第3章 深谈Python函数、变量、数据类型与输入输出

引 言

有了前两章的编程基础，接下来深入学习 Python 语言中的一些重要知识——函数、变量、数据类型、输入输出。函数作为 Python 的重要组成部分，我们要了解函数是怎么编写的？什么是好的函数编写方式？不要撰写出可能有“副作用”的函数。接着讨论全局变量和局部变量的差异，体会为什么在函数中要尽量少用全局变量；也会讲解参数的传递和嵌套函数的各种知识。除了对 Python 函数部分的学习，我们也将学习一些数据类型，包括列表、字符串、元组和字典等，它们都是编程中会经常被用到的。其中，列表、字符串和元组这种可以通过下标来索引到内部元素的结构都算是序列，而字典实际上是一种映射。本章除了详细描述这些数据类型的使用外，更强调了可能会出错之处和作为函数参数传递时的注意事项。最后将介绍输入输出、文件操作与异常处理。学完本章之后，同学们会对 Python 的使用更加得心应手，并且可以避免许多 Python 编程特有的错误。

3.1 深入了解函数的各种性质

前面曾简单讲解过函数，包括函数的基本概念、基本形式以及一些小例子。本节将带领大家深入了解函数的作用；理解如何避免函数执行时的错误；掌握函数中参数、返回值、变量、嵌套函数等的性质，从而让大家能够编写完整而又完美的函数。

3.1.1 编写完美函数

在讲解完美函数的思想前，首先大家需要清楚的是，为什么需要函数？我们在编程的时候，有很多操作过程是重复的，也就是说，功能是一样的，只是处理的数据不一样。比如说在前面的章节中我们介绍过多项式计算的问题，在求解多项式乘法的时候其实就是多次多项式加法的应用，这时没有必要来回写这些重复的代码，而是可以将加法写成一个函数，在编写多项式乘法的代码时只需要在 for 循环中调用它就可以了。再比如有 A 函数、B 函数和 C 函数，在执行的过程中都需要进行排序，那么我们就可以单独写一个排序函数，方便不同的函数分别调用它。由此可知，在编写程序时，我们可以构造一些基本函数，方便后面的其他函数来调用它，从而实现完整的功能。

对于函数的思想，我们可以将其理解为把完成某一功能的代码封装到一个“盒子”里，再给它起一个名字。每次需要执行这个功能的时候就把它调出来，输入我们想进行处理的数据，最终这个盒子会返回给我们想要的返回值。整个流程如图 3-1 所示。

我们希望自己所定义的函数是个“完美”的函数。所谓“完美函数”，它应该就像一个封装好的黑盒子，只需要传递给它需要的参数，盒子里面进行的一系列操作都是独立的，此函数的执行不会影响到外界的环境（如外面的变量等）。这是什么意思呢？下面通过图 3-2 来解释。

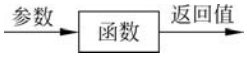


图 3-1　函数示意图

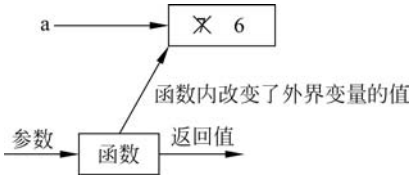


图 3-2　非完美函数示意图

在图 3-2 中，变量 a 既不是函数的输入也不是函数的输出，只是函数外面的一个变量。在执行函数之前，a＝7；而执行了函数之后，a＝6，即在函数执行的过程中竟然改变了外界的环境，这就不是一个完美的函数，而是一个比较“危险”的函数。编写函数时，我们希望的是在函数执行过程中，除了传入的参数以外，应尽量不改变其外界的环境。

3.1.2 参数与返回值

现在再来看看函数中的参数与返回值。对于 Python 中函数的参数，它们是在函数内

才有意义的变量,我们可以将其理解为是“随机应变”的容器。至于容器中将要装的是什么类型的值,函数在定义时并不确定。所以与其他语言(如 C、Java 等)不同的是,C 或 Java 在编写函数时一定要明确声明参数的类型,但 Python 不需要声明参数的类型,只有在函数调用时参数的真实类型才会被绑定。

此外,在前面也介绍过,函数中参数的个数是随意的,但即使没有参数,函数后面的括号也是必须要有的。

兰　兰:如果我定义的函数本来是希望判断一个整数是否奇数,比如 def is_odd(a): return not a%2==0,但执行时传入进来的却是个列表,那么程序是不是就会出错而终止?但我们不希望因为一个函数的参数类型错误而被全盘终止。

沙老师:没错!所以说,在编写 Python 函数时应该主动去检查所传参数的类型是否我们想要的类型,比如对于 is_odd 函数,应该添加如“if type(a) != type(1): return False”的语句来判断 a 是否为整数。为了减少篇幅,本书默认所有程序调用函数时所传的参数类型都是正确的,但是同学们在编写正式程序时需要主动检查所传入的参数类型。

对于函数的返回值,可以有也可以没有。如果有返回值,就必须要有 return 关键字,多个返回值之间用逗号隔开,Python 允许函数返回多个返回值这一特点为我们的编程提供了很大的方便!同样,返回值的类型也不需要声明,Python 会自行判断返回值的类型。

下面举一个具体的例子来说明,见<程序:参数与返回值举例>。该例子所实现的功能是检验一个给定的序列中指定位置的字符是否要查找的字符。如果指定的位置超过了字符串的长度,则返回“Error:position exceeds length of string”。

```
#<程序:参数与返回值举例>
def find(str,pos,key):
    if(len(str)<pos):                    #指定的位置大于字符串的长度
        return False,-99
    else:
        if(str[pos]==key):
            return True,True
        else:
            return True,False
mystr = "abcdefgh"
correct,res = find(mystr, 2, 'c')
if(not correct): print("Error:position exceeds length of string")
elif res: print("Find it!")
else: print("Not in it!")
```

可以看到,函数 find 的参数有 str、pos 和 key,我们并没有声明它们的类型,其实它们可以承载任意类型的变量,只是在调用 find 函数的时候传入的参数为 mystr、2 和'c',即分别是字符串型、整型和字符类型,也就是我们所希望的参数类型。请各位同学自己试验一下:find([7,3,2,8],1,3)的执行结果是否是“Find it!”。

3.1.3　局部变量与全局变量

对于变量，可以分为局部变量和全局变量。局部变量也就是只能在特定的函数中可以访问的变量，而全局变量是定义在所有函数最外面的变量。学会分辨哪些是局部变量，哪些是全局变量对编程者来说十分重要，否则编程时就可能无法判断我们想要改变的变量是否已经被改变了，从而导致代码的混乱。首先给出分辨局部变量与全局变量的规则。

