表。

代码 3-11 序列元素排序。

```
>>> list1 = ['ABCDE','Hello',"ok",'''Python''']
>>> s1 = 'qwertyuiop'
>>> t1 = (1.23,3.1416,1.414); t2 = ('a','y',1,2,'n')
>>> sorted(list1,key = str.lower)
['ABCDE', 'Hello', 'ok', 'Python']
>>> sorted(s1)
['e', 'i', 'o', 'p', 'q', 'r', 't', 'u', 'w', 'y']
>>> sorted(s1,reverse = True)
['y', 'w', 'u', 't', 'r', 'q', 'p', 'o', 'i', 'e']
>>> sorted(t1)
[1.23, 1.414, 3.1416]
>>> sorted(t2)                #错误，t2 中含不可比较元素
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    sorted(t2)
TypeError: '<' not supported between instances of 'int' and 'str'
```

说明：产生一个错误的原因是无法对不可相互比较的序列元素排序。

6. 序列拆分

一个序列可以按照元素数量被拆分。下面分 3 种情形讨论。

1）变量数与元素数一致

当变量数与元素数一致时，将为每个变量按顺序分配一个元素。

代码 3-12 变量数与元素数一致时的序列拆分示例。

```
>>> t1 = ("zhang",'male',20,"computer",3,(70,80,90,65,95))
>>> name,sex,age,major,year,grade = t1
>>> name
'zhang'
>>> sex
'male'
>>> age
20
>>> major
'computer'
>>> year
3
>>> grade
(70, 80, 90, 65, 95)
```

2）变量数少于元素数

变量数与元素数不一致，将导致 ValueError。但是，用比序列元素个数少的变量拆分一个序列，可以获取一个子序列。办法是，在欲获取子序列的变量前加一个星号。

代码 3-13 在序列中获取一个子序列的拆分示例。

```
>>> grade = (70, 80, 90, 65, 95)
>>> first,*middles,last = sorted(grade)    #用 middles 获取数据排序后,再去掉两头的最高和最低成绩
>>> sum(middles)/len(middles)              #计算中间段的平均成绩
80.0
```

3）仅获取关心的元素

为了仅获取关心的元素，可以用匿名变量（_）进行虚读。

代码 3-14 在序列中安排部分虚读示例。

```
>>> t1 = ("zhang",'male',20,"computer",3,(70,80,90,65,95))
>>> name,_,_,*learningStatus = t1          #嵌入虚读的匿名变量
>>> name
'zhang'
>>> learningStatus
['computer', 3, (70, 80, 90, 65, 95)]
```

7. 序列遍历

遍历（traversal）是指按某条路径巡访容器中的元素，使每个元素均被访问一次，而且仅被访问一次。遍历的关键是设计巡访路径。对于序列这样的线性容器来说，最容易理解的遍历路径列表是通过对象值的迭代，形成一条遍历路径。为此可以使用如下 3 种 for 循环。

（1）用 len()函数计算出序列长度，用 range()函数产生一个索引序列控制 for 循环。

（2）隐匿列表长度，利用本身的序列控制 for 循环。

（3）用内置的 enumerate()将序列转换为索引序列控制 for 循环。

代码 3-15 遍历序列的 3 种办法示例。

```
>>> t1 = ('one', 'two', 'three')
#办法 1：利用 len()和 range()控制 for 循环
>>> for i in range (0,len(t1)):
  print (i,t1[i])
0 one
1 two
2 three
#办法 2：直接用 t1 序列控制 for 循环
>>> for i in t1:
  print (i)
one
two
three
#办法 3：用 enumerate()控制 for 循环
>>> for i, element in enumerate(t1):
  print(i,t1[i])
0 one
1 two
2 three
```

