第3章 Python 可迭代对象

在 Python 中，实现了特殊方法 __iter__() 的类（面向对象程序设计的内容见第 6 章）的对象称作可迭代对象，同时实现了特殊方法 __iter__() 和 __next__() 的类的对象称作迭代器对象。**迭代器对象具有惰性求值特点，只在必要时才会生成下一个值，每个值只会生成一次。**迭代器对象不持有任何元素，不支持使用下标直接访问指定位置上的元素，也不支持切片运算和内置函数 len()，只能从前往后逐个生成值。例如第 2 章学习的 map 对象、zip 对象、enumerate 对象、filter 对象以及本章 3.2.3 节的生成器对象，都属于迭代器对象。可迭代对象包括迭代器对象以及列表、元组、字典、集合、字符串和 range 对象。其中，列表、元组、字典、集合、字符串这几种类型的对象在任何时刻都在相应的内存中包含了全部的元素，称作容器类对象。range 对象比较特殊，不在内存中包含具体的元素但支持下标和切片运算，没有实现特殊方法 __next__()，所以又不属于迭代器对象。严格来说，Python 官方文档认为列表是可变序列，字符串和元组是不可变序列，字典是映射类型，另外几种对象也具有很多同样或类似的操作。本章重点介绍前面几种容器对象，字符串相关内容见第 7 章和第 8 章，迭代器的内容见第 2 章相关函数以及 3.2.3 节和 5.5 节。

从是否有序的角度来看，列表、元组、字符串属于有序容器对象，支持使用表示序号和位置的整数作为下标来直接访问任意位置上的元素，也支持切片运算来访问其中的部分元素；字典和集合属于无序容器对象，使用这两种对象时不用关注元素的访问顺序，不支持切片运算，虽然字典支持使用“键”作为下标来访问相应元素的“值”，但不能说字典中第几个元素是什么。从是否可变的角度来看，列表、字典、集合属于可变容器对象，支持元素的增加、修改、删除操作；元组和字符串属于不可变容器对象，定义后其中**元素的数量和引用不能通过任意方式进行修改**。两种角度的分类如图 3-1 所示。

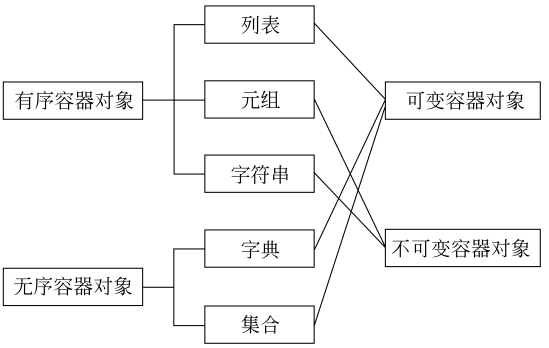

图 3-1 Python 容器对象分类示意图

3.1 列 表

列表（list）是最重要的 Python 内置对象之一，是包含若干元素的有序连续内存空间。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，从而保证相邻元素之间没有缝隙**。这个内存自动管理功能可以大幅度减少程序员的负担，但插入和删除非尾部元素时涉及大量元素的移动，会严重影响效率。另外，在非尾部位置插入和删除元素时会改变该位置后面的元素在列表中的索引，这对于某些操作可能会导致意外的错误结果。所以，除非确实有必要，否则应尽量从列表尾部进行元素的追加与删除操作。

在形式上，列表的所有元素放在一对方括号 [] 中，相邻元素之间使用逗号分隔。同一个列表中元素的数据类型可以各不相同，可以同时包含整数、实数、字符串等基本类型的元素，也可以包含列表、元组、字典、集合、函数以及其他任意对象。如果只有一对方括号而没有任何元素则表示空列表。下面几个都是合法的列表对象：

```
[10, 20, 30, 40]
['富强','民主','文明']
['和谐', 2.0, 5, [10, 20]]
[['自由', 200, 7], ['平等', 260, 9]]
[{3}, {5: 6}, (1, 2, 3)]
```

Python 采用基于值的自动内存管理模式，变量并不直接存储值，而是存储值的引用或内存地址，这也是 Python 中变量可以随时改变类型的重要原因。同理，**Python 列表中的元素也是值的引用**，所以列表中各元素可以是不同类型的数据。

需要注意的是，列表的功能虽然非常强大，但是负担也比较重，开销较大，在实际开发中，最好根据实际的问题选择一种合适的数据类型，要**尽量避免过多使用列表**。

3.1.1　列表创建与删除

使用“=”直接将一个列表赋值给变量即可创建列表对象。

```
>>> a_list = ['a', 'b', 'mpilgrim', 'z', 'example']
>>> a_list = []                                    # 创建空列表
```

也可以使用 list() 函数把元组、range 对象、字符串、字典、集合或其他有限长度的可迭代对象转换为列表。需要注意的是，把字典转换为列表时默认是将字典的“键”转换为列表，而不是把字典的元素转换为列表，如果想把字典的元素转换为列表，需要使用字典对象的 items() 方法明确说明，或者使用 values() 来明确说明要把字典的“值”转换为列表，见 3.3 节。

```
>>> list((3, 5, 7, 9, 11))              # 将元组转换为列表
[3, 5, 7, 9, 11]
>>> list(range(1, 10, 2))               # 将 range 对象转换为列表
[1, 3, 5, 7, 9]
>>> list('hello world')                 # 将字符串转换为列表
['h', 'e', 'l', 'l', 'o', '', 'w', 'o', 'r', 'l', 'd']
>>> list({3, 7, 5})                     # 将集合转换为列表，不用在意元素顺序
[3, 5, 7]
>>> list({'a': 3, 'b': 9, 'c': 78})     # 将字典的“键”转换为列表
['a', 'b', 'c']
>>> list({'a': 3, 'b': 9, 'c': 78}.items())  # 将字典的“键:值”元素转换为列表
[('a', 3), ('b', 9), ('c',78)]
>>> x = list()                          # 创建空列表
```