分辨局部变量与全局变量的规则

假设有一个变量为 a，它出现在函数 f()中，应怎样判断它在函数内是什么变量？可以使用如下规则来判定：

(1) 如果有 global 关键字修饰变量 a，则 a 为全局变量。

(2) 否则，假如 a 是参数或者出现在等号左边，则 a 是局部变量。

(3) 否则，a 与函数 f 外层的变量 a 的属性相同。

注意：

- 在判断变量是局部还是全局变量的时候一定要按照上述规则的顺序逐个判断。
- 对于定义在所有函数外的变量，叫作**全局变量**。
- 假如 f()不是下面所说的嵌套函数(嵌套函数是指在函数内定义的函数)，那么在 f()外层的变量就是全局变量。
- 如果有 global 关键字修饰，那么无论 f()是不是嵌套函数，被定义的变量都是全局变量。
- 一个较为完美的函数，应尽量少用全局变量，尽量使用参数来进行传值，并且尽量不要在函数内更改全局变量。

先来看这样一个例子，见<程序：局部变量与全局变量举例 1>，对应规则(1)。

```
#<程序：局部变量与全局变量举例 1>
a = 1                    #所有函数最外面的变量，全局变量
def fun(x,y):
    global a             #global 关键字表明 a 是全局变量
    a = x+y
    return a
sum = fun(10, 100)
print(a)
```

这个例子的打印结果是 a=110。对于 a = 1 中的 a，它是定义在所有函数外面的变量，故为全局变量。又因为 global 关键字把 fun()函数中的变量 a 变成了全局变量(global 关键字后面跟着一个或多个用逗号分开的变量名)，所以在执行 a = x+y 的时候就改变了全局变量 a 原来的值，则 a 的值由 1 变成了 110。

再来看第二个例子，见<程序：局部变量与全局变量举例 2>，对应规则(2)。

```
#<程序：局部变量与全局变量举例 2>
a=1                      #所有函数最外面的变量，全局变量
def fun(x,y):
```

```
    a = x + y              #a在函数内的等号左侧,局部变量,不改变全局变量的值
    return a
sum = fun(10, 100)
print(a)                   #打印的是函数外的变量a
```

该例中,最终打印的结果是a=1。根据规则,在fun()函数中,a没有被global关键字修饰,则判断是否符合条件(2),可以看出a符合“出现在等号左边”的条件,因此,fun()函数中的a是局部变量。所以,在fun()函数中的操作实际上改变的是局部变量a的值,并不会影响函数外面的全局变量a,也就是说,这两个a没有任何的关系,所以最终输出的结果a的值还是1。接下来,再思考一下x和y是什么变量?它们都不满足规则(1),但满足规则(2),即都是函数的参数,故均为局部变量。

再根据规则(3)做一个举例,见<程序:a,b,c,d是否为局部变量?>。

```
#<程序:a,b,c,d是否为局部变量?>
b,c = 2,4                  #在所有函数最外层,即全局变量
def g_func():
    a = b * c              #a是局部变量
    d = a                  #d是局部变量,b和c都是全局变量
    print(a,d, ';',end = "")
g_func()
print(b,c)
```

上述程序的输出结果如下:

```
8  8 ; 2  4
```

在函数g_func()中,变量a和d是局部变量,因为它们没有被声明为global且出现在等号左边。变量b和c是全局变量,尽管它们没有被声明为global,但是它们不是函数的参数,且只是出现在g_func()函数中语句的等号右边,即不满足规则(1)和(2),则利用规则(3)来判断,变量b和c与函数外层的变量b和c属性相同。函数外层的变量b和c都是全局变量,所以在g_func()函数内部的变量b和c也是全局变量。

为加深理解,下面给出一个较为复杂的Python代码,见<程序:复杂运算例子>。请同学们先自行思考,这个例子最终的打印结果是什么。

```
#<程序:复杂运算例子>
def do_sub(a, b):
    c = a - b              #a、b、c都是do_sub()函数中的局部变量
    print(c)
    return c
def do_add(a, b):          #参数a和b是do_add()函数中的局部变量
    global c
    c = a + b              #全局变量c,修改了c的值
    c = do_sub(c, 1)       #再次修改了全局变量c的值
    print(c)
#所有函数外先执行
```

```
a = 3                                  # 全局变量 a
b = 2                                  # 全局变量 b
c = 1                                  # 全局变量 c
do_add(a, b)                           # 全局变量 a 和 b 作为参数传递给 do_add()
print(c)                               # 全局变量 c
```

该例子输出的结果是“4，4，4”。

3.1.4　嵌套函数

在 Python 中，有时候函数不只有一层，有可能会有多层函数嵌套的情况，在这种情况下，更加应当注意对局部变量和全局变量的分辨。

嵌套函数是指在函数内定义的函数，嵌套函数如同局部变量那样是个“局部”函数，它只能在外层定义它的函数中使用。以前面讲过的选择排序作为例子，我们将代码修改一下，见<程序：嵌套函数举例>。

```
#<程序：嵌套函数举例>
def selection_sort(L):
    def find_min(L):                   #返回 L 中最小值所在的索引
        min = 0                        #这个 min 是 find_min()函数中的局部变量
        for i in range(len(L)):
            if L[i]< L[min]: min = i
        return min
    for i in range(len(L) - 1):
        min = find_min(L[i:])          #min 是 selection_sort()函数中的局部变量
        L[min + i],L[i] = L[i],L[min + i]
    return L
```

可以看到，在 selection_sort 中，我们在内部添加了 find_min()函数，该函数的作用是找出传入列表中最小值的索引。这样，在 selection_sort()函数的 for 循环中就可以调用这个函数，即每次循环都找出当前序列最小值的索引。

嵌套函数具体的作用是什么呢？为什么不把 find_min()函数写在 selection_sort()函数的外面，使得两个函数是并列关系？这就又一次涉及完美函数的思想，我们希望整个函数是一个黑盒子，拥有完整的功能，嵌套函数的主要作用就是希望内部函数只属于自己，且能完成完整功能。对于本例来说，selection_sort()函数只希望自己可以调用 find_min()函数，而不希望其他外界函数调用 find_min()函数，故将其写成嵌套函数的形式。

在有嵌套函数的前提下，应该如何分辨局部变量与全局变量？下面给出分辨规则。

有嵌套函数情况下分辨局部变量与全局变量的规则

假设有一个变量为 a，它出现在函数 f()中，应该怎样判断它在函数 f()内是什么变量？可以定义如下规则：

(1) 如果有 global 关键字修饰变量 a，那么不管函数 f()是不是嵌套函数，a 都为全局变量。

(2) 否则,假如 a 是参数或者出现在“=”左边,则 a 是局部变量。

(3) 否则,a 应继承上层函数中 a 的属性。如果函数 f()不是嵌套函数,那么 a 为全局变量。如果 f 是嵌套函数,a 就是上层的 a。

我们再分别举例讲解。先来看<程序:局部变量与全局变量举例 3>,它的最终打印结果是什么?

```
#<程序:局部变量与全局变量举例 3>
a = 1                              #全局变量
def F3():
    def F():
        global a                   #a 是最外层的全局变量
        print("In F3's F, a = ",a)
    a = 3
    F()
F3()
```