3.1.3 列表的个性化操作

列表的基本特征是有序和可变。有序是序列容器的共性。体现这个共性的操作已经介绍。表 3.2 是体现其可变性的操作函数。这些函数仅属于列表这个类型，是列表容器的另一方面属性。为体现类（类型）的属性，这种函数都被特别称为方法（method），并且要用圆点（.）操作符（也称为分量操作符）进行访问。可以看出，内置的序列操作函数，是将所操作对象作为参数，而列表个性化操作方法是作为操作对象的分量调用。

表 3.2 列表个性化操作的主要方法（设 aList = [3,5,7,5]）

方 法 名	功 能	参 数 示 例	执 行 结 果
aList.append(obj)	将对象 obj 追加到列表末尾	obj = 'a'	aList：[3,5,7,5, 'a']
aList.clear()	清空列表 aList		aList：[]
aList.copy()	复制列表 aList	bList = aList.copy() id(aList) id(bList)	bList：[3,5,7,5] 2049061251528 2049061251016
aList.count(obj)	统计元素 obj 在列表中出现的次数	obj = 5	2
aList.extend(seq)	把序列 seq 一次性追加到列表末尾	seq = ['a',8,9]	aList：[3,5,7, 'a',8,9]
aList.index(obj)	返回 obj 首个位置索引值；若无 obj，则抛出异常	obj = 5	1
aList.insert(index,obj)	将对象 obj 插入列表中下标 index 位置	index = 2,obj =8	aList：[3,5,8,7,5]
aList.pop(index)	移除 index 指定元素（默认尾元素），返回其值	index = 3	5, aList：[3,5,7]
aList.remove(obj)	移除列表中 obj 的第一个匹配项	obj = 5	aList：[3,7,5]
aList.reverse()	列表中元素进行原地反转		aList：[5,7,5,3]
aList.sort()	对原列表进行原地排序		aList：[3, 5, 5, 7]

下面从应用的角度，讨论列表的几种个性化操作。

1. 向列表增添元素

向列表增添元素，有如下 5 种办法。

（1）利用加号（+）。

代码 3-16 利用加号向序列添加元素。

```
>>> aList = [3,5,9,7];bList = ['a','b']
>>> aList += bList
>>> aList
[3, 5, 9, 7, 'a', 'b']
```

（2）利用乘号（*）。

代码 3-17 利用乘号向序列添加元素。

```
>>> aList = [3,5,9,7]
>>> aList * 3
[3, 5, 9, 7, 3, 5, 9, 7, 3, 5, 9, 7]
```

（3）用 append()方法向列表尾部添加一个对象。

代码 3-18 用 append()方法向列表尾部添加一个对象。

```
>>> aList = [3,5,9,7];bList = ['a','b']
>>> aList.append(bList)
>>> aList
[3, 5, 9, 7, ['a', 'b']]
>>> aList.append(True)
>>> aList
[3, 5, 9, 7, ['a', 'b'], True]
```

（4）用 extend()方法向列表尾部添加一个列表。

代码 3-19 用 extend()方法向列表尾部添加一个对象。

```
>>> aList = [3,5,9,7];bList = ['a','b']
>>> aList.extend(bList)
>>> aList
[3, 5, 9, 7, 'a', 'b']              #将这个结果与代码 3-17 的结果比较
```

（5）用 insert()方法将一个对象插入到指定位置。

代码 3-20 用 insert()方法将一个对象插入到指定位置。

```
>>> aList = [3,5,9,7];bList = ['a','b']
>>> aList.insert(2,bList)
>>> aList
[3, 5, ['a', 'b'], 9, 7]
>>> aList.insert(4,2)
>>> aList
[3, 5, ['a', 'b'], 9, 2, 7]
```

