第 3 章　数据的组织——列表

第 2 章介绍了几种基本的数据类型：整型、浮点型、布尔型等。本章介绍一种非常有用的数据结构：列表。利用列表可以将基本的数据类型有效地组织起来，再借助列表对象提供的方法可以方便地对这些基本数据进行处理。

3.1　列表概述

列表是一种序列类型，通常用来存储一组相同类型的对象。

```
>>> primes = [2, 3, 5, 7]
>>> primes
[2, 3, 5, 7]
```

上述代码中，赋值语句将变量 primes 绑定到一个列表，其中包含 4 个整型对象。如图 3-1 所示，每个对象在列表中具有相对固定的位置。该位置可以用整数表示，也称为索引。第 1 个对象的索引是 0，第 2 个对象的索引是 1，第 n 个对象的索引是 $n-1$。可以通过索引来引用列表中的对象，方法是：数组名后跟方括号，然后方括号中放置该对象的索引。例如 primes[0]引用对象 2，而 primes[3]引用对象 7。列表中的对象也被称为列表的元素。

图 3-1 中的列表 prime 包含 4 个整型对象，这是一种简化的说法。列表中实际存储了这 4 个整型对象的引用。如图 3-2 所示，prime[0]存放的是整型对象 2 的引用，prime[1]存放的是整型对象 3 的引用。虽然对象 2 和对象 3 在列表 prime 中的相对位置是相邻的，但它们在内存中的实际存放位置可能并不相邻。了解列表这种存放引用的工作机制非常重要，

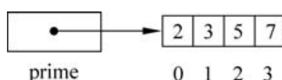

图 3-1　包含 4 个整数的列表

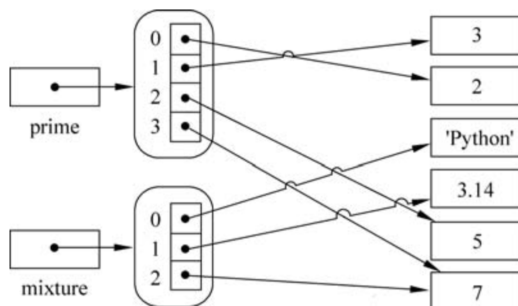

图 3-2　列表的引用机制

后面将看到这种机制会导致极其隐晦的问题。

得益于引用机制，列表也可以用来存储不同类型的对象。图 3-2 中的列表 mixture 就是一个例子。可以看到，变量 mixture 绑定到一个列表上，mixture[0]引用字符串对象 'Python'，mixture[1]引用浮点型对象 3.14，而 mixture[2]引用整型对象 7。注意，primes[3]也引用了整型对象 7。务必记住两点：第一，变量绑定到一个对象；第二，列表存储的是对象的引用。

为了提高可读性，本书会简化表达。例如下列两种表述都是指变量 mixture 绑定到列表['Python', 3.14, 7]。表述一：变量 mixture 的值是列表['Python', 3.14, 7]。表述二：将列表['Python', 3.14, 7]赋值给变量 mixture。同样，下列两种表述都是指 mixture[1]引用浮点型对象 3.14。表述一：mixture[1]的值是 3.14。表述二：将 3.14 赋值给 mixture[1]。此外，本书在对列表进行图示时，通常也采用图 3-1 的简化方式。

3.2 创建列表的方法

创建列表最直接的方法是在方括号中放置以逗号分隔的字面量。如果方括号中为空，那么创建的就是一个空列表。

```
>>> empty_list = [ ]
>>> primes = [2, 3, 5, 7]
>>> mixture = ['Python', 3.14, 7]
>>> matrix = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
>>> matrix[0]
[1, 0, 0]
>>> matrix[0][1]
0
```

上述代码创建了 4 个列表，并绑定到相应的变量上。注意，empty_list 是一个空列表，不包含任何对象。列表 matrix 包含 3 个对象，每个对象又分别是列表，例如 matrix[0]是[1, 0, 0]。因为 matrix[0]是一个列表，所以可以使用索引来引用其中的对象，例如 matrix[0][1]就是 matrix[0]中索引为 1 的对象，即整型对象 0。这种包含列表的列表也称为嵌套列表。