该程序的最终结果为:In F3's F, a= 1。由于 F()函数中的变量 a 被 global 关键字修饰(满足规则(1)),所以它是全局变量。

再看<程序:局部变量与全局变量举例 4>,结果又是什么?

```
#<程序:局部变量与全局变量举例 4>
a = 1
def F4():
    global a
        def F():
            a = 2                  #a 是 F 的局部变量
            print("In F4's F,a = ",a)
        F()
    print("In F4,a = ",a)          #a 是全局变量
F4()
```

上述程序最终的结果为:In F4's F,a=2;In F4,a=1。在 F 函数中 a 没有被 global 修饰,在“=”左边,故为局部变量(满足规则(2))。而在第二个 print 中,a 是由 global 修饰的全局变量,故值为 1。

再看<程序:局部变量与全局变量举例 5>,结果是什么?

```
#<程序:局部变量与全局变量举例 5>
a = 1
def F5():
    def F():                   #a 不是 F 的局部变量,那就继承上层函数中 a 的属性
        print("In F5's F,a = ",a)
    a = 3
    F()
F5()
```

该例中,最终打印的结果为:In F5's F,a=3。F()函数中的 print 语句中的 a 既不是全

局变量，也没有在“＝”左边，那么按照规则(3)判断，a 应该继承上层函数(F5 函数)的属性。在 F1 函数中，“a＝3”，a 在“＝”左边，是 F5 函数的局部变量，故调用 F()函数的时候 a 的值为 3。

3.1.5　参数类型

在前面介绍过的函数知识中，我们使用到的参数都算作是普通参数，除了普通参数以外，其实还有很多其他类型的参数。本节将具体介绍以下 3 种不同的参数类型：默认参数、关键参数和可变长度参数。大家在编写程序时可根据实际情况选择使用哪一类型的参数。

1. 默认参数

默认参数(Default-value Parameter)是在定义函数的时候就为参数设定一个默认的值，这样，在调用带有默认参数的函数时，可以不用为设置了默认值的参数进行传值，函数会自动使用默认值对该参数进行赋值。带有默认参数的函数定义语法如下：

```
def 函数名(… ,参数名 = 默认值):
```

当调用带有默认参数的函数时，既可以不对默认参数进行赋值，也可以在传参时通过自己赋值的方式来替换该参数的默认值。下面来看一个例子(见<程序：默认参数举例>)。

```
#<程序：默认参数举例>
def add(a,b,c = 3):
    res = a + b + c
    return res
print(add(1,2))                #打印结果为 6
print(add(1,2,5))              #打印结果为 8
```

上面的 add 函数中，“c＝3”即为默认参数，当第一次调用 add()函数时，只写了两个参数值，即只有参数 a 和 b 的值，那么在计算时参数 c 就会被赋值为默认值 3，所以 res＝1＋2＋3＝6。而第二次调用 add()函数的时候，传递了 3 个参数的值，那么默认参数 c 的默认值 3 就会被覆盖，实际赋值给 c 的值是 5，所以 res＝1＋2＋5＝8。

需要特别注意的是，在定义带有默认参数的函数时，默认参数只能出现在所有参数的最右端，并且任何一个默认参数的右侧都不能再定义非默认参数，否则会报错。

2. 关键字参数

例如，在之前学习的 print("Hello World",end＝" ")中，使用 end 参数时只需要通过参数名对它传递值即可，其实 end 就是 print()函数的关键字参数(Keyword Parameter)。关键字参数指的是在调用函数时的一种参数传递方式。通过关键字参数进行传参时，只需要按照参数的名字传递值即可，不需要关心定义函数时参数的顺序。来看下面的例子(见<程序：关键字参数举例 1 >)。

```
#<程序：关键字参数举例 1>
def my_print(a,b,c):
    print(a,b,c)
my_print(c = 4,a = 8,b = 3)                    #打印结果为 8 3 4
```

在上例中可以看到，虽然定义 my_print()函数时参数的顺序是 a、b、c，但是在调用 my_print()函数的时候，使用关键参数的方式传参，就可以不按照顺序来传，关键参数会根据参数的名字“对号入座”。所以调用 my_print()函数之后，还是可以正常地打印出 3 个参数的值。我们可以进一步思考一下，在使用关键参数这一方式时，“c＝4，a＝8，b＝3”中的变量 a、b、c 都是什么变量？可以做一个小试验，见<程序：关键字参数举例 2>。

```
#<程序：关键字参数举例 2>
a = 3                                          #全局变量
def add(a,b,c):
    a = a + b + c
add(c = 4,a = 8,b = 3)
print(a)                                       #打印结果为 3
```

在上面的举例中可以看出，其实在使用关键参数这一方式的时候，“c＝4，a＝8，b＝3”中的变量 a、b、c 都是局部变量，所以 add()函数的操作不会影响全局变量 a 的值，故执行 add()函数之后，a 的值仍为 3。但这种方式还是容易产生混淆，故建议大家将关键参数与外界其他变量区分开来，或尽量少用关键参数。

3. 可变长度参数

编程时可能会遇到这样一种情况：不确定参数的个数是多少。那么这时候就可以使用可变长度参数(Variable-length Parameter)。用 * parameter 的形式来表示可变长度参数，该参数会将接收来的任意多的参数存放在一个“元组”中(这里大家先将元组看成是个列表，只是它“不可变”，我们将在后面介绍元组这一数据类型)，则后续所有的对该参数的操作就是对元组的操作。我们通过<程序：可变长度参数举例>来具体解释一下。

```
#<程序：可变长度参数举例>
def fun( * p):
    num = p.count(0)              #count(a)函数为元组内置函数,统计元组中 a 的个数
    if num > 0:
        print("The number of 0 in the parameter is: %d" % num)
    else:
        print("There is no number 0 in the parameter.")
fun('a',"bcd",0,'n')              #输出结果为 The number of 0 in the parameter is:1
```

在该程序中可以看到，我们只用 * p 来接收传进来的所有参数，并将其保存为元组。fun()函数统计传递进来的参数中数字 0 的个数。

需要说明的是，若有必要给函数传入其他参数和可变长度参数时，仍需将可变长度参数放在所有参数的最右端，且可变长度参数后面不能再定义其他参数。

兰　兰：如果默认参数和可变参数一起使用，哪个放在最右边呢？

沙老师：大家可以试一下。我们定义一个函数：def f(a,b,*p,c=3):print(a,b,p,c)，当执行时，除了将值传给 a 和 b 外，其他参数全部传给 p，这样就无法改变默认参数 c 的值了。而当 def f(a,b,c=3,*p,)时，如果要对 p 参数赋值，就必须对 c 赋值，传入的参数就必定会改变默认参数 c 的值，那么参数 c 就不具有默认参数的意义了。这两种使用方式都使得默认参数 c 不再有意义。所以，同学们要注意，在 Python 中最好慎重将默认参数和可变参数一起使用。