这 5 种办法有各有特色，但也有异曲同工之效。

2. 从列表中删除元素

从列表中删除元素，Python 有 del、remove、pop 三种操作。它们的区别如下。
（1）del 是根据索引（元素所在位置）来删除的。
（2）remove 是删除首个符合条件的元素。
（3）pop 返回的是弹出的那个数值。
代码 3-21　在列表中删除元素示例。

```
>>> aList = [3,5,7,9,8,6,2,5,7,1]
>>> del aList[3]
>>> aList
[3, 5, 7, 8, 6, 2, 5, 7, 1]
>>> aList.remove(7)
>>> aList
[3, 5, 8, 6, 2, 5, 7, 1]
>>> aList.remove(10)
Traceback (most recent call last):
  File "<pyshell#39>", line 1, in <module>
    aList.remove(10)
ValueError: list.remove(x): x not in list
>>> aList.pop(3)
6
>>> aList
[3, 5, 8, 2, 5, 7, 1]
```

使用时要根据具体需求选用合适的方法。

3. 赋值（=）与复制（copy()）

（1）赋值（=）只是对象的引用赋值，让另一个变量指向同一个对象。
（2）复制（copy()）是创建另一个同值对象。
代码 3-22　赋值（=）与复制（copy()）异同示例。

```
>> aList=[3,5,7,9]
>>> bList = aList
>>> bList
[3, 5, 7, 9]
>>> id(aList),id(bList)
(2788871571976, 2788871571976)
>>> cList = aList.copy()
>>> cList
[3, 5, 7, 9]
>>> id(aList),id(cList)
(2788871571976, 2788871570312)
```

3.1.4 字符串的个性化操作

1. Python 的驻留机制

字符串的一个重要特征是其内存驻留机制，简称字符串驻留（string interning）机制。意思就是，为每个取值相同的字符串，在内存中只保留一个副本。但这种驻留是有条件的。

（1）仅限数字、大小写拉丁字母和下画线组成的字符串可以驻留。

代码 3-23 对字符串组成字符的限制。

```
>>> a ='a0123456789012345678912345'
>>> b ='a0123456789012345678912345'
>>> a is b
True
>>> a = 'qwertyuiop[]asdfghjklzxcvbnm,./'          #含非规定字符
>>> b = 'qwertyuiop[]asdfghjklzxcvbnm,./'
>>> a is b
False
>>> a = "王"; b = "王"; a is b                      #含非规定字符
False
```

（2）编译期间就确定的字符串——静态字符串，采用驻留机制，但是，仅限于规定字符。

代码 3-24 静态字符串驻留示例。

```
>>> a = 'abcdef'                                   #静态字符串
>>> b = 'abc' + 'def'                              #静态字符串
>>> c = ''.join(['abc','def'])                     #动态字符串
>>> a,b,c
('abcdef', 'abcdef', 'abcdef')
>>> a is b, a is c
(True, False)
```

（3）字符串不能太长，特别是通过乘法运算符得到的字符串，长度必须小于 20。

代码 3-25 长度限制示例。

```
>>> a ='abc0123456789012345678912345abcdefaabc0123456789012345678912345abcdefef'
>>> b ='abc0123456789012345678912345abcdefaabc0123456789012345678912345abcdefef'
>>> a is b
True
>>> a ='abc0123456789012345678912345abcdefaabc0123456789012345678912345abcdefefaabc0123
4567890123456789123 45abcdefaabc0123456789012345678912345abcdefef'        #长字符串
>>> b ='abc0123456789012345678912345abcdefaabc0123456789012345678912345abcdefefaabc0123
4567890123456789123 45abcdefaabc0123456789012345678912345abcdefef'        #长字符串
>>> a is b
False
>>> b ='abcde'*4; a ='abcde'*4; a is b                                    #长度为 20
True
>>> b ='abcde'*4 +'a'; a ='abcde'*4 +'a'; a is b                          #长度为 21
False
```

（4）Python 的 sys 模块提供的 intern()方法可以强制两个取值相同的变量指向同一个对象。

代码 3-26 intern()应用示例。

```
>>> import sys
>>> a='abcdef!'; b='abcdef!'; a is b
False
>>> a = sys.intern(b); a is b
True
```