创建列表的第二种方法是使用函数 list()。一方面，可以在调用函数时不提供任何参数，这时创建一个空列表。另一方面，也可以提供一个可迭代对象，例如 range 对象、字符串，甚至是一个列表，这时创建的列表中包含可迭代对象中的所有元素，并且这些元素的相对顺序也保持一致。

```
>>> empty_list = list()
>>> empty_list
[]
>>> a = list(range(5))
>>> a
```

```
[0, 1, 2, 3, 4]
>>> b = list('Python')
>>> b
['P', 'y', 't', 'h', 'o', 'n']
>>> c = list(a)
>>> c
[0, 1, 2, 3, 4]
```

上述代码中，变量 empty_list 绑定到空列表[ ]。变量 a 绑定到列表[0，1，2，3，4]。注意，range(5)中的元素依次是 0，1，2，3，4。变量 b 绑定到列表['P'，'y'，'t'，'h'，'o'，'n']。特别需要注意的是，变量 c 绑定的列表的值也是[0，1，2，3，4]，但和变量 a 绑定的列表并不是同一个列表。验证这一点非常容易，可以执行语句 c[0] ＝ 1，然后查看列表 a 和 c 的值。

```
>>> c[0] = 1
>>> c
[1, 1, 2, 3, 4]
>>> a
[0, 1, 2, 3, 4]
```

创建列表的第三种方法是使用列表推导，形如[expression for x in iterable]，其含义是依次将可迭代对象 iterable 中的元素赋值给变量 x，然后基于 x 的值求解表达式 expression，并将表达值 expression 的值放入新建的列表中。例如，[x ** 2 for x in range(5)]会创建列表[0，1，4，9，16]。这是因为变量 x 首先绑定到 range(5)的第一个元素 0，此时表达式 x ** 2 的值(即 0)会加入列表，然后 x 绑定到 range(5)的第二个元素 1，此时表达式 x ** 2 的值(即 1)会加入列表，以此类推，直至遍历完 range(5)中的所有元素。

```
>>> [x ** 2 for x in range(5)]
[0, 1, 4, 9, 16]
```

注意，表达式 expression 可能与变量 x 无关。例如，matrix＝[[0，0，0] for x in range(3)]创建了一个嵌套列表[[0，0，0]，[0，0，0]，[0，0，0]]。这里无论 x 取值如何，表达式 expression 的值始终是[0，0，0]。再次强调，虽然 matrix[0]、matrix [1]、matrix [2]的值都是[0，0，0]，但实际上它们绑定到不同的列表，所以修改 matrix[0]的值并不影响 matrix[1]和 matrix[2]的值。

```
>>> matrix = [[0, 0, 0] for x in range(3)]
>>> matrix
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> matrix[0][0] = 1
>>> matrix
[[1, 0, 0], [0, 0, 0], [0, 0, 0]]
```

列表推导中还可引入条件判断，形如[expression for x in iterable if condition]，其含义是依次将可迭代对象 iterable 中的元素赋值给变量 x，并且仅当表达式 condition 为真时才

求解表达式 expression，并将其值放入新建的列表中。例如[x for x in range(10) if x % 2 == 1]会创建列表[1, 3, 5, 7, 9]。这是因为仅当表达式 x % 2 == 1 为真时(即 x 为奇数)，才会将表达式 x 的值加入列表中。

```
>>> [x for x in range(10) if x % 2 == 1]
[1, 3, 5, 7, 9]
```

又如[x for x in 'alice' if x in 'aeiou']会创建列表['a', 'i', 'e']。这是因为仅当表达式 x in 'aeiou'为真时，才会将表达式 x 的值加入列表中。

```
>>> [x for x in 'alice' if x in 'aeiou']
['a', 'i', 'e']
```

3.3 列表基本操作

3.3.1 索引

创建列表之后，提取列表中的元素是常规需求。在 3.1 节中提到，可以通过索引提取单个元素。如果列表中包含 n 个元素，第一个元素的索引是 0，最后一个元素的索引是 $n-1$。此外，Python 还支持负索引，并规定倒数第一个元素的索引是 −1，倒数第二个元素的索引是 −2，其余以此类推。图 3-3 显示了一个列表中所有元素的正索引和负索引。不难看出，某元素的负索引加上该列表的长度恰好等于该元素的正索引。所以，长度为 n 的列表，其索引的有效范围是 $-n \sim n-1$。

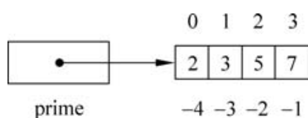

图 3-3 列表的正索引和负索引

注意，如果索引不在有效范围之内，那么代码在运行时会产生越界错误。

```
>>> primes = [2, 3, 5, 7]
>>> primes[4]              # 执行该语句会产生运行错误
IndexError: list index out of range
>>> primes[-5]             # 执行该语句会产生运行错误
IndexError: list index out of range
```