3.2 再谈序列与字典数据类型

列表(List)、元组(Tuple)、字符串(String)这种能够通过下标索引到内部元素的类型统称为**序列**(Sequence)，而除了序列，Python 中还有字典(Dictionary)这种**映射**(Map)关系的数据类型，有了本书前面章节介绍的一些基础知识之后，本节将继续深入探讨这些数据类型。首先大家要有这样一个概念：列表和字典是可变的，即可以直接在原数据的基础上做修改；而元组和字符串是不可变的，即不能直接在原数据上做修改，只能重新创建新的数据。关于可变与不可变问题将在后面详细讲解。

3.2.1 列表与元组

前面简单介绍了一下列表(List)的通用操作，知道列表是以逗号相间隔的序列，列表中的每项称为元素(Element)，比如语句 L=[8,7,6,5]定义了一个含有 4 个元素的列表。列表是由有限个元素组合而成的一种有序集合，该结构为每个元素分配了一个序号(从 0 开始)，或称为索引(Index)。在 Python 中，将这种有索引编号的结构统称为“序列”，序列主要包括列表(List)、元组(Tuple)、字符串(String)等，本节将深入介绍列表以及元组。

首先介绍列表，列表中的元素类型可以是不同类型的，也可以合理地嵌套。也就是说，列表中的元素可以是整数型、浮点型、字符串，还可以是列表或其他数据类型。例如，L=[1,1.3,'2','China',['I','am','another','list']]，将不同元素类型融合到一个列表中。由于列表中的元素可以是不同类型，那么需要提醒读者的是，在对列表元素做运算时一定要注意元素的类型，否则会出现错误。如上述的 L，L[0]+L[2]操作将产生错误，因为整数型不能与字符串相加。

对列表有了初步了解后，本节将从以下两个方面对列表进行介绍：

(1) 分片操作。

(2) 列表的专有方法。

最后介绍与列表很相似的一种序列：元组。

1. 分片操作

列表的通用操作在第 1 章已经基本介绍过了，相信读者已经学会了索引的使用、列表的增删改等操作，此处不再赘述，下面重点强调分片。

Python 为序列提供了强大的分片操作，我们将以列表为例系统地讲解一下分片操作。再次提醒读者，下面所讲的所有列表的分片操作对其他序列来说都是通用的。分片操作的运算符仍然为下标运算符，即“[]”，而分片内容通过冒号相隔的两个索引来实现。假设一个列表为 L，则分片的格式为：L[index1:index2:stride]。index1 是起始分片的索引号，而 index2 是结束分片的索引号且不包含 index2 处的值，也就是说，只有在 L[index1]～L[index2－1]的元素才会出现在分片结果中；stride 表示以规定的步长取数据，即先取索引为 index1 的元素，再取索引为 index1＋ stride 的元素，再取索引为 index1＋ 2 * stride 的元素，一直到取得的最后一个元素的索引刚好没有越过 index2。

例如，L＝[1,1.3,'2','China',['I','am','another','list']]，如果只希望获得 L 中的 3 个元素："2"、"China"和['I','am','another','list']，则 L[2:5:1]即可实现(即从 L[2]开始分片，一直到 L[4]结束，步长为 1)，在没有指定步长的情况下，步长默认值为 1，所以 L[2:5:1]还可以简写为 L[2:5]。

还有一些特殊情况如下：如果分片为 L[index1:index2]且 index2≤index1，那么分片结果将为空。如果 index2 置空，即 L[index1:]，分片结果将包括索引为 index1 及之后的所有元素。index1 也可以置空，即 L[:index2]，这个分片表示从列表开头 0～index2－1 的分片结果。而当 index1 与 index2 都置空时，将复制整个列表，如 L[:](注意：这是复制列表的一个很常用的方式)。如果步长大于 1，那么就会跳过某些元素，例如，要得到 L 的偶数位的元素，就要取索引为 0、2、4、6 的元素，那么步长应该设置为 2，而且我们是要取得整个列表的偶数位元素，index1 和 index2 都置空即表示整个列表，所以用语句 L[::2]即可实现，如下面的代码所示。

```
#<程序：得到列表 L 偶数位的元素>
L=[1,2,3,4,5,6,7,8]
L[::2]                    #输出[1, 3, 5, 7]
```

那么列表的逆序操作该如何实现呢？首先试一下分片语句是 L[len(L)－1:－1:－1]，输入并执行这条语句，神奇的情况发生了，如下面代码所示。

```
#<程序：得到列表的逆序 1>
L=[1,2,3,4,5,6,7,8]
L[7:-1:-1]                #输出[]
```

为什么执行该语句返回的是空列表呢？原因是，Python 是支持负索引的，比如－1 是倒数第一个元素的负索引，－2 是倒数第二个元素的负索引，以此类推，－len(L)是第一个元素的索引。那么看一下刚才的语句 L[7:－1:－1]，index1 为 7，是倒数第一个元素的索引；index2 是－1，也是倒数第一个元素的索引。所以 index1 和 index2 之间是没有元素的，我们只能得到一个空列表！那么到底该如何取得列表的倒序呢？答案就是，全用负索引表示：

因为要倒序取元素，所以步长设置为－1，我们想要得到元素的负索引依次为－1、－2、－3、－4、－5、－6、－7、－8，那么 index1 应该为－1，而 index2 是分片中最后一个元素的下一个元素的索引，即 index2 应该为－8－1＝－9，所以分片语句是 L[－1:－9:－1]，大家试试看。

还有一个更为简便的分片语句可以实现列表倒序：L＝[::－1]，这是因为当 stride 为负数时，如果 index1 为空，那么 index1 默认为－1；如果 index2 为空，那么 index2 默认为－len(L)－1，所以语句 L＝[::－1]其实就是 L＝[－1:－len(L)－1:－1]，只不过省略了 index1 和 index2，其实是同样的分片。

注意，分片是一种复制。也就是说，分片操作是复制原来列表中的某些内容来产生一个新的列表。例如，列表的分片操作 L1＝ L[:]，是个常用操作，将复制列表 L 的内容，构建一个新的列表值，然后把这个新的列表值和 L1 关联起来。这样 L1 就是 L 的一个副本，那么 L1 和 L 之间不会相互影响(注意，如果 L 是多层列表，则有可能还是会相互影响，以后再讨论此种情形)。如果不使用分片，直接通过“L1＝L”将 L 赋值给 L1，则 L 和 L1 指向同一个列表，是一定会互相影响的。代码如<程序：元素是整数型的列表的复制>所示。

兰　兰：分片操作和 range 容易混淆，我们应该如何区别它们呢？

沙老师：我们只要知道分片和 range 的差异：

(1) 分片中使用[]；range 中使用()。

(2) 进行分片操作时[]内至少要有 1 个冒号，即“:”，使用 range 时可以省略()内的逗号。

(3) 对列表进行分片时索引－1 有特殊含义，而 range 中的索引－1 没有。有了这个基本认识，range 从 n～0 迭代，range 的写法为 range(n,－1,－1)，而用分片的方法表示为 L[::－1]或 L[－1:－len(L)－1:－1]。其实我们很少使用分片操作来倒置列表，而利用 range 的倒置更常见。

```
#<程序：元素是整数型的列表的复制>
L = [5,4,3,2,1]
L1 = L
L[4] = 0
print(L)              #输出[5, 4, 3, 2, 0]
print(L1)             #输出[5, 4, 3, 2, 0]
```