当一个变量不再使用时，可以使用 del 命令将其删除，这一点适用于所有类型的变量。

```
>>> x = [1, 2, 3]
>>> del x                               # 删除变量，解除与列表的关联
>>> x                                   # 变量删除后无法再访问，抛出异常
NameError:name 'x' is not defined
```

3.1.2　列表元素访问

创建列表之后，可以**使用整数作为下标**来访问其中的元素，其中下标为 0 的元素表示第 1 个元素，下标为 1 的元素表示第 2 个元素，下标为 2 的元素表示第 3 个元素，以此类推；列表还支持使用负整数作为下标，其中下标为 -1 的元素表示最后一个元素，下标为 -2 的元素表示倒数第 2 个元素，下标为 -3 的元素表示倒数第 3 个元素，以此类推，如图 3-2 所示（以列表 ['P', 'y', 't', 'h', 'o', 'n'] 为例）。

```
>>> x = list('Python')                    # 创建列表对象
>>> x
['P', 'y', 't', 'h', 'o', 'n']
>>> x[0]                                  # 下标为 0 的元素，第一个元素
'P'
>>> x[-1]                                 # 下标为 -1 的元素，最后一个元素
'n'
```

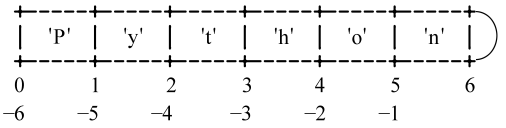

图 3-2　双向索引示意图

3.1.3 节

3.1.3　列表常用方法

列表、元组、字典、集合、字符串有很多操作是通用的，不同类型的容器对象又有一些特有的方法或者支持某些特有的运算符和内置函数。列表对象常用的方法如表 3-1 所示，表中没有列出前后各有两个下画线的特殊方法，可参考 6.4 节。

表3-1　列表对象常用的方法

方　法	功 能 描 述
append(object, /)	将任意类型的对象 object 追加至当前列表的尾部，不影响当前列表中已有的元素下标，也不影响当前列表的引用，没有返回值（或者说返回空值 None，见第 5 章）
clear()	删除当前列表中所有元素，不影响列表的引用，没有返回值
copy()	返回当前列表的浅复制，把当前列表中所有元素的引用复制到新列表中
count(value, /)	返回值为 value 的元素在当前列表中的出现次数，如果当前列表中没有值为 value 的元素则返回 0，对当前列表没有任何影响
extend(iterable, /)	将可迭代对象 iterable 中所有元素追加至当前列表的尾部，不影响当前列表中已有元素的位置和列表的引用，没有返回值
insert(index, object, /)	在当前列表的 index 位置前面插入对象 object，该位置及后面所有元素自动向后移动，索引加 1，没有返回值
index(value, start=0, stop=9223372036854775807, /)	返回当前列表指定范围中第一个值为 value 的元素的索引，若不存在值为 value 的元素则抛出异常 ValueError，可以使用参数 start 和 stop 指定要搜索的下标范围，start 默认为 0 表示从头开始，stop 默认值为最大允许的下标值（默认值由 sys.maxsize 定义，在 64 位操作系统中对应 8 字节能表示的最大整数）

续表

方　法	功 能 描 述
pop(index=-1, /)	删除并返回当前列表中下标为 index 的元素，该位置后面的所有元素自动向前移动，索引减 1。index 默认为 -1，表示删除并返回列表中最后一个元素。列表为空或者参数 index 指定的位置不存在时会引发异常 IndexError，不影响当前列表的引用
remove(value, /)	在当前列表中删除第一个值为 value 的元素，被删除元素所在位置之后的所有元素自动向前移动，索引减 1，不影响当前列表的引用，没有返回值；如果列表中不存在值为 value 的元素则抛出异常 ValueError
reverse()	对当前列表中的所有元素进行原地翻转，首尾交换，不影响当前列表的引用，没有返回值
sort(*, key=None, reverse=False)	对当前列表中的元素进行原地排序，是稳定排序（在指定规则下相等的元素保持原来的相对顺序）。参数 key 用来指定排序规则，可以为任意可调用对象；参数 reverse 为 False 表示升序，为 True 表示降序。不影响当前列表的引用，没有返回值

1. append()、insert()、extend()

这 3 个方法用于向列表对象中添加元素，其中 append() 用于向列表尾部追加一个元素，insert() 用于向列表任意指定位置插入一个元素，extend() 用于将另一个可迭代对象中的所有元素追加至当前列表的尾部。这 3 个方法都属于**原地操作，不影响列表对象在内存中的起始地址**。对于长列表而言，使用 insert() 方法在列表首部或中间位置插入元素时效率较低。如果确实需要在首部按序插入多个元素，可以先在尾部追加，然后使用 reverse() 方法进行翻转，或者考虑使用标准库 collections 中的双端队列 deque 对象提供的 appendleft() 方法。

```
>>> x = [1, 2, 3]
>>> x.append(4)                  # 在尾部追加元素
>>> x.insert(0, 0)               # 在指定位置插入元素
>>> x.extend([5, 6, 7])          # 在尾部追加多个元素
>>> x
[0, 1, 2, 3, 4, 5, 6, 7]
```