从图 3-3 可以看出，列表 prime 上索引的有效范围是 −4～3，所以索引 4 和索引 −5 都会越界，因此上述代码会产生运行错误。

3.3.2 切片

通过索引可以提取列表的单个元素，而通过切片可以提取列表的多个元素并组成一个新的列表。对于列表 L，切片的一般形式是 L[start: stop]，其中整数 start 和 stop 分别表示切片的开始和结束位置。如图 3-4 所示，切片 L[start: stop]从索引 start 开始，向右依次提取元素，直至索引 stop 结束(索引为 stop 的元素并不包含在内)。也就是说，切片 L[start: stop]从 L 中提取 stop−start 个元素，生成列表[L[start]、L[start+1]、…、L[stop−1]]。例如 L[1: 4]提取 L[1]、L[2]和 L[3]3 个元素，构成列表[3, 5, 7]。如果 start 大于或等于

stop,那么切片提取不到任何元素,返回一个空列表。

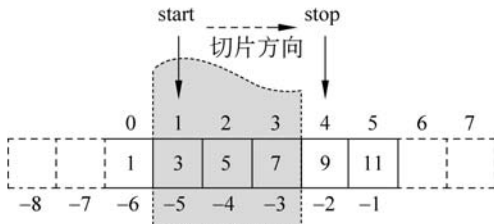

图 3-4　列表上的切片操作

为了方便理解切片,可以认为 start 和 stop 定义了一个区域(见图 3-4 的灰色框),而切片结果就来自位于这个区域之内的列表元素。当 start 大于或等于 stop 时,这个区域显然为空,所以列表没有任何元素属于这个区域,从而切片结果是一个空列表。

在切片时可以省略 start 或者 stop。如果省略 start,那么就认为 start 的值是 0,即从头提取元素。如果省略 stop,那么就认为 stop 的值是列表 L 的长度,即提取到列表结尾。例如 L[：3]的值是[1，3，5],而 L[3：]的值是[7，9，11]。如果两者都省略,那么切片构成的列表和列表 L 具有相同的值,例如 L[：]的值是[1，3，5，7，9，11]。

切片可以指定步长,形式为 L[start：stop：step],表示从 L[start：stop]中每隔 step−1 个元素进行提取。考虑切片 L[1：5：2],因为 L[1：5]是[3，5，7，9],从元素 3 开始,每隔 1 个元素进行提取,所以最后结果是[3，7]。再考虑切片 L[：：3],注意到 L[：]是[1，3，5，7，9，11],从元素 1 开始,每隔 2 个元素进行提取,所以结果是[1，7]。如果不指定步长,那么使用默认值 1。

注意,步长 step 不能等于 0,但可以小于 0,此时从索引 start 开始,向左提取元素。具体来说,当 step 为负时,L[start：stop：step]表示从 L[start：stop：−1] 中每隔|step|−1 个元素进行提取。例如要分析切片 L[5：1：−2]的值,可以先考虑 L[5：1：−1]。步长为负意味着提取方向是从右向左,从图 3-5 中不难看出 L[5：1：−1]的结果是[11，9，7，5]。因此 L[5：1：−2]从元素 11 开始,每隔|−2|−1=1 个元素进行提取,所以 L[5：1：−2]的值是[11，7]。注意,当 step 为负时,如果 start 小于或等于 stop,那么切片提取不到任何元素,返回一个空列表。

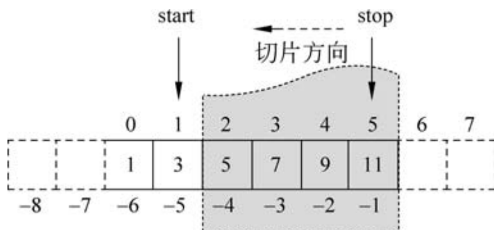

图 3-5　步长为负时的切片

记住,在切片时可以省略 start 或者 stop,这种省略在切片步长为负时表现更加智能。例如切片[:4:3]和切片[:4:−3]中都省略了 start。由于步长为正意味着从左向右提取元素,因此切片[:4:3]认为 start 的值是 0,即从头开始提取元素。而步长为负意味着从右向