2. 字符串搜索与测试

1）字符串搜索

字符串搜索是在给定的区间[beg，end]搜索指定字符串，默认的搜索区间是整个字符串。Python 字符串的搜索方法如表 3.3 所示。

表 3.3 Python 字符串的搜索方法

方 法	功 能
s.count(str, beg=0, end=len(s))	返回区间内 str 出现的次数
s.endswith(obj, beg=0, end=len(s))	在区间内，检查字符串是否以 obj 结尾：是，则返回 True；否则返回 False
s.find(str, beg=0, end=len(s))	在区间内，检查 str 是否包含在 s 中：是，则返回开始的索引值；否则返回-1
s.index(str, beg=0, end=len(s))	跟 find()方法一样，只不过如果 str 不在 s 中会报一个异常
s.rfind(str, beg=0,end=len(s))	类似于 find()函数，不过是从右边开始查找
s.rindex(str, beg=0,end=len(s))	类似于 index()，不过是从右边开始
s.startswith(obj, beg=0,end=len(s))	在区间内，若以 obj 开头，返回 True；否则返回 False

2）字符串测试

字符串测试是判断字符串元素的特征，Python 字符串不划分区间的检查统计类操作方法如表 3.4 所示。

表 3.4 Python 字符串不划分区间的检查统计类操作方法

方 法	功 能
s.isalnum()	如果 s 非空且所有字符都是字母或数字则返回 True，否则返回 False
s.isalpha()	如果 s 至少有一个字符并且所有字符都是字母则返回 True，否则返回 False
s.isdecimal()	如果 s 只包含十进制数字则返回 True，否则返回 False
s.isdigit()	如果 s 只包含数字则返回 True，否则返回 False
s.islower()	如果 s 中包含有区分大小写的字符，并且它们都是小写，则返回 True，否则返回 False
s.isnumeric()	如果 s 中只包含数字字符则返回 True，否则返回 False
s.isspace()	如果 s 中只包含空格则返回 True，否则返回 False
s.istitle()	如果 s 是标题化的（见 title()）则返回 True，否则返回 False
s.isupper()	如果 s 中包含有区分大小写字符，并且它们都是大写，则返回 True，否则返回 False
s.isdecimal()	检查字符串是否只包含十进制字符，只用于 Unicode 对象

这些方法都比较简单，就不举例说明了。

3. 字符串修改

应当注意，字符串对象是不可变（immutable）序列对象。这种不可变意味着一旦创建就不可改变，不可在同一个位置进行赋值。但是，这并不妨碍创建一个新的字符串，而用原来字符串的变量指向它，就好像是把原来的字符串进行了改变似的。

表 3.5 列出了 Python 字符串的修改操作方法。

表 3.5　Python 字符串的修改操作方法

方　法	功　能
s.capitalize()	把字符串 s 的第一个字符大写
s.center(width)	返回一个原字符串居中并使用空格填充至长度 width 的新字符串
s.expandtabs(tabsize=8)	把字符串 s 中的 tab 符号转为空格，tab 符号默认的空格数是 8
s.ljust(width)	返回一个原字符串左对齐，并使用空格填充至长度 width 的新字符串
s.lower()	转换 s 中所有大写字符为小写
s.lstrip()	删除 s 首部的空格
s.rstrip()	删除 s 末尾的空格
s.strip([obj])	删除 s 首尾空格
s.maketrans(intab, outtab)	创建字符映射转换表。intab 表示需要转换的字符串；outtab 为转换的目标字符串
s.replace(str1, str2, num=s.count(str1))	把 s 中的 str1 替换成 str2，若指定 num，则替换不超过 num 次
s.rjust(width)	返回一个原字符串右对齐，并使用空格填充至长度 width 的新字符串
s.swapcase()	翻转 s 中的大小写
s.title()	返回“标题化”的 s，即所有单词都以大写开始，其余字母均为小写（见 istitle()）
s.translate(table, del="")	根据 table 给出的翻译表转换 s 的字符，要过滤掉的字符放到 del 参数中
s.upper()	转换 s 中的小写字母为大写
s.zfill(width)	返回长度为 width 的字符串，原字符串 s 右对齐，前面填充 0