2. pop()、remove()、clear()

这 3 个方法用于删除列表中的元素，其中 pop() 用于**删除并返回**指定位置（默认是最后一个）上的元素，如果指定的位置不是合法的索引则抛出异常，对空列表调用 pop() 方法也会抛出异常；remove() 用于删除列表中**第一个**值与指定值相等的元素，如果列表中不存在该元素则抛出异常；clear() 用于清空列表中的所有元素。这 3 个方法也属于**原地操作**，不影响列表对象的内存地址。另外，还可以使用 del 命令删除列表中指定位置的元素，这个方法同样也属于原地操作。

```
>>> x = [1, 2, 3, 4, 5, 6, 7]
>>> x.pop()                          # 弹出并返回尾部元素
7
>>> x.pop(0)                         # 弹出并返回指定位置的元素
1
>>> x.clear()                        # 删除所有元素
>>> x
[]
>>> x = [1, 2, 1, 1, 2]
>>> x.remove(2)                      # 删除首个值为 2 的元素
>>> del x[3]                         # 删除指定位置上的元素
>>> x
[1, 1, 1]
```

列表具有内存自动收缩和扩张功能，在列表中间位置插入或删除元素时，不仅效率较低，而且该位置后面所有元素在列表中的索引也会发生变化，必须牢牢记住这一点。

3. count()、index()

列表方法 count() 用于返回列表中指定元素出现的次数；index() 用于返回指定元素在列表中**首次出现**的位置，如果该元素不在列表中则抛出异常。

```
>>> x = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
>>> x.count(3)                       # 元素 3 在列表 x 中的出现次数
3
>>> x.count(5)                       # 不存在，返回 0
0
>>> x.index(2)                       # 元素 2 在列表 x 中首次出现的索引
1
>>> x.index(5)                       # 列表 x 中没有 5，抛出异常
ValueError:5 is not in list
```

通过前面的介绍我们已经知道，列表对象的很多方法在特殊情况下会抛出异常，一旦出现异常，整个程序就会崩溃，这是我们不希望的。为避免引发异常导致程序崩溃，一般来说有两种方法：①使用选择结构确保列表中存在指定元素再调用有关的方法，见第 4 章；②使用异常处理结构，见第 11 章。下面的代码使用异常处理结构保证用户输入的是三位数，然后使用关键字 in 来测试用户输入的数字是否在列表中，如果存在则输出其索引，否则提示不存在。

```
from random import sample

lst = sample(range(100,1000), 100)      # 100 个不重复的随机数
while True:                             # 循环结构，见第 4 章
    x = input('请输入一个三位数：')
    try:                                # 异常处理结构，见第 11 章
        assert len(x)==3, '长度必须为 3'
        x = int(x)
        break                           # 结束循环结构
    except:                             # 如果 try 块中的代码出错，执行 except
```

```
            pass                    # 空语句，什么也不做
if x in lst:                        # 选择结构，见第 4 章
    print(' 元素 {0} 在列表中的索引为 :{1}'.format(x, lst.index(x)))
else:
    print('列表中不存在该元素.')
```

4. sort()、reverse()

列表对象的 sort() 方法用于按照指定的规则对所有元素进行排序，默认所有元素按大小升序排序；reverse() 方法用于将列表中的所有元素逆序或翻转，也就是第一个元素和倒数第一个元素交换位置，第二个元素和倒数第二个元素交换位置，以此类推。

```
>>> x = list(range(11))                 # 包含 11 个整数的列表
>>> import random
>>> random.shuffle(x)                   # 把列表 x 中的元素随机乱序
>>> x
[6, 0, 1, 7, 4, 3, 2, 8, 5, 10, 9]
>>> x.sort(key=lambda item:len(str(item)), reverse=True)
                                        # 按转换成字符串以后的长度降序排列
>>> x                                   # 长度一样的保持原来的相对顺序
[10, 6, 0, 1, 7, 4, 3, 2, 8, 5, 9]
>>> x.sort(key=str)                     # 按转换为字符串后的大小升序排序
>>> x
[0, 1, 10, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x.sort()                            # 按元素大小升序排序
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> x.reverse()                         # 把所有元素翻转或逆序
>>> x
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

列表对象的 sort() 和 reverse() 分别对列表进行原地排序 (in-place sorting) 和逆序，**没有返回值**。所谓“原地”，就是用处理后的数据替换原来的数据，列表首地址不变，列表中元素原来的顺序全部丢失。如果不想丢失原来的顺序，可以使用 2.4.4 节介绍的内置函数 sorted() 和 reversed()。

5. copy()(选讲)

列表对象的 copy() 方法返回列表的浅复制。**所谓浅复制，是指生成一个新的列表，把原列表中所有元素的引用都复制到新列表中。**如果原列表中只包含整数、实数、复数等基本类型或元组、字符串这样的不可变类型的数据，是没有问题的。但是，如果原列表中包含列表之类的可变数据类型，由于浅复制时只是把子列表的引用复制到新列表中，于是修改任何一个都可能会影响另外一个。

```
>>> x = [1, 2, [3, 4]]                  # 原列表中包含子列表
>>> y = x.copy()                        # 浅复制
>>> y                                   # 两个列表中的内容完全一样
```