从程序的执行结果来看，当通过“＝”直接使得 L1 引用原来的列表 L 时，如果对 L 中的元素进行修改，那么 L1 中的元素也会随之发生改变。

下面介绍使用分片时不会相互影响的情况，代码如<程序：元素是整数型的列表的分片>所示。

```
#<程序：元素是整数型的列表的分片>
L = [5,4,3,2,1]
L1 = L[:]
L[4] = 0
print(L)              #输出[5, 4, 3, 2, 0]
print(L1)             #输出[5, 4, 3, 2, 1]
```

2. 列表的专有方法

Python 实现了序列的一些通用操作函数，如表 3-1 所示，表 3-1 中的方法对于列表、字符串、元组都是通用的，但列表还提供了额外的很多方法(method)，这里所说的方法事实上与函数是一个概念，但它们都专属于列表，其他的序列类型是无法使用这些方法的。

这些专有方法的调用方式(见表 3-2)也与如表 3-1 所示的通用序列函数调用方式不同。如果要统计列表 L 的长度，那么使用表 3-1 中的 len()函数，其调用语句为 len(L)，这个函数调用意味着要将 L 作为参数传递给 len()函数。

表 3-1　通用序列函数

序号	函　　数	说　　明
1	len(seq)	返回序列 seq 的元素个数
2	min(seq)	返回序列中的“最小值”
3	max(seq)	返回序列中的“最大值”
4	sum(seq)	序列求和(注：字符串类型不适用)

但是，当使用列表的专用方法时，方法的调用形式是 L. method(parameter)，其中 parameter 不包含 L，在调用这些专用方法时，并不会显式地传递 L。另外需要注意的是，这里使用了“.”操作符，该操作符意味着要调用的方法是列表 L 的方法。表 3-2 给出了列表的专有方法，操作的初始列表为 s＝[1,2]，参数中的[]符号表示该参数可以传递也可以不传递。

表 3-2　列表的专有方法

序号	函　　数	作用/返回	参数	L 结果/返回
1	s. append(x)	将一个数据添加到列表 s 的末尾	'3'	[1,2,'3']/none
2	s. clear()	删除列表 s 的所有元素	无	[]/none
3	s. copy()	返回与 s 内容一样的列表	无	[1,2]/[1,2]
4	s. extend(t)	将列表 t 添加到列表 s 的末尾	['3','4']	[1,2,'3','4']/none
5	s. insert(i, x)	将数据 x 插入 s 的第 i 号位置	0,'3'	['3',1,2] /none
6	s. pop(i)	将 s 第 i 个元素(默认最后)弹出并返回其值	1 或无	[1]/2
7	s. remove(x)	删除列表 s 中第一个值为 x 的元素	1	[2] /none
8	s. reverse()	反转 s 中的所有元素	无	[2,1] /none
9	s. sort()	将 s 中的所有元素按升序排列	无	[1,2]/none

值得关注的一点是，使用 L1＝L. copy()和 L1＝L[:]时都会复制出新的列表，但这两种方法都是所谓的**顶层复制**，也就是只有复制了列表的第一层，当 L 是多层列表时，则所嵌套的列表并没有真正地被复制，它们还是被共享的。所以虽然 L1. append()不会影响到 L，但是 L1[0]. append()竟然会影响到 L[0]。大家可以试试看。为了达到所有深层次的复制，可以使用 deepcopy 方法。注意，使用 deepcopy 方法时要引入 copy 库。可以看一下<程序：copy 和 deepcopy 的使用举例>。

```
#<程序：copy 和 deepcopy 的使用举例>
import copy
```

```
L = [[0,0],[1,1]]
L1 = L[:]; L2 = L.copy()
L3 = copy.deepcopy(L)
L[0][0] = 100
print("L1 = ",L1)
print("L2 = ",L2)
print("L3 = ",L3)
```

这个程序的输出结果为：L1＝[[100,0],[1,1]]；L2＝[[100,0],[1,1]]；L3＝[[0,0],[1,1]]。

兰　兰：我们需要记住这些专有方法吗？

沙老师：完全不需要，对于列表的专有方法而言，append 比较常用，pop 有时会使用，其他都很少用到。大家要尽量自己编写程序来完成需要的功能，这样既可以实现想要的功能，又锻炼了编程能力。举例而言，如果我想要将列表 L 排序后产生一个新列表，而不改动列表 L，这样就不能使用 L.sort()方法，而需要自己去实现。

3. 列表的“近亲”：元组

说元组(Tuple)和列表是“近亲”，是因为 Python 中元组与列表极其相似，不同之处在于元组的元素和长度是不可变的(immutable)，即不能在原来数据基础上做修改；要修改的话，只能产生新的元组。元组使用小括号，而列表使用中括号。元组创建很简单，只需要在括号中添加元素，并使用逗号隔开即可。

例如，创建一个空元组：tup1 ＝ ()；创建非空元组：tup2 ＝ ('hello', 'world', 2017, 2000)。

如果你觉得可以用“[]”和“()”来区别列表和元组，那么你就错了！其实对于元组来说，“,”才可以真正标识元组。可以来试验一下：若元组中只包含一个元素，则需要按照如下的方式写，即在元素后面一定要添加逗号，这样在调用 type(tup1)的时候得到的结果才是<class 'tuple'>。代码如<程序：元组举例>所示。

```
#<程序：元组举例>
tup1 = (50,)
type(tup1)              #输出：<class 'tuple'>
```

元组与列表类似，索引还是用“[index]”表示，下标索引从 0 开始，可以进行分片等序列的基本操作，例如，tup2 ＝ (1, 2, 3, 4, 5, 6, 7)，tup3＝tup2[1:5]，经此分片操作后复制得到元组 tup3＝(2, 3, 4, 5)。

元组有以下内置函数：

(1) len(tuple)——计算元组元素个数。

(2) max(tuple)——返回元组中元素的最大值。

(3) min(tuple)——返回元组中元素的最小值。

(4) tuple(seq)——有时可能需要将列表转化为元组，Python 为此提供了函数 tuple

(seq),它将把列表转换为元组。

元组并没有实现像列表中 append()这类可以直接修改元组的内置函数。注意,元组中的元素值是不允许修改的,但可以对元组进行连接组合,以创建新的元组。例如,“tup1=('hello', 'world', 2017, 2000); tup2=(12,),tup3 = tup1 + tup2”,将产生新的序列 tup3=('hello', 'world',2017, 2000,12)。同时,元组的元素也是不允许删除的,但可以使用 del 语句来删除整个元组,例如,“tup1 = (12, 34); del tup1”这个语句会删除整个元组 tup1。