左提取元素，因此切片[:4:-3]认为start的值是5，即从列表结尾开始提取元素。因此，两个切片值分别是[1, 7]和[11]。

索引可能导致越界错误，而切片却不会。之前提过可以认为start和stop定义了一个区域，从这个角度就容易理解为什么切片不会导致越界错误。如图3-6所示，stop=7相对于列表L来说不是一个有效索引，然而位于start和stop定义的灰色区域之内的列表元素仍然有9和11，所以切片L[4: 7]的值是[9, 11]。那么切片L[4: 100]呢？显然还是同样的结果。

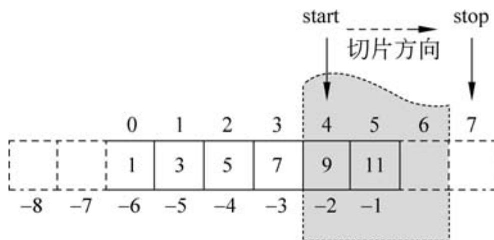

图3-6　切片的容错性

上述讨论中考虑的都是正索引，不要忘记Python还支持负索引，所以切片中的start和stop也可以是负整数，此时上面介绍的规则同样适用。先考虑简单的情况，例如切片L[-5: -2]。因为步长为正，所以从索引-5开始向右提取元素直至索引-2(不包含该元素)，这正是图3-4所示的情况。切片L[-1: -5]的结果如何？注意，这里步长为正，还是向右提取，显然此时start和stop定义的区域为空，因此切片结果是一个空列表。那么切片L[-1: -5: -1]呢？不难看出，该切片正好对应图3-5，结果是[11, 9, 7, 5]。最后考虑切片L[: -100: -2]。因为步长为负，所以start的值是5，这样位于start和stop定义区域内的列表元素是11, 9, 7, 5, 3, 1。而步长为-2意味着每隔一个元素提取，所以切片结果是[11, 7, 3]。

3.3.3 连接和重复

通过"+"运算符可以将两个列表连接起来形成一个新列表。

```
>>> [1, 2] + [3, 4]
[1, 2, 3, 4]
```

通过"*"运算符可以将一个列表中的元素重复多次并形成一个新列表。

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

如果列表中的元素是可变类型，那么重复操作只是复制了对该元素的引用。

```
>>> matrix = [[0, 0]] * 2              # 语句 1
>>> matrix
[[0, 0], [0, 0]]
>>> matrix[0][0] = 1                   # 语句 2
>>> matrix
[[1, 0], [1, 0]]
```

考虑上述代码,为什么修改 matrix[0][0]后,matrix[1][0]的值也发生变化?从图 3-7 不难看出原因。图 3-7(a)展示了语句 1 执行后的情况,matrix[0]和 matrix[1]引用了同一个列表[0,0]。图 3-7(b)展示了语句 2 执行后的情况,matrix[1]引用的列表已经通过 matrix[0]发生了变化。如果要避免复制引用带来的影响,可以使用下面的方法。

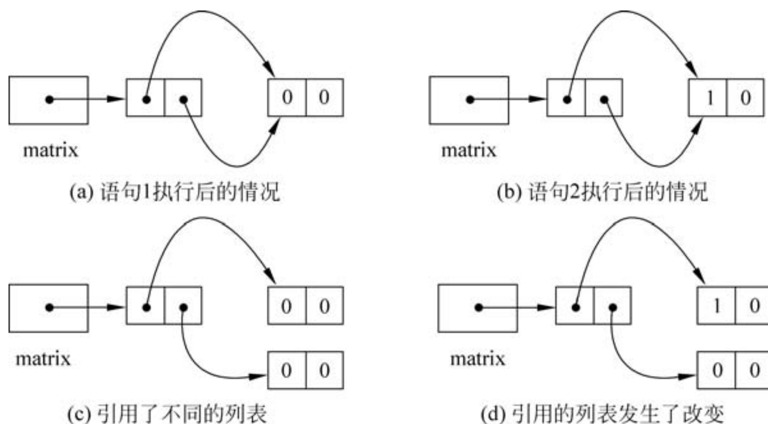

图 3-7 对象引用对列表重复操作的影响

```
>>> matrix = [[0, 0] for i in range(2)]          # 语句 1
>>> matrix
[[0, 0], [0, 0]]
>>> matrix[0][0] = 1                             # 语句 2
>>> matrix
[[1, 0], [0, 0]]
```

上述代码中,语句 1 使用列表推导创建了一个列表。如图 3-7(c)所示,现在 matrix[0] 和 matrix[1]引用了不同的列表,虽然它们的值都是[0, 0]。执行语句 2 之后,只有 matrix [0]引用的列表发生了改变,结果如图 3-7(d)所示。