这些方法的使用比较简单，就不举例说明了。

4. 字符串分隔与连接

表 3.6 给出 Python 字符串分隔和连接的方法。

表 3.6　Python 字符串分隔和连接的方法

方　法	功　能
s.split(str="", num=s.count(str))	返回以 str 为分隔符将 s 分隔为 num 个子字符串组成的列表，num 为 str 个数
s.splitlines()	返回在每个行终结处进行分隔所产生的行列表，并剥离所有行终结符
s.partition(str)	返回第一个 str 分隔的三个字符串元组：(s_pre_str,str,s_post_str)。 若 s 中不含 str，则 s_pre_str == s
s.rpartition(str)	类似于 partition()，不过是从右边开始查找
str.join(seq)	以 str 作为连接符，将 seq 中各元素（的字符串表示）连接为一个新的字符串

代码 3-27 字符串分隔与连接示例。

```
>>> s1 = "red/yellow/blue/white/black"
>>> list1 = s1.split('/')                    #返回用每个'/' 分隔子串列表
>>> list1
['red', 'yellow', 'blue', 'white', 'black']
>>>
>>> s1.partition('/')                        #返回用第一个'/'分隔为 3 个子串的元组
('red', '/', 'yellow/blue/white/black')
>>> s1.rpartition('/')                       #返回用最后一个'/'分隔为 3 个子串的元组
('red/yellow/blue/white', '/', 'black')
>>>
>>> s2 = '''red
yellow
blue
white
black'''
>>> s2.splitlines()                          #返回按行分隔的列表
['red', 'yellow', 'blue', 'white', 'black']

>>> '#'.join(list1)                          #用#连接各子串
'red#yellow#blue#white#black'
```

3.1.5 字符串编码与解码

1. 有关概念

在计算机底层，任何数据都是用 0、1 表示的。为了能用 0、1 表示文字，并且能共享，一些标准化组织制定了一些编码标准。下面介绍与 Python 字符串编码相关的概念。

1）ASCII 编码

ASCII（American Standard Code for Information Interchange，美国标准信息交换代码）由美国国家标准学会（American National Standard Institute，ANSI）制定，后被国际标准化组织（International Organization for Standardization，ISO）定为国际标准。它使用 7 位或 8 位二进制数组合来表示基于拉丁字母的语言文字符号，形成 128 或 256 种可能的字符，包括大写和小写拉丁字母、数字 0~9、标点符号、非打印字符（换行符、制表符等 4 个）以及控制字符（退格、响铃等）。这种字符在全世界范围内的应用是极为有限的。

2）Unicode 与 UTF

Unicode（统一码、万国码、单一码）是一种 2 字节计算机字符编码，为欧洲、非洲、中东、亚洲大部分国家文字的每个字符设定了统一并且唯一的二进制编码，以满足跨语言、跨平台进行文本转换与处理的要求。1990 年开始研发，1994 年正式公布。它比 ASCII 占用大一倍的空间，可以用 ASCII 表示的字符使用 Unicode 就是浪费。为了解决这个问题，出现了一些中间格式的字符集，被称为通用转换格式（Unicode Transformation Format，UTF）。

3）UTF-8

UTF-8（8-bit Unicode Transformation Format）是多种 UTF 格式中的一种，是一种变长

编码。例如，对于 ASCII 字符集中的字符，UTF-8 只使用 1B，并且与 ASCII 字符表示一样，而其他的 Unicode 字符转化成 UTF-8 需要至少 2B。

4）GBK

GBK（国标扩展）码是《汉字内码扩展规范》（Chinese Internal Code Specification）的简称，中华人民共和国全国信息技术标准化技术委员会 1995 年 12 月 1 日制定。由于 IBM 在编写 Code Page 的时候将 GBK 放在第 936 页，所以也叫 CP936。