```
[1, 2, [3, 4]]
>>> y[2].append(5)                    # 为子列表追加元素，影响 x
>>> x[0] = 6                          # 整数、实数等不可变类型不受此影响
>>> y.append(6)                       # 在新列表尾部追加元素，不是修改子列表
>>> y
[1, 2, [3, 4, 5], 6]
>>> x                                 # 原列表不受影响
[6, 2, [3, 4, 5]]
```

列表对象的 copy() 方法和切片操作（见 3.1.7 节）以及标准库 copy 中的 copy() 函数都是返回浅复制，如果想避免上面代码演示的问题，可以使用标准库 copy 中的 deepcopy() 函数实现深复制。**所谓深复制，是指对原列表中的元素进行递归，把所有的值都复制到新列表中，对嵌套的子列表不再是直接复制引用。**这样一来，新列表和原列表是互相独立的，修改任何一个都不会影响另外一个。

```
>>> import copy
>>> x = [1, 2, [3, 4]]
>>> y = copy.deepcopy(x)              # 深复制，x 和 y 完全独立，互不影响
>>> x[2].append(5)                    # 为原列表中的子列表追加元素
>>> y.append(6)                       # 在新列表尾部追加元素
>>> y
[1, 2, [3, 4], 6]
>>> x
[1, 2, [3, 4, 5]]
```

下面的代码把同一个列表字面值赋值给两个不同的变量，这两个变量是互相独立的，是两个不同的列表，修改任何一个列表都不会影响另外一个列表。

```
>>> x = [1, 2, [3, 4]]
>>> y = [1, 2, [3, 4]]          # x 和 y 是两个不同的列表，只是包含同样的值
>>> x.append(5)
>>> x[2].append(6)              # 修改其中一个列表的子列表
>>> x
[1, 2, [3, 4, 6], 5]
>>> y                           # 不影响另外一个列表
[1, 2, [3, 4]]
```

下面的代码演示的是另外一种情况，**把一个列表变量赋值给另外一个变量，这样两个变量引用同一个列表对象，对其中一个做的任何修改都会立刻在另外一个变量得到体现。**

```
>>> x = [1, 2, [3, 4]]
>>> y = x                             # 两个变量引用同一个列表
>>> x[2].append(5)
>>> x.append(6)
>>> x[0] = 7
>>> x
```

```
[7, 2, [3, 4, 5], 6]
>>> y                                    # 对 x 做的任何修改，y 都会受到影响
[7, 2, [3, 4, 5], 6]
```

3.1.4　列表对象支持的运算符

3.1.4 节

加法运算符 + 也可以实现列表增加元素的目的，但这个运算符不属于原地操作，而是返回新列表，并且涉及大量元素的复制，效率非常低。使用复合赋值运算符 += 实现列表追加元素时属于原地操作，与 extend() 方法等价。

```
>>> x = [1, 2, 3]
>>> id(x)                               # 返回内存地址，不同会话中可能不一样
53868168
>>> x = x + [4]                         # 连接两个列表
>>> x
[1, 2, 3, 4]
>>> id(x)                               # 内存地址发生改变
53875720
>>> x += [5]                            # 为列表追加元素
>>> x
[1, 2, 3, 4, 5]
>>> id(x)                               # 内存地址不变
53875720
```

乘法运算符 * 可以用于列表和整数相乘，表示序列重复，返回新列表，从一定程度上来说也可以实现为列表增加元素的功能。与加法运算符 + 一样，该运算符也适用于元组和字符串。另外，运算符 *= 也可以用于列表元素重复，与运算符 += 一样属于原地操作。

```
>>> x = [1, 2, 3, 4]
>>> x = x * 2                           # 元素重复，返回新列表
>>> x
[1, 2, 3, 4, 1, 2, 3, 4]
>>> x *= 2                              # 元素重复，原地进行，x 的引用不变
>>> x
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
>>> [1, 2, 3] * 0                       # 重复 0 次，清空
[]
```

成员测试运算符 in 可用于测试列表中是否包含某个元素，查询时间随着列表长度的增加而线性增加，同样的操作对于集合则是常数级的，与集合大小关系不大。

```
>>> 3 in [1, 2, 3]
True
>>> 3 in [1, 2, '3']
False
```

3.1.5 内置函数对列表的操作

除了列表对象自身方法之外，很多 Python 内置函数也可以对列表进行操作。例如，max()、min() 函数用于返回列表中所有元素的最大值和最小值，sum() 函数用于返回列表中所有元素之和，len() 函数用于返回列表中元素的个数，zip() 函数用于将多个列表中对应位置的元素重新组合为元组并返回“包含”这些元组的 zip 对象，enumerate() 函数返回包含若干下标和值的迭代器对象，map() 函数把函数映射到列表上的每个元素，filter() 函数根据指定函数的返回值对列表元素进行过滤，all() 函数用来测试列表中是否所有元素都等价于 True，any() 函数用来测试列表中是否有等价于 True 的元素，见 2.4 节。