3.3.4 查询操作

本节介绍几种和查询相关的操作。因为这些操作没有对列表进行修改,所以它们也适用于其他的不可变序列,例如字符串和元组。

运算符 in 用来判断一个对象是否在一个列表中。如果对象 x 在列表 L 中,那么表达式 x in L 的值是 True,否则是 False。

```
>>> 3 in [1, 2, 3]
True
>>> 4 in [1, 2, 3]
False
```

函数 len(L)返回列表 L 的长度。函数 min(L)和函数 max(L)分别返回列表 L 中的最小值和最大值。注意,这两个函数能够正确工作的前提是列表 L 中的元素可以比较大小。例如语句 min([0, '1'])会导致一个名为 TypeError 的运行时错误,因为整型对象 0 和字符串对象'1'无法比较大小。

```
>>> L = [2, 3, 5, 7]
>>> len(L)
4
>>> min(L)
2
>>> max(L)
7
>>> min([0, '1'])                #该操作会导致运行时错误
TypeError: '<' not supported between instances of 'str' and 'int'
```

操作 L.count(x)返回对象 x 在列表 L 中的出现次数。

```
>>> L = [3, 1, 4, 1, 5, 9, 2, 6, 5]
>>> L.count(1)
2
>>> L.count(7)
0
```

操作 L.index(x)返回对象 x 在列表 L 中第一次出现的位置,而操作 L.index(x, start, stop)返回对象 x 在列表 L 特定范围内第一次出现的位置,该范围开始于 start,结束于 stop(不包括在内)。如果该范围内对象 x 没有出现,则该操作会导致一个名为 ValueError 的运行错误。

```
>>> L = [3, 1, 4, 1, 5, 9, 2, 6, 5]
>>> L.index(5)
4
>>> L.index(5, 5, len(L))
8
>>> L.index(7) #该操作会导致运行时错误
ValueError: 7 is not in list
```

3.3.5 修改操作

本节介绍两类常见的修改列表的操作:插入元素和删除元素。

向列表中插入元素有三种常见的方法。第一,操作 L.append(x)将对象 x 添加至列表 L 的末端。第二,操作 L.extend(iterable)将可迭代对象 iterable 中的元素依次添加至列表

L 的末端。值得注意的是，运算符“+=”也可以实现同样的功能。第三，操作 L.insert(i,x)将对象 x 添加至列表 L 中索引为 i 的位置。insert()方法不检查索引 i 是否越界，同时也支持负索引。

```
>>> L = [1, 2]
>>> L.append(3)
>>> L
[1, 2, 3]
>>> L.extend([4, 5])             # 将列表[4, 5]中的元素依次添加至列表 L 的末端
>>> L
[1, 2, 3, 4, 5]
>>> L.append([4, 5])             # 将列表[4, 5]作为一个对象添加至列表 L 的末端
>>> L
[1, 2, 3, 4, 5, [4, 5]]
>>> L.insert(3, 0)               # 将整数 0 添加至列表 L 中索引为 3 的位置
>>> L
[1, 2, 3, 0, 4, 5, [4, 5]]
>>> L.insert(-1, 0)              # insert()方法支持负索引
>>> L                            # 注意:插入到索引为 -1 的位置会导致最后一个元素后移
[1, 2, 3, 0, 4, 5, 0, [4, 5]]
>>> L.insert(100, 0)             # insert()方法不检查索引 i 是否越界,并且能智能处理
>>> L
[1, 2, 3, 0, 4, 5, 0, [4, 5], 0]
```

删除列表中的元素也有多种方法。操作 L.clear()删除列表 L 中的所有元素，最终 L 成为一个空列表。操作 L.pop(i)返回列表 L 中索引为 i 的元素并将其从列表中删除。如果没有指定 i，那么 i 取默认值−1，即列表 L 中最后一个元素。操作 L.remove(x)删除列表 L 中第一次出现的对象 x。

```
>>> L = [1, 2, 1, 3, 1, 4]
>>> L.pop(4)             # 返回索引为 4 的元素并将其从列表 L 中删除
1
>>> L
[1, 2, 1, 3, 4]
>>> L.pop()              # 返回列表 L 中最后一个元素并将其从列表 L 中删除
4
>>> L
[1, 2, 1, 3]
>>> L.remove(1)          # 删除列表 L 中第一次出现的 1
>>> L
[2, 1, 3]
>>> L.clear()            # 删除列表 L 中的所有元素
>>> L
[]
```