5）字节序列

为了便于存储和传输，Python 内置了两种基本的二进制序列——字节序列：Python 3 引入的不可变 bytes 类型和 Python 2.6 添加的可变 bytearray 类型，对应的构造函数为 bytes() 和 bytearray()。它们的对象都是以 8b 为单位组织，每个元素 x（0≤x<256）为整数。为与字符串区别，在字面量之前要加个字符 b。例如，b'abc'、b'abc\'x"、b'abc"x'"、b"xyz"、b"""等。

2. 字符串编码与解码操作

简单地说，把字节序列变成人类可读的文本字符串就是解码，而把字符串变成用于存储或传输的字节序列就是编码。在 Python 3 中，字符串不再区分 ASCII 编码和 Unicode 编码，默认采用 UTF-8，并允许创建字符串时指定编码方式。表 3.7 为字符串编码与解码有关的方法。

表 3.7　字符串编码与解码有关的方法

（s：字符串变量；b：字节码变量；object：序列对象；encoding：编码格式；errors：错误控制）

方　法	功　能	说　明
str(object = b''.encoding='UTF-8', ='strict')	构造函数	如果出错默认报一个 ValueError 异常，除非 errors 指定的是 'ignore' 或者 'replace'
b.decode(encoding='UTF-8', ='strict')	解码字节码 b 为相应的字符串对象	
s.encode(encoding='UTF-8', errors='strict')	将 s 编码为字节码对象	

代码 3-28　字符串的编码与解码示例。

```
>>> s1 = '我喜欢 Python!'
>>> b1 = s1.encode(encoding = 'CP936')     #将 s1 按 CP936 格式编码为字节码
>>> b1
b'\xce\xd2\xcf\xb2\xbb\xb6Python!'
>>> b1.decode(encoding = 'CP936')          #将 b1 按 CP936 格式解码
'我喜欢 Python!'
```

3．字符串中的汉字

严格地说，str 其实是字节串，它是 Unicode 经过编码后的字节组成的序列。对 UTF-8 编码的 str'汉'使用 len()函数时，结果是 3，实际上，UTF-8 编码的'汉' == '\xE6\xB1\x89'。Unicode 才是真正意义上的字符串，对字节串 str 使用正确的字符编码进行解码后获得，并且 len(u '汉') == 1。

4．字符编码声明

在 Python 源代码文件中，如果要用到非 ASCII 字符，则需要在文件头部进行字符编码

的声明。字符编码声明的语法如下。

```
# coding: 编码名称
```

例如，采用 UTF-8，可以声明为

```
# coding: UTF-8
```

也可以写成

```
#***** coding: UTF-8*****
```

因为 Python 只检查#、coding 和编码代号，其他字符都不影响 Python 的判断。

另外，Python 中可用的字符编码很多，并且还有许多别名，还不区分大小写，例如，UTF-8 可以写成 u8。

另外需要注意的是，声明的编码必须与文件实际保存时用的编码一致，否则很大几率会出现代码解析异常。

3.1.6 字符串格式化与 format()方法

1. 字符串格式化表达式

1.5.2 节介绍了 print()函数的格式控制。由于 print()的输出就是创建字符流的过程，所以 print()的输出格式控制，就是字符串的格式控制。下面较深入地讨论字符串的格式化控制，对于理解 print()的输出格式控制会有深刻的体会。

Python 的格式化表达式主要进行格式化操作，由字符串格式化操作符（%）连接两个表达式组成，格式如下。

```
格式化字符串 % 被格式化对象
```

（1）格式化字符串由格式化字段和一些可缺省字符组成。格式字段（也称为格式指令），用于指示被格式化对象在字符流中的格式，其一般结构如下。

```
%[flag][width][.precision]typecode
```