```
>>> x = list(range(11))                    # 生成列表
>>> import random
>>> random.shuffle(x)                      # 打乱列表中元素的顺序
>>> x
[0, 6, 10, 9, 8, 7, 4, 5, 2, 1, 3]
>>> all(x)                                 # 测试是否所有元素都等价于 True
False
>>> any(x)                                 # 测试是否存在等价于 True 的元素
True
>>> max(x)                                 # 返回最大值
10
>>> max(x, key=str)                        # 转换为字符串之后的最大值
9
>>> min(x)
0
>>> sum(x)                                 # 所有元素之和
55
>>> len(x)                                 # 列表元素个数
11
>>> list(zip(['a', 'b', 'c'], [1, 2]))    # 如果两个列表不等长，则以短的为准
[('a', 1), ('b', 2)]
>>> list(enumerate(x))                     #  enumerate 对象可以转换为列表、元组、
                                           # 集合
[(0, 0), (1, 6), (2, 10), (3, 9), (4, 8), (5, 7), (6, 4), (7, 5), (8, 2),
(9, 1), (10, 3)]
```

3.1.6 节

3.1.6 列表推导式语法与应用

列表推导式 (list comprehension) 也称为列表解析式，可以使用非常简洁的方式对列表或其他可迭代对象的元素进行遍历、过滤或再次计算，生成满足特定需求的新列表，代码非常简洁，具有很强的可读性，是 Python 程序开发时应用较多的技术之一。Python 的内部实现对列表推导式做了大量优化，可以保证很快的运行速度，也是推荐使用的一种技术。列表推导式的语法形式为

```
[expression for expr1 in sequence1 if condition1
           for expr2 in sequence2 if condition2
           for expr3 in sequence3 if condition3
           ⋮
           for exprN in sequenceN if conditionN]
```

列表推导式在逻辑上等价于一个循环语句（见第4章），只是形式上更加简洁。例如：

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

相当于

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

再如：

```
>>> freshfruit = ['banana', 'loganberry', 'passion fruit']
>>> aList = [w.strip() for w in freshfruit] #字符串方法 strip() 删除两端空格
```

等价于下面的代码：

```
>>> aList = []
>>> for item in freshfruit:
    aList.append(item.strip())
```

也等价于下面两种函数式编程模式的代码：

```
>>> aList = list(map(lambda x: x.strip(), freshfruit))
```

或

```
>>> aList = list(map(str.strip, freshfruit))
```

有这样一个故事，阿凡提（也有的版本中是阿基米德）与国王比赛下棋，国王说要是自己输了，阿凡提想要什么他都可以拿得出来。阿凡提说那就要点米吧，棋盘一共64个小格子，在第一个格子里放1粒米，第二个格子里放2粒米，第三个格子里放4粒米，第四个格子里放8粒米。以此类推，后面每个格子里的米都是前一个格子里的2倍，一直把64个格子都放满。那么到底需要多少粒米呢？使用列表推导式再结合内置函数 sum() 就很容易知道答案。

```
>>> sum([2**i for i in range(64)])
18446744073709551615
```

按每千克大米约 52 000 粒计算，为放满棋盘，需要大概 3500 亿吨大米。结果可想而知，最后国王没有办法拿出那么多米。

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．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]
```

在这个列表推导式中有两个循环，其中第一个循环可以看作外循环，执行得慢；第二个循环可以看作内循环，执行得快。上面代码的执行过程等价于下面的写法：

```
>>> vec = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> result = []
>>> for elem in vec:
    for num in elem:
       result.append(num)        # 这里需要按两次 Enter 键来执行代码
>>> result
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> from itertools import chain
>>> list(chain(*vec))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

这里演示的只是一层嵌套列表的平铺，如果有多级嵌套或者不同子列表嵌套深度不同，就不能使用上面的方法了。这时，可以使用函数递归（见第 5 章）实现。

```
def flatList(lst):                      # 函数定义，见第 5 章
    result = []                         # 存放最终结果的列表
    def nested(lst):                    # 函数嵌套定义，见第 5 章
        for item in lst:
            if isinstance(item, list):
                nested(item)            # 递归处理子列表
            else:
                result.append(item)     # 扁平化列表
    nested(lst)                         # 调用嵌套定义的函数
    return result                       # 返回结果
```

2．过滤不符合条件的元素

在列表推导式中可以使用 if 子句对列表中的元素进行筛选，只在结果列表中保留符合条件的元素。下面的代码用于从列表中选择符合条件的元素组成新的列表：

```
>>> aList = [-1, -4, 6, 7.5, -2.3, 9, -11]
>>> [i for i in aList if i>0]          # 所有大于 0 的数字

[6, 7.5, 9]
```

再如，已知有一个包含一些同学成绩的字典（见3.3节），现在需要计算所有成绩的最高分、最低分、平均分，并查找所有最高分同学，代码可以这样编写：

```
>>> scores = {'Zhang San':45, 'Li Si':78, 'Wang Wu':40, 'Zhou Liu':96,
              'Zhao Qi':65, 'Sun Ba':90, 'Zheng Jiu':78, 'Wu Shi':99,
              'Dong Shiyi':60}
>>> highest = max(scores.values())                    #最高分
>>> lowest = min(scores.values())                     #最低分
>>> average = sum(scores.values()) / len(scores)      #平均分
>>> highest, lowest, average
(99,40,72.33333333333333)
>>> highestPerson = [name for name, score in scores.items() if score==highest]
>>> highestPerson
['Wu Shi']
```

下面的代码使用列表推导式查找列表中最大元素的所有位置：

```
>>> from random import randint
>>> x = [randint(1, 10) for i in range(20)]
                                # 20个介于[1, 10]区间的整数
>>> x
[10, 2, 3, 4, 5, 10, 10, 9, 2, 4, 10, 8, 2, 2, 9, 7, 6, 2, 5, 6]
>>> m = max(x)
>>> [index for index, value in enumerate(x) if value==m]
                                #最大整数的所有出现位置
[0, 5, 6, 10]
```

3．同时遍历多个列表或可迭代对象

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

上面第一个列表推导式等价于