除了上面介绍的方法，语句 del 也可以删除列表中的元素，但用法更加复杂。语句 del L[i]删除列表 L 中索引为 i 的元素。注意，索引 i 可以为负，但不能超过有效范围，否则会导致名为 IndexError 的运行错误。语句 del L[i:j:k]从列表 L 中删除切片 L[i:j:k]包含的

元素。

```
>>> L = [1, 2, 3, 4, 5, 6, 7, 8]
>>> del L[::3]
>>> L
[2, 3, 5, 6, 8]
>>> del L[1:4]
>>> L
[2, 8]
>>> del L[1]
>>> L
[2]
>>> del L                #注意,该语句删除了变量 L,而不是 L 中的元素
>>> L                    #变量必须先赋值再使用,所以该语句会导致运行时错误
NameError: name 'L' is not defined
```

3.3.6 反转

Python 提供了操作 L.reverse()来反转列表 L 中的元素,即第一个元素变成倒数第一个元素,第二个元素变成倒数第二个元素,其余以此类推。

```
>>> L = [1, 2, 3, 4, 5]
>>> L.reverse()
>>> L
[5, 4, 3, 2, 1]
```

可以看出 L.reverse()修改了列表 L。还有一些方法可以在不修改 L 的情况下反转 L 中的元素。步长为−1 的切片是一种选择。此外,Python 还提供了函数 reversed(L)。该函数将 L 中的所有元素反转,并以迭代器的形式返回。

```
>>> L = [1, 2, 3, 4, 5]
>>> L[::-1]              #使用切片进行列表元素的反转,可以自行验证列表 L 有没有发生变化
[5, 4, 3, 2, 1]
>>> list(reversed(L)) #将函数 reversed()返回的迭代器作为参数调用函数 list()构造列表
[5, 4, 3, 2, 1]
```

3.3.7 复制

复制列表也是常见操作之一。除了使用 list()函数和切片操作,列表对象的 copy()方法也可以实现复制。

```
>>> L = [1, 2, 3]
>>> A = list(L)
>>> A
[1, 2, 3]
>>> B = L[:]
>>> B
```

```
[1, 2, 3]
>>> C = L.copy()
>>> C
[1, 2, 3]
```

上述方法创建的列表 A、B 和 C 与 L 具有相同的值,但它们分别绑定到不同的列表对象。为了验证这一点,可以使用运算符“==”和运算符“is”做一些测试。

```
>>> L = [1, 2, 3]
>>> C = L.copy()
>>> L == C
True
>>> L is C
False
```

回忆一下,直接使用赋值语句会使得两个变量绑定到同一个列表。有时候这种复制引用并不是期望的行为。

```
>>> L = [1, 2, 3]
>>> A = L
>>> L == A
True
>>> L is A
True
```

对于嵌套列表,复制会更加复杂。考虑下面的例子。

```
>>> ?L = [[1, 2], [3, 4]]
>>> A = L.copy()
>>> L is A
False
>>> A
[[1, 2], [3, 4]]
>>> L[0][0] = 10
>>> A
[[10, 2], [3, 4]]
```

列表对象的 copy()方法将 L 的值复制到 A。然而,赋值语句修改了 L[0][0]之后,A[0][0]的值也发生了变化。造成这种情况的原因是 copy()方法仅支持浅拷贝,即仅仅复制顶层对象,但对于顶层对象的内部对象,复制的还是它们的引用。在上面的例子中,L[0]和 A[0]实际上引用了同一个列表[1, 2]。

为了解决这一问题,Python 提供了 deepcopy()方法来实现对象的深拷贝,即完整地复制整个对象。为了使用该方法,首先需要导入 copy 模块。

```
>>> import copy
>>> L = [[1, 2], [3, 4]]
>>> A = copy.deepcopy(L)
```

```
>>> L[0][0] = 10
>>> A
[[1, 2], [3, 4]]
```


3.4 列表与排序

对列表中的元素进行排序也是常见需求之一。这在 Python 中非常简单，直接使用列表对象的 sort()方法即可。

```
>>> numbers = [3, 1, 4, 1, 5, 9, 2]
>>> numbers.sort()
>>> numbers
[1, 1, 2, 3, 4, 5, 9]
```