那么什么时候用元组,什么时候用列表呢?一般来说,声明序列之后元素不会变动的时候用元组,需要变动的时候用列表。例如,可变长度参数序列是用元组表示的,因为一旦创建后就不应该被改变,而举例而言,若要存储每个课程与其对应的学时数,那么由于课程是会随时增加或删除的,所以整体可以用列表来存储,而列表中的每个元素,则以(课程名称,学时数)的元组形式存在,因为每个课程的学时数是已经确定好、不会再更改的。元组的好处就是,若程序执行时不小心要去修改不应该被改变的元组变量,就会报错,这样程序员就会意识到代码出错了,所以元组类型的存在也是有意义的。

3.2.2 字符串

字符串(String)也是编程时常用的一个数据类型,生活中很多信息都是以字符串的形式存在的,例如,我们的名字、Internet 网址、文件中的内容,等等。本节将重点讲述以下 3 点内容:字符串的基本操作;字符串的格式化;字符串的专有方法。

1. 字符串的基本操作

1) 引号的使用

在第 1 章中我们曾经提到过字符串的引号可以是单引号也可以是双引号。但是在某些特殊场景下区分单双引号的使用还是十分必要的,比如英文中的简写、字符串中包含对话等情况。我们看几个例子就明白了。首先来看<程序:引号的使用举例 1>。

```
#<程序:引号的使用举例 1>
S = "Let's play!"
print(S)                #输出: "Let's play!"
SS = 'She told that:"I love China!".'
print(SS)               #输出: 'She told that:"I love China!".'
```

如果字符串中同时使用了单引号和双引号,例如,字符串 S 为“She said :"Let's play!"”。那在这个字符串的外面,单引号和双引号都不可用,我们可以使用反斜线方式转义其为普通字符。如“S='She said :"Let\'s play!"'”。

此外,Python 还专门为多行文字编写提供了一种格式,就是使用三重双引号将多行文字括起来,这样我们在输入的时候就显得很方便,见<程序:引号的使用举例 2>。

2) 字符串的增删改

我们已经知道可以用 s=''的方式来表示一个空的字符串,也知道可以直接用'abc'+'def '的方式来得到连接后的字符串'abcdef '。那么当需要修改字符串中的内容时应该怎么

做呢？

```
#<程序：引号的使用举例 2>
S = """ My name is Lily,
I live in China now,
I love China."""
print(S)                #输出：'My name is Lily,\nI live in China now,\nI love China.'
                        #"\n"表示换行
```

但同时字符串也如同元组一般，是不可变的变量，即它里面的元素不可以直接改变，所以需要通过序列分片的方式将不同的字符串连接起来，从而形成所需要的新的字符串。例如，想把字符串 s='abcd'的第一个字符改成'o'，就不能通过 s[0]='o'来改变(大家可以试一下 Python 是否会报错)，我们只能重新创建新的字符串。那么这就使用到了字符串的分片操作，可以通过 s_new='o'+s[1:]得到字符串 s_new='obcd'。注意，此时 s 的值不会改变，仍旧是'abcd'，若想使 s 的值变为'obcd'，则可以令 s='o'+s[1:]，原来的'abcd'假如没有任何变量指向它，则 Python 会将其视为垃圾而回收。

下面通过字符串的分片操作具体介绍如何对字符串中的字符做修改、增加以及删除操作。假设 S='abcde'，将字符串 S 中'b'改为'z'，可以这样做：通过分片操作将'a'、'z'、'cde'连接在一起，执行 A= S[:1] +'z'+ S[2:]。

对于 S= 'abcde'，如果需要做字符增加操作，例如，在'b'后增加'z'，得到字符串'abzcde'，通过分片操作实现，A= S[:2] +'z'+ S[2:]。

同样，做删除操作时不能直接删除一个字符串中的某个字符，因为字符串是不可变的，所以，只能提取出字符串中我们需要的部分，再把这些部分重新组合起来形成一个新串。例如，想要从'abcde'中删除'b'，则执行 A=S[:1] + S[2:]。

2. 字符串的格式化

编程中常常需要将数据处理成统一的格式，这个时候就需要将字符串进行格式化处理，返回值同样是字符串。这既可以使得编写代码变得很方便，也可以在输出给用户的时候显得很美观。举个小例子，假设有两个列表 name=['Tom','John','Bob','Jake','Paul']和 age=[18,20,17,21,23]，分别存储全班同学名字和对应的年龄，为了清楚地查看每位同学的年龄，需要得到像“Tom is 18 years old”这样的输出，可以看到这种输出格式是以字符串"X is Y years old"作为模板的，其中 X 对应 name 列表中的某个值，它是字符串类型的，Y 对应 age 列表中的某个值，是整数类型的。使数据以一个统一的模板输出的操作，称为**格式化**。

格式化操作时，Python 使用一个字符串作为模板，模板中有格式符，这些格式符为真实值预留位置，并说明真实数值应该呈现的格式。首先看一个简单的例子，代码如<程序：格式化输出举例 1>所示。

```
#<程序：格式化输出举例 1>
s = '%s is %d years old'%('Tom',18)
print(s)
```

这个例子的字符串 s 被设定为"Tom is 18 years old"。其中'%s is %d years old'为模板。在模板和待填充数据之间,有一个%分隔,它代表了格式化操作。%s 为模板中的第一个格式符,表示一个字符串;%d 为第二个格式符,表示一个整数。%s 和%d 会分别替换为元组('Tom',18)中的两个元素'Tom'和 18。在了解了这些之后,再结合前面所学的知识,可以写一个函数清楚地查看例子中班上每位同学的年龄,代码见<程序:格式化输出举例 2>。

```
#<程序:格式化输出举例 2>
def my_result(L,A):
    for i in range (0,len(L)):
        print('%s is %d years old'%(L[i],A[i]))
    return
name = ['Tom','John','Bob','Jake','Paul']
age = [18,20,17,21,23]
my_result(name,age)
```

如此,就可以得到格式化的输出,输出结果如下:

```
Tom is 18 years old
John is 20 years old
Bob is 17 years old
Jake is 21 years old
Paul is 23 years old
```

除了上面例子中使用的格式符%s、%d,Python 还提供了很多其他的格式符,比如%f 表示浮点数、%b 表示二进制整数、%d 表示十进制整数,等等。

同学可能会问:假如有个变量 x,这个变量可能是浮点数或者是整数,有没有比较简单的方式可以格式化输出 x 呢?