```
>>> result = []
>>> for x in [1, 2, 3]:
    for y in [3, 1, 4]:
        if x!= y:
           result.append((x, y))              #这里要按两次回车键
>>> result
[(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转置

```
>>> 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]]
```

上面列表推导式的执行过程等价于下面的代码，循环结构见 4.3 节。

```
>>> matrix = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
>>> result = []
>>> for i in range(len(matrix[0])):
    temp = []
    for row in matrix:
        temp.append(row[i])
    result.append(temp)
>>> result
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

也可以使用内置函数 zip() 和 list() 来实现矩阵转置：

```
>>> list(map(list, zip(*matrix)))
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

5．列表推导式中可以使用函数或复杂表达式

```
>>> def f(v):
    if v%2 == 0:
        v = v ** 2
    else:
        v = v + 1
    return v                                    # 这里要按两次回车键
>>> print([f(v) for v in [2, 3, 4, -1] if v>0])
[4, 4, 16]
>>> print([v**2 if v%2==0 else v+1 for v in [2, 3, 4, -1] if v>0])
[4, 4, 16]
```

3.1.7 节

3.1.7 切片操作

切片除了适用于列表之外，还适用于元组、字符串、range 对象，但列表的切片操作具有最强大的功能。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部分元素，还可以通过切片操作为列表对象增加元素。

在形式上，切片使用 2 个冒号分隔的 3 个数字来完成。

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

其中，3 个数字的含义与内置函数 range(start，stop，step) 完全一致，第一个数字 start 表示切片开始的位置，默认为 0；第二个数字 stop 表示切片截止（但不包含）的位置（默认为列表长度）；第三个数字 step 表示切片的步长（默认为 1）。当 step 为 1 时可以省略，省略步长时还可以同时省略最后一个冒号。另外，当 step 为负整数时，表示反向切片，这时 start 应该在 stop 的右侧才行，否则返回空列表。

1．使用切片获取列表的部分元素

使用切片可以返回列表中部分元素组成的新列表。与使用索引作为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**，而是简单地截断或者返回一个空列表，代码具有更强的健壮性。

```
>>> aList = [3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> aList[::]              # 返回包含原列表中所有元素的新列表
[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> aList[::-1]            # 返回包含原列表中所有元素逆序排列的列表
[17, 15, 13, 11, 9, 7, 6, 5, 4, 3]
>>> aList[::2]             # 隔一个元素取一个元素，获取偶数位置的元素
[3, 5, 7, 11, 15]
>>> aList[1::2]            # 隔一个元素取一个元素，获取奇数位置的元素
[4, 6, 9, 13, 17]
>>> aList[3:6]             # 下标范围为 [3, 6) 的元素
[6, 7, 9]
>>> aList[0:100]           # 切片结束位置大于列表长度时，在列表尾部截断

[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> aList[100]             # 抛出异常，使用下标时不允许越界访问
IndexError:list index out of range
>>> aList[100:]            # 切片开始位置大于列表长度时，返回空列表
[]
>>> aList[-15:3]           # 在列表开始处进行必要的截断处理
[3, 4, 5]
>>> len(aList)
10
>>> aList[3:-10:-1]        # 位置 3 在位置 -10 的右侧，步长 -1 表示反向切片
[6, 5, 4]
>>> aList[3:-5]            # 位置 3 在位置 -5 的左侧，正向切片
[6, 7]
```

2．使用切片为列表增加元素（选讲）

可以使用切片操作在列表任意位置插入新元素，不影响列表对象的内存地址，属于原地操作。此时，切片只是用来标记原列表中的位置，并没有“切”出来。

```
>>> aList = [3, 5, 7]
>>> aList[len(aList):] = [9]            # 在列表尾部增加元素
>>> aList[:0] = [1, 2]                  # 在列表头部插入多个元素
>>> aList[3:3] = [4]                    # 在列表中间位置插入元素
>>> aList
[1, 2, 3, 4, 5, 7, 9]
```

3．使用切片替换和修改列表中的元素（选讲）

```
>>> aList = [3, 5, 7, 9]
>>> aList[:3] = [1, 2, 3]               # 替换列表元素，等号两边的列表长度相等
```

```
>>> aList
[1, 2, 3, 9]
>>> aList[3:] = [4, 5, 6]        # 步长为 1 时等号两边的列表长度可以不相等
>>> aList
[1, 2, 3, 4, 5, 6]
>>> aList[::2] = [0] * 3         # 隔一个元素修改一个元素，等号两边的长度必须相等
>>> aList
[0, 2, 0, 4, 0, 6]
>>> aList[::2] = ['a', 'b', 'c']# 隔一个元素修改一个元素
>>> aList
['a', 2, 'b', 4, 'c', 6]
>>> aList[1::2] = range(3)       # 序列解包的用法，见 3.5 节
>>> aList
['a', 0, 'b', 1, 'c', 2]
>>> aList[1::2] = map(lambda x: x!=5, range(3))
>>> aList
['a', True, 'b', True, 'c', True]
>>> aList[1::2] = zip('abc', range(3))
                                 # map、filter、zip 对象都支持这样的用法
>>> aList
['a', ('a', 0), 'b', ('b', 1), 'c', ('c', 2)]
>>> aList[::2] = [1]          # 步长不为 1 时等号两边列表的长度必须相等，否则出错
ValueError:attempt to assign sequence of size 1 to extended slice of size 3
```