可以看出，使用 sort()方法之后，列表 numbers 的内容发生了变化。注意，该方法的返回值是 None。如果误认为该方法的返回值是排序后的列表，并将该方法放在赋值语句的右边，那么赋值语句左边的变量会绑定到 None 对象上，这显然不是期望发生的情况。

```
>>> numbers = [3, 1, 4, 1, 5, 9, 2]
>>> numbers = numbers.sort()          # 该语句会让 numbers 绑定到 None 对象
>>> print(numbers)
None
```

sort()方法对列表正确排序的前提是列表中任意两个对象之间定义了小于(<)关系。对于整数或者浮点数，这种关系是显然的。对于字符串，会按照字典序进行排列。

```
>>> numbers = ['one', 'two', 'three', 'four']
>>> numbers.sort()
>>> numbers
['four', 'one', 'three', 'two']
```

如果列表中的元素又是列表，该如何排序呢？这就涉及如何判断两个列表的大小关系。给定两个列表，从左向右逐个元素进行比较，直至遇到不相等的元素，此时元素小的列表就小(例如列表[1，2]小于列表[1，3])。如果有一方没有元素可以用于比较，那么这一方就小(例如列表[1，2]小于列表[1，2，3])。下面的 Python 代码对一个嵌套列表进行了排序。

```
>>> nested_list = [[1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]
>>> nested_list.sort()
>>> nested_list
[[1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]
```

sort()方法默认将列表中的元素从小到大排序。如果需要从大到小排序，那么可以在调用 sort()方法时指定参数 reverse=True。

```
>>> numbers = [3, 1, 4, 1, 5, 9, 2]
>>> numbers.sort(reverse = True)            #按照降序排列
>>> numbers
[9, 5, 4, 3, 2, 1, 1]
```

可以为 sort()方法的参数 key 指定一个函数，重新定义列表中任意两个对象之间的大小关系。该函数只有一个参数，通常是列表中的对象，而返回值可以看成是该对象的“值”，用来确定对象之间的大小关系。考虑包含多个字符串的列表 names，现在要将其中的元素按照它们的长度进行排序，那么可以定义一个函数 helper(s)，其返回值是参数 s 的长度。将该函数传递给 sort()方法的参数 key，那么就能根据字符串的长度而不是按字典顺序进行排序了。

```
>>> def helper(s):
...   return len(s)
>>> ?names = ['David', 'Alice', 'Charlotte', 'Bob']
>>> names.sort(key = helper)
>>> names
['Bob', 'David', 'Alice', 'Charlotte']
```

注意，sort()方法是稳定的排序。比如，在上面的例子中，字符串'David'和字符串'Alice'的长度相等，而它们在排序前后的相对顺序维持不变。

目前介绍的排序都是按照单一条件进行排序，有时候还需要根据多个条件进行排序。考虑上面的姓名列表 names，现在首先需要根据姓名长度进行排序，对具有相同长度的姓名，再按照字典顺序进行排序。如何实现这类复杂的排序呢？还是需要借助传递给参数 key 的函数，只不过该函数的返回值是一个列表(其实返回一个元组更自然，元组的概念要在第 6 章介绍)，其中第一个元素对应排序的第一个条件，即姓名的长度；第二个元素对应排序的第二个条件，即姓名本身：

```
>>> def helper(s):
...   return [len(s), s]
>>> names = ['David', 'Alice', 'Charlotte', 'Bob']
>>> names.sort(key = helper)
>>> names
['Bob', 'Alice', 'David', 'Charlotte']
```

为什么会有这样的排序结果？关键在于上述函数 helper()返回的是一个列表，而 sort()方法要根据该列表来判断两个姓名(即字符串)的大小关系。之前已经讨论过如何判断两个列表的大小关系，这里再简单解释一下：首先比较两个列表的第一个元素，即姓名的长度，如果不相等，则大小关系已经给出；否则继续比较第二个元素，即姓名本身，也就是字符串的字典顺序。

根据多个条件排序时，还可以为每个条件指定是升序排列还是降序排列。继续考虑上面的排序例子，现在要求按姓名长度降序排列。对于这类数值型的排序条件，可以简单地使用相反数，即将原数值乘以－1：

```
>>> def helper(s):
...   return [-1 * len(s), s]
>>> names = ['David', 'Alice', 'Charlotte', 'Bob']
>>> names.sort(key = helper)
>>> names
['Charlotte', 'Alice', 'David', 'Bob']
```