按照 Python 原来格式化的方式,如果对整数使用%f,产生的字符串会有小数部分。所以一定要用%d 才会避免小数部分。因此,我们就必须在格式化之前判断 type(x)是整数还是浮点数。但是,Python 还有个新的格式化方式 format()来按照 x 的类型输出 x。

从 Python 2.6 开始,新增了一种格式化字符串的函数 str.format(),在 format()函数中,使用"{}"符号来当作格式化操作符,这个函数的功能十分强大,不需要指定待格式化元素的类型。下面简单介绍一下关于 format()函数的常用操作。

(1) format()函数通过参数的位置格式化字符串:"{}"中的数字为参数的位置,位置可以不按顺序,字符串的 format()函数可以接收多个参数,下面通过例子来说明它的用法。比如输入'{0},{1},{0}'.format('ab',123)。注意:因为 format 是字符串的专有方法,所以大家一定不要忘记在字符串之后和 format 之前加上"."。其中输出的格式'{0},{1},{0}'分别表示第 0 个参数(即'ab',计算机中类似索引这种计数方式都是从 0 开始的)和第 1 个参数(即 123),第 0 个参数(即'ab'),所以它的输出为'ab,123,ab'。当输入'{},{}'.format('ab',123)时,输出为'ab,123',因为{}中没有参数位置时,所以默认按参数顺序输出。

练习题 3.2.1 请大家使用 format 改写<程序:格式化输出举例 2>。

【答案】 将循环体中的 print 改为如下:

```
print('{0} is {1} years old'.format(L[i],A[i]))
```

(2) format 函数填充字符串：在 format 函数中，填充常跟对齐一起使用，例如，语句'{:s^8}'.format(123)，这个语句是什么意思呢？“:”号后面是待填充的字符，只能是一个字符，对于这个例子填充字符为's'，不指定的话默认是用空格填充；^、<、>分别表示居中、左对齐、右对齐，本例中对齐方式是居中；后面再写输出的字符串长度，这个例子的字符串长度为 8，所以会输出一个长度为 8，数字 123 居中，空位用 s 填充的字符串，即'ss123sss'。

(3) format 函数格式化数字：在 format 函数中，精度(精度可以认为是保留小数点后几位的精确度)常跟浮点类型 f 一起使用。比如：'{:.2f}'.format(321.33545)，其中.2 表示精度为 2，f 表示 float 类型，所以输出为'321.34'。当然，如果用前面学习的格式符的方法也可以表示为%.2f。另外还有很多特殊的格式化输出，比如在涉及财务问题的时候，经常需要用千位分隔符将较大的金额数字隔开，以方便查看金额，format 函数就提供了用“,”来做金额的千位分隔符，当输入'{:,}'.format(1234567890)时，可以得到输出为'1,234,567,890'，是不是十分方便？

3. 字符串的专有方法

与列表类似，字符串也提供了很多专有方法(Method)，表 3-3 给出了字符串的常用的 10 种方法并给出了相应的范例。以 str="HEllO"作为例子，表 3-3 给出了相关操作后的输出结果。参数中的[ ]表示调用方法时，该参数可以传递也可以省略。比如 str.count('O')与 str.count('O',2)，以及 str.count('O',2,4)的语法都是正确的，但是第一个调用表示统计整个字符串中的'O'，第二个调用表示统计从 2 号索引开始到结束出现'O'的次数，而第三个调用表示统计 str 中索引为 2 和 3 的位置'O'出现的次数。

表 3-3　字符串的专有方法

序号	函　　数	作用/返回	参数	print 结果
1	str.capitalize()	首字母大写、其他小写的字符串	无	"Hello"
2	str.count(sub[, start[, end]])	统计 sub 字符串出现的次数	'O'	1
3	str.isalnum()	判断是否是字母或数字	无	True
4	str.isalpha()	判断是否全部是字母	无	True
5	str.isdigit()	判断是否全部是数字	无	False
6	str.strip([chars])	开头结尾不包含 chars 中的字符	'HEO'	'll'
7	str.split([sep], [maxsplit])	以 sep 为分隔符分割字符串	'll'	['HE','O']
8	str.upper()	返回字符均为大写的 str	无	"HELLO"
9	str.find(sub[, start[, end]])	查找 sub 第一次出现的位置	'll'	2
10	str.replace(old, new[, count])	在 str 中，用 new 替换 old	'l','L'	"HELLO"

再次提醒注意，上述 str 的专有方法并不改变 str 字符串的内容。如果希望 str 变为返回的字符串，可以用 str=str.method(…)语句将返回的字符串赋值给 str。例如，str='abcd'，需要把将字符串 str 中的'cd'变为'ef'，我们可以通过调用 replace()函数实现，即 str.replace('cd','ef')，如图 3-3(a)所示，函数会返回字符串'abef'，注意此时原字符串 str 没有改

变还是'abcd',如果希望'abef'变为返回的字符串,可以执行 str=str.replace('cd','ef'),使 str 指向返回的字符串'abef',如图 3-3(b)所示,此时 str='abef'。

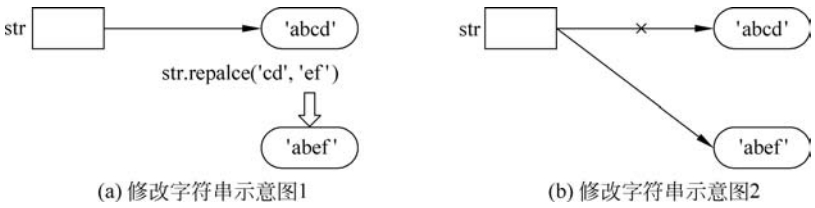


图 3-3　修改字符串示意图

查看元素是否在序列中的方法

我们来总结一下"查看一个元素是否在序列中"都有什么方法?假设元素为 e,序列为 L。其中(1)、(2)方法对任何序列都是通用的,而(3)、(4)只能用于字符串序列。

(1) 最直接的方法:for 循环遍历 L 查找。for i in L: if i==e: print("find it!")。

(2) 用关键字 in:if e in L: print("find it!")。

(3) 若 L 为字符串,可以用 find 函数:if L.find(e)>-1: print("find it!")。注意:find 函数还可以返回所找元素第一次出现的位置。

(4) 若 L 为字符串,可以用 count 函数:if L.count(e)>0: print("find it!")。

兰　兰:字符串和元组都是不可改变的数据类型,它们的关系是什么?

沙老师:字符串可以看作一种特殊的元组。元组可以是多层次的,元组内的元素可以是任意的数据类型,而字符串中的每个元素只能是字符。

3.2.3　字典

字符串、列表、元组都是序列,而 Python 的基本数据结构除了序列外,还包括映射(Mapping),字典用于存放(键,值),英文是(key,value)这样的映射关系的数据结构。

回忆一下高中所学的函数概念。定义:设 X、Y 是两个非空集合,如果存在一个法则 f,使得对 X 中每个元素 x,按法则 f,在 Y 中有唯一确定的元素 y 与之对应,则称 f 为 X 到 Y 的映射,记作:f: X→Y。集合 X 为 f 的定义域(Domain),集合 Y 为 f 的值域(Range),要注意的是对映射 f,每个 x∈X,有唯一确定的 y=f(x)与之对应,也就是说,映射可以是一对一映射,也可以是多对一映射,但不能是一对多,如图 3-4 所示。

字典(Dictionary)的形式为{ }。Python 中既可以创建空字典,也可以直接创建带有元素的字典。字典中的每个元素都是一个键值对(Key:Value),而键 Key 在字典中只会出现一次,也就同大家知道函数一样是不可以有一对多的映射关系的。键是集合 X 中的一个元素,而 Value 指的是集合 Y 中的一个元素,它们的关系为 f(key)=value。比如要存放"Hello"中每个字符出现的频次数,mdict = {'H':1, 'e':1, 'l':2, 'o':1},这个例子中 X 是集合{'H', 'e', 'l', 'o'},Y 是集合{1,2},mdict['H']=1,mdict['l']=2,…可以注意到,字典并不能像列表、字符串一样通过下标写 0、1、2、…的方式来索引元素,而是通过索引关键