4．使用切片删除列表中的元素（选讲）

把列表指定范围的切片赋值为空列表，即可删除元素。

```
>>> aList = [3, 5, 7, 9]
>>> aList[:3] = []                      # 删除列表中前 3 个元素
>>> aList
[9]
```

也可以使用 del 命令与切片结合来删除列表中的部分元素，并且步长可以不为 1。

```
>>> aList = [3, 5, 7, 9, 11]
>>> del aList[::2]              # 切片元素不连续，隔一个元素删除一个元素
>>> aList
[5, 9]
```

5．切片得到的是列表的浅复制

在 3.1.3 节介绍列表对象的 copy() 方法时曾经提到，切片返回的是列表元素的浅复制，与列表对象的直接赋值并不一样，和 3.1.3 节介绍的深复制也有本质的不同。

```
>>> aList = [3, 5, 7]           # 列表中元素为可哈希对象
>>> bList = aList[::]           # 切片，浅复制
>>> aList == bList              # 两个列表的值相等
True
```

```
>>> aList is bList                  # 浅复制，不是同一个对象
False
>>> bList[1] = 8                    # 修改 bList 列表元素的值不会影响 aList
>>> bList                           #  bList 的值发生改变
[3, 8, 7]
>>> aList                           #  aList 的值没有发生改变
[3, 5, 7]
>>> x = [[1], [2], [3]]             # 列表中包含列表
>>> y = x[:]                        # 浅复制
>>> y
[[1], [2], [3]]
>>> y[0] = [4]                      # 直接修改 y 中下标为 0 的元素值，不影响 x
>>> y
[[4], [2], [3]]
>>> y[1].append(5)                  # 通过列表对象的方法原地增加元素
>>> y
[[4], [2, 5], [3]]
>>> x                               # 列表 x 也受到同样的影响
[[1], [2, 5], [3]]
```

3.2　元　　组

3.2 节

3.2.1　元组创建与元素访问

列表的功能虽然很强大，但负担也很重，影响了运行效率。有时并不需要那么多功能，很希望能有个轻量级的列表，元组 (tuple) 正是这样一种类型。在形式上，元组的所有元素放在一对圆括号中，元素之间使用逗号分隔，**如果元组中只有一个元素，则必须在最后增加一个逗号。**

```
>>> x = (1, 2, 3)                   # 直接把元组字面值赋值给一个变量
>>> type(x)                         # 使用 type() 函数查看变量的类型
<class 'tuple'>
>>> x[0]                            # 元组支持使用下标访问指定位置的元素
1
>>> x[-1]                           # 最后一个元素，元组支持双向索引
3
>>> x[1] = 4                        # 元组是不可变的，尝试修改时报错
TypeError:'tuple' object does not support item assignment
>>> x = (3)                         # 这和 x = 3 是一样的
>>> x
3
>>> x = (3,)                        # 如果元组中只有一个元素，必须在后面多写一个逗号
>>> x
(3,)
>>> x = ()                          # 空元组
>>> x = tuple()                     # 空元组
>>> tuple(range(5))                 # 将其他有限长度的可迭代对象转换为元组
(0, 1, 2, 3, 4)
```

除了上面的方法可以直接创建元组之外，很多内置函数的返回值也是“包含”了若干元组的可迭代对象，如 enumerate()、zip() 等。

```
>>> list(enumerate(range(5)))
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]
>>> list(zip(range(3), 'abcdefg'))
[(0, 'a'), (1, 'b'), (2, 'c')]
```

3.2.2 元组与列表的异同点

列表和元组都属于有序序列，都支持使用双向索引访问其中的元素，以及使用 count() 方法统计元素的出现次数和 index() 方法获取元素首次出现的索引，len()、map()、filter() 等大量内置函数和 +、*、+=、in 等运算符也都可以作用于列表和元组。虽然列表和元组有着一定的相似之处，但在本质上和内部实现上都有着很大的不同。

元组属于不可变 (immutable) 序列或可哈希 (hashable) 序列，不可以修改元组中元素的引用，也无法为元组增加或删除元素。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，不允许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。从一定程度上讲，可以认为元组是轻量级的列表，或者“常量列表”。

Python 的内部实现对元组做了大量优化，占用内存比列表略少，访问速度比列表略快。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，不需要对其元素进行任何修改，那么一般建议使用元组而不用列表。元组在内部实现上不允许修改其元素的引用，从而使得代码更加安全。例如，调用函数时使用元组传递参数可以防止在函数中修改元组，使用列表则很难保证这一点。

最后，作为不可变序列，与整数、字符串一样，元组可用作字典的键，也可以作为集合的元素。**列表永远都不能当作字典键使用，也不能作为集合中的元素**，因为列表是可变的。内置函数 hash() 可以用来测试一个对象是否可哈希。一般来说，并不需要关心该函数的返回值具体是什么，重点是对象是否可哈希，如果对象不可哈希（或者说可变）会抛出异常。

```
>>> hash((1,))                  # 元组、数字、字符串都是可哈希的
3430019387558
>>> hash(3)
3
>>> hash('hello world.')        # 不用关心具体的值，有返回值就说明可哈希
-4012655148192931880
>>> hash([1, 2])                # 列表不可哈希，报错
TypeError:unhashable type:'list'
```