如果现在要求先按姓名长度降序排列，然后再按姓名字顺序降序排列，该如何处理呢？因为第二个条件是字符串类型，所以不能通过乘以−1来实现降序排列。这里介绍一个更通用的做法。首先根据第二个条件（也就是排序次要条件）来对列表排序，然后再根据第一个条件（也就是排序首要条件）来对列表排序：

```
>>> def helper(s):
...   return len(s)
>>> names = ['David', 'Alice', 'Charlotte', 'Bob']
>>> names.sort(reverse=True)                    # 首先根据次要条件(即字典序)降序排列
>>> names                                       # 此时字符串'David'在字符串'Alice'之前
['David', 'Charlotte', 'Bob', 'Alice']
>>> names.sort(key = helper, reverse=True)      # 然后根据首要条件(即姓名长度)降序排列
>>> names      # 由于 sort()方法的稳定性,此时字符串'David'仍然在字符串'Alice'之前
['Charlotte', 'David', 'Alice', 'Bob']
```

上述代码能够正确排序的关键在于 Python 提供的 sort()方法是一个稳定的排序方法。根据首要条件对相关对象进行排序时，不会改变次要条件决定的这些对象之间的顺序。

上面介绍的方法容易扩展到三个甚至更多条件的排序，这里不再赘述。此外，传递给 sort()方法的函数 helper()通常被定义为匿名函数，更多关于函数定义的细节将在后续章节中介绍。

除了列表对象的 sort()方法外，Python 还提供了通用的排序函数 sorted()。该函数返回已经排好序的列表副本，并且不会改变待排序列表的内容：

```
>>> numbers = [3, 1, 4, 1, 5, 9, 2]
>>> sorted(numbers)
[1, 1, 2, 3, 4, 5, 9]
>>> numbers # 列表的内容没有改变
[3, 1, 4, 1, 5, 9, 2]
>>> sorted(numbers, reverse=True)
>>> numbers
[9, 5, 4, 3, 2, 1, 1]
```

习题

1. 自行定义两个列表，列表元素个数大于10。将两个列表合并，然后截取第8～15个元素，输出最后得到的结果。
2. 求无序整数列表的中位数。如列表元素为偶数个，则取列表升序排列时中间两数中

数值较小的元素为中位数。

3. 已知一个整数列表，判断列表内容是否为回文，即无论正序还是倒序，列表的内容是否相同。

4. 随机生成一个10以内整数平方的列表，要求从大到小排序。

5. 随机生成一个20以内的奇数列表再随机生成一个[0,20]的整数unm，判断unm是否在列表中存在。

6. 现有一个列表[1,3,4,6,6,7,8,8,10,21,22,22]，编写程序，直接操作列表，使得列表不存在重复元素，且元素均小于10。

7. 现有一组列表存放了若干姓名，例如["张三","李四","王五"]。编写程序，将这些姓名的姓氏单独组成一个列表并输出，假定不存在复姓。

8. 已知一个整数列表，筛选出该列表中不同的素数(又称质数)，并求出该列表中有多少个素数可以表达为该列表中另外两个素数的和。

9. 筛法是一种用来求所有小于 N 的素数的方法。编写程序，基于筛法求500之内的所有素数，并打印输出这些素数，每行输出5个素数。

10. 现有列表[35,46,57,13,24,35,99,68,13,79,88,46]，编写程序将其中重复的元素去除，并按从小到大的顺序排列后输出。

11. 编写程序，对一个4×4的矩阵进行随机赋值，然后对该矩阵进行转置，并输出转置后的结果。

12. 现有5名同学期中考试高等数学和线性代数成绩如表3-1所示。

表3-1　5名同学期中考试高等数学和线性代数成绩

姓　　名	高等数学/分	线性代数/分
张飞	78	75
李大刀	92	67
李墨白	84	88
王老虎	50	50
雷小米	99	98

编写程序，按照总分从高到低进行排序后输出姓名和成绩。

13. 现有5名同学期末考试高等数学和线性代数成绩如表3-2所示。

表3-2　5名同学期末考试高等数学和线性代数成绩

姓　　名	高等数学/分	线性代数/分
张飞	78	75
李大刀	92	67
李墨白	84	88
王老虎	84	50
雷小米	92	98

编写程序，按照高数成绩从高到低进行排序，如果高等数学分数一样，则按照线性代数分数从高到低排序，最后输出姓名和相关成绩。

14. 从键盘输入 n，打印 n 阶魔方阵(n 为奇数)。魔方阵的每一行、每一列和两个对角线的和都相等。