① flag：可以为+（右对齐）、-（左对齐）、0（0 填充）、"（空格）。

② width：宽度。

③ precision：小数点后的精度（仅对浮点类型有用）。

④ typecode：格式化对象类型。

（2）被格式化对象可以是一个值（如一个字符串、一个数值等），也可以是由多个值组成的元组。

（3）格式化字符串中的格式化字段数目，与被格式化对象要一致，并且在类型上相对应。

（4）格式化字符串中可以包括其他字符，这些字符不参与格式化操作，只原样返回。

（5）格式化表达式执行时，进行两个操作：一是将被格式化的对象按照格式化字段指定的格式转换为字符串；二是将用这些转换得到的字符串替换格式化字符串中对应的格式字段。

代码 3-29 格式化表达式应用示例。

```
>>> name = 'Zhang'
>>> "Hello,this is%+10.5s, and you?"% name          #对字符串格式化
'Hello,this is    Zhang, and you?'
>>> 'I\'m %-05.3d years old. How about you? '%20    #对整数格式化
"I'm 020  years old. How about you? "
>>> 'The book is priced at %08.3f yuan. '%23.45     #对浮点数格式化
'The book is priced at 0023.450 yuan. '
```

显然，print()的作用仅仅是将被格式化字符串插入到流向标准输出设备的字符流中。

2．format()方法

format()方法是从 Python 2.6 开始新增的一个格式化字符串函数。它有两种不同的使用形式：一种是由格式化模板字符串调用，用被格式化的对象（字符串和数字）作参数。另一种是在程序中直接调用，有两个参数：一个是被格式化对象；另一个是格式化模板字段。这里介绍第一种方式。

在用模板字符串调用 format()时，模板字符串中，含有一个或多个可替换的模板字段。模板字段用大括号括起，其作用是给某种类型的对象提供一个转换为字符串的模板。format()方法可以有一个或多个类型不同的对象参数。format()方法执行时，首先进行对象参数与模板字段项的匹配，然后将每个对象参数按照所匹配的模板字段指示格式转换为字符串，并替换所匹配的模板，返回一个被替换后的字符串。

1）format()的匹配方式

与字符串格式化表达式相比，format()方法的优势主要在于其对象参数与模板的匹配方式非常灵活。可以通过位置、序号、名称、索引（下标）进行匹配。

代码 3-30 format()的匹配方式应用示例。

```
>>> #位置匹配
>>> '{} is {}, {} is {}.'.format('This','Zhang','he','Wang')
'This is Zhang, he is Wang.'
>>> #序号匹配
>>> '{2} is {3}, {0} is {1}.'.format('he','Wang','This','Zhang')
'This is Zhang, he is Wang.'
>>> #名字匹配
>>> '{pronoun1} is {name1}, {pronoun2} is {name2}.'
    .format(name2 ='Wang',pronoun1 ='This',pronoun2 ='he',name1='Zhang') 'This is Zhang,
he is Wang.'
>>> #下标匹配
>>> pronoun=('he','This');name =('Wang','Zhang')
>>> '{1[1]} is {0[1]}, {1[0]} is {0[0]}.'.format(name,pronoun)
'This is Zhang, he is Wang.'
```

2）format()格式规约

格式规约用于对格式进行精细控制，并采用冒号（:）后面的格式限定符控制。这些格式限定符主要有如下几类。

（1）对齐、填充、宽度。出现模板字段的前面部分，对所有对象都适用，主要包括3种。

① 对齐，包括<（左对齐）、^（居中）和>（右对齐）。

② 填充，用一个字符表示，默认为空格。

③ 宽度一般是最小宽度。如果需要最大宽度，就在最小宽度后加一个圆点（.），后跟一个整数。

这三者的排列顺序是填充、对齐、最小宽度。

代码 3-31 格式化字符串中的对齐与填充示例。

```
>>> ls = 'left aligned'; cs = 'centered'; rs = 'right aligned'
>>> '{:<30}'.format(ls)
'left aligned                  '
>>> '{:>30}'.format(rs)
'                 right aligned'
>>> '{:^30}'.format(cs)
'         centered           '
>>> '{:=^30}'.format(cs)
'===========centered==========='
>>> '{:>>30}'.format(rs)
'>>>>>>>>>>>>>>>>right aligned'
>>> '{:<<30}'.format(ls)
'left aligned<<<<<<<<<<<<<<<<<<'
```