3.2.3　生成器表达式

生成器表达式（generator expression）语法与列表推导式非常相似，在形式上生成器表达式使用圆括号（parentheses）作为定界符，而不是列表推导式所使用的方括号（square brackets）。**生成器推导式的结果是一个生成器对象**，生成器对象属于迭代器对象，具有惰性求值的特点，只在需要时生成新元素，比列表推导式具有更高的效率，内存占用非常少，尤其适合大数据处理的场合。

使用生成器对象的元素时，可以将其转换为列表或元组，也可以使用生成器对象的 __next__() 方法或者内置函数 next() 进行遍历，或者使用 for 循环来遍历其中的元素。但是不管用哪种形式，只能从前往后逐个访问其中的元素，**没有任何方法可以再次访问已访问过的元素，也不支持使用下标或切片访问其中的元素。**当所有元素访问结束以后，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。enumerate、filter、map、zip 等迭代器对象也具有同样的特点，见 2.4.5 节和 2.4.6 节。最后，包含 yield 语句的函数也可以用来创建生成器对象，详见 5.5 节。

```
>>> g = ((i+2)**2 for i in range(10))   # 创建生成器对象
>>> g
<generator object<genexpr>at 0x0000000003095200>
>>> tuple(g)                            # 将生成器对象转换为元组
(4, 9, 16, 25, 36, 49, 64, 81, 100, 121)
>>> list(g)                             # 生成器对象已遍历结束，没有元素了
[]
>>> g = ((i+2)**2 for i in range(10))   # 重新创建生成器对象
>>> g.__next__()                 # 使用生成器对象的 __next__() 方法获取元素
4
>>> g.__next__()                 # 获取下一个元素，更推荐使用下面的 next() 函数
9
>>> next(g)                             # 使用 next() 函数获取生成器对象中的元素
16
>>> g = ((i+2)**2 for i in range(10))
>>> for item in g:                      # 使用循环遍历生成器对象中的元素
    print(item, end=' ')
4 9 16 25 36 49 64 81 100 121
```

3.3　字　　典

3.3 节

字典（dict）是包含若干“键：值”元素的**无序可变**容器对象，字典中的每个元素包含用英文半角冒号分隔开的“键”和“值”两部分，表示一种映射或对应关系，也称为关联数组。定义字典时，每个元素的“键”和“值”之间用冒号分隔，不同元素之间用英文半角逗号分隔，所有的元素放在一对花括号“{}”中。

字典中元素的“键”可以是 Python 中任意不可变数据，如整数、实数、复数、字符串、元组等类型的可哈希数据，但不能使用列表、集合、字典或其他可变类型作为字典的

“键”。另外，**字典中的“键”不允许重复**，“值”是可以重复的。**字典在内部维护的哈希表使得检索操作非常快。**使用内置字典类型 dict 时不要太在乎元素的先后顺序，如果确实在乎元素顺序可以使用 collections 标准库中的 OrderedDict 类。Python 3.6 和更新中对内置类型 dict 进行了优化，比 Python 3.5 能节约 20%~25% 的内存空间，并且使用二次索引技术使得字典元素看起来有序，但不建议依赖这个顺序。

运算符与内置函数对字典的操作可以参考 2.2 节和 2.4 节，本节重点介绍字典对象本身提供的方法，表 3-2 中列出了完整清单和功能简单描述。

表3-2 字典对象的方法与功能

方 法	功 能 描 述
clear()	不接收参数，删除当前字典对象中的所有元素，没有返回值
copy()	不接收参数，返回当前字典的浅复制
fromkeys(iterable, value=None, /)	以参数 iterable 中的元素为“键”、以参数 value 为“值”创建并返回字典对象。字典中所有元素的“值”的引用都一样，要么是 None，要么全部引用参数 value 指定的值
get(key, default=None,/)	返回当前字典对象中以参数 key 为“键”对应的元素的“值”，如果当前字典对象中没有以 key 为“键”的元素，返回参数 default 的值
items()	不接收参数，返回包含当前字典对象中所有元素的 dict_items 对象，其中每个元素形式为元组 (key，value)，dict_items 对象可以和集合进行并集、交集、差集等运算
keys()	不接收参数，返回当前字典对象中所有的“键”，结果为 dict_keys 类型的可迭代对象，可以和集合进行并集、交集、差集等运算
pop(k[, d])	删除以 k 为“键”的元素，返回对应元素的“值”，如果当前字典中没有以 k 为“键”的元素，返回参数 d，此时如果没有指定参数 d，抛出 KeyError 异常
popitem()	不接收参数，按后进先出 (Last In First Out，LIFO) 顺序删除并返回一个元组 (key，value)，如果当前字典为空则抛出 KeyError 异常
setdefault(key, default=None, /)	如果参数 key 是当前字典的“键”，返回对应的“值”；如果 key 不是当前字典的“键”，插入元素“key：default”并返回 default 的值
update([E,]**F)	使用 E 和 F 中的数据更新当前字典对象，** 表示参数 F 只能接收字典或关键参数 (见 5.2.3 节)，没有返回值
values()	不接收参数，返回包含当前字典对象中所有的“值”的 dict_values 对象

3.3.1 字典创建与删除

使用等号“=”将一个字典字面值赋值给一个变量即可创建一个字典变量。

```
>>> aDict = {'server':'db.diveintopython3.org','database':'mysql'}
>>> data = {'国家': ['富强', '民主', '文明', '和谐'],
            '社会': ['自由', '平等', '公正', '法治'],
            '公民': ['爱国', '敬业', '诚信', '友善']}
```