（2）对数值数据，增加如下限定符。

① =，用于填充0与宽度之间的分隔。

② 可选的符号字符：+（必须带符号的数值）、-（仅用于负数）、空格（让正数前空一格、负数带字符-）。

代码 3-32 数值填充与符号指定符应用示例。

```
>>> m = 12345678
>>> '{:=20}'.format(m)
'            12345678'
>>> '{:0=20}'.format(m)
'00000000000012345678'
>>> '{:0=20}'.format(-m)
'-00000000000012345678'
>>> '{:#^20}'.format(m)
'######12345678######'
>>> '{:%>20}'.format(m)
'%%%%%%%%%%%%12345678'
```

（3）仅用于整数的进制指定符：d（十进制）、x 与#x（小写十六进制）、X 与#X（大写十六进制）、o 与#o（八进制）、b 与#b（二进制）。其中，在#引导下可以获取前缀。

代码 3-33 进制指定符应用示例。

```
>>> "int:{0:d}; hex:{0:x}; oct:{0:o}; bin:{0:b}".format(56)       #不获取前缀
'int:56; hex:38; oct:70; bin:111000'
>>> "int:{0:d}; hex:{0:#x}; oct:{0:#o}; bin:{0:#b}".format(56)    #获取前缀
'int:56; hex:0x38; oct:0o70; bin:0b111000'
>>> "hex: {0:x}(x); {0:#x}(#x); {0:X}(X); {0:#X}(#X)".format(56)  #十六进制前缀大小写
'hex: 38(x); 0x38(#x); 38(X); 0X38(#X)'
```

（4）仅用于浮点数的格式限定符有如下两项，它们要一起使用。

① 小数点后的精度：在最小宽度后面加一个圆点（.），后跟一个整数。

② 类型字符：e 或 E（科学记数法表示）、f（标准浮点形式）、g（浮点通用格式）、%（百分数格式）。这类符号位于最后。

代码 3-34 浮点数格式指定符应用示例。

```
>>> x = 0.123456
>>> '{0:15.3e},{0:15.3f},{0:15.3%}'.format(x)
'     1.235e-01,          0.123,       12.346%'
>>> '{0:*<15.3e},{0:#^15.3f},{0:*>15.3%}'.format(x)
'1.235e-01******,#####0.123#####,********12.346%'
```

3.1.7 正则表达式

正则表达式（regular expression，简写为 regex、regexp、RE，复数为 regexps、regexes、regexen、REs，又称为正规表示法、常规表示法）最早由神经生理家 Warren McCulloch 和 Walter Pitts 提出，作为描述神经网络模型的数学符号系统。1956 年，Stephen Kleene 在其论文《神经网事件的表示法》中将其命名为正则表达式。后来被 UNIX 之父 Ken Thompson 把这一成果应用于计算机领域。现在，在很多文本编辑器中，正则表达式被用来检索、替换那些符合某个模式的文本。

1. 模式与匹配

模式（pattern）是关于规则、规律的表达或命名，是一个与问题（problem）、解决方案（solution）和效果（consequences）相关的概念。在文本处理时，会遇到许多问题，例如：

在一段文本中，是否含有数字？

在一段文本中，是否含有手机号码？

在一段文本中，是否含有 E-mail 地址？

⋮

对于这些问题，需要制定一些规则。例如，如何判断什么是手机号码等。正则表达式就是一套用于制定在文本处理时进行模式描述的小型语言。

在一段文本中，查找满足模式的字符串的过程，就称为匹配（matching）。通常，匹配成功就返回一个 match 对象。