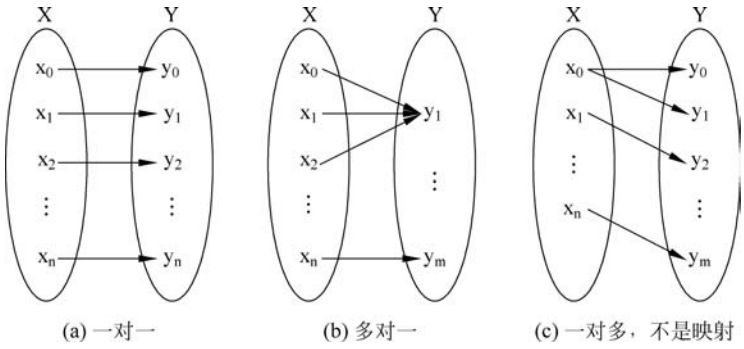


图 3-4　映射的类型

字的方式来得到其对应的值。同时字典也可以嵌套字典或列表。下面举几个获取字典中元素的例子，代码详见<程序：字典获取元素>。

```
#<程序：字典获取元素>
d_info1 = {'XiaoMing':[ 'stu','606866'],'AZhen':[ 'TA','609980']}
print(d_info1['XiaoMing'])              #['stu', '606866'] 为打印结果,下同
print(d_info1['XiaoMing'][1])           #606866

d_info2 = {'XiaoMing':{ 'role': 'stu','phone':'606866'},
'AZhen':{ 'role': 'TA','phone':'609980'}}
print(d_info2['XiaoMing'])              #{'role': 'stu', 'phone': '606866'}
print(d_info2['XiaoMing']['phone'])     #606866
```

兰　兰：字典只能是一对一或多对一的映射，那么如果我的 mdict 是如下形式：mdict = {'H':1, 'e':1, 'l':2, 'o':1, 'H':2}会不会出错呢？

沙老师：不会的，当后出现的键值对的 Key 已经出现过，那么将会覆盖原来键值对中该键所对应的值，所以实际上 mdict={'H':2, 'e':1, 'l':2, 'o':1}。

Python 中提供字典这个映射类型，使得 Python 对数据的组织和使用更加灵活。Python 字典是符合数据库数据表格的概念，它能够表示基于关系模型的数据库，关系模型中最基本的概念是关系(Relation)。表 3-4 给出的字符频次表就是一个关系。关系中的每一行(Row)称为一个记录；每一列(Column)称为一个属性。在每个关系结构中，必须要有键(Key)作为寻找记录的依据。所以必须有某一个属性或者属性组的值在这个关系表中是唯一的。这个属性或属性组称为该关系的键(Key)。例如，在如表 3-4 所示的关系中共有 3 个属性：字符、频次、频率。可以看到，字符属性是没有重复的，所以可以用“字符”这一属性来当作键。

表 3-4　字符出现频次表

字　　符	频　　次	频　　率
H	1	0.2
e	1	0.2
l	2	0.4
o	1	0.2

字典中的键值对用 f(x)=y 来表示关系，在 Python 字典中是可以很灵活地定义 x 和 y 的结构。然而 x 必须是不可变的类型，例如，元组、字符串、数值等，x 不可以是列表类型，但 y 可以是任意类型，如列表或字典。所以，当关系中的键 x 是由多个属性组成时，在 Python 中可以用元组的方式来表示 x。当对于属性 y 有多个值时，Python 中可以用列表或字典的形式来表示 y。

表 3-4 中的关系可使用 Python 中的字典类型进行存放，如：mdict = {'H':[1,0.2], 'e':[1,0.2], 'l':[2,0.4], 'o':[1,0.2]}，这时，mdict['H'][1]即为字母'H'出现的频率。对于该关系，Python 还有另一种表达形式，即 f(x)=y 中的 y 还可以是字典类型，如：mdict2 = {'H':{'count':1,'freq':0.2}, 'e':{'count':1,'freq':0.2}, 'l':{'count':2,'freq':0.4}, 'o':{'count':1,'freq':0.2}}，这时，mdict2['H']['freq']表示字母'H'出现的频率。第一种方式，要获取一个记录的某个属性，需要知道该属性在记录中的索引顺序；而第二种方式，要获取一个记录的某个属性，需要给出属性名。

与序列一样，映射也有内置操作符与内置函数，最常用的内置操作符仍然是[ ]，如 mdict['H']，将返回'H'所对应的 value，即 1。操作符[]也可以作为字典赋值使用。例如，mdict['H']=1，假如 mdict 里面没有'H'，就会将'H':1 加入 mdict 里面，假如有'H'这个键，其值就被更改为 1 了。另外，in 与 not in 在字典中仍然适用，例如，'o' in mdict 将返回 True，而'z' in mdict 将返回 False；常用的函数是 len(dict)，它将返回字典中键值对的个数，例如，len(mdict)将返回 4。

字典类型也提供了很多专用方法，表 3-5 列出了字典常用的 10 种方法，以 mdict = {'H':1, 'e':2}为例。

表 3-5　字典常用的方法

序号	函　　数	作用/返回	参数	print 结果
1	mdict.clear()	清空 mdict 的键值对	无	{}
2	mdict.copy()	得到字典 mdict 的一个副本	无	{'H':1, 'e':2}
3	mdict.has_key(key)	判断 key 是否在 mdict 中	H/h	True/False
4	mdict.items()	得到全部键值对的 list	无	[('H',1),('e',2)]
5	mdict.keys()	得到全部键的 list	无	['H','e']
6	mdict.update([b])	以 b 字典更新 a 字典	{'H':3}	{'H':3,'e':2}
7	mdict.values()	得到全部值的 list	无	[1,2]
8	mdict.get(k[, x])	若 mdict[k]存在则返回对应值，否则返回 x	'o',0	0
9	mdict.setdefault(k[, x])	若 mdict[k]不存在，则添加 k:x	'x',3	{'H':1,'e':2,'x':3}
10	mdict.pop(k[, x])	若 mdict[k]存在，则删除	H	{'e':2}

下面通过一个例子来看看如何利用字典这一数据结构来解决实际问题。

【问题描述】 统计给定字符串 mstr="Hello world, I am using Python to program." 中各个字符出现的次数。

【解题思路】 要完成这项任务，要对字符串的每个字符进行遍历，将该字符作为键插入字典，或更新其出现的次数。其实现如<程序：统计字符串中各字符出现次数>所示。

```
#<程序：统计字符串中各字符出现次数>
mstr = "Hello world, I am using Python to program."
mlist = list(mstr)                    #将字符串转换成列表
mdict = {}
for e in mlist:
    if mdict.get(e, -1) == -1:        #还没出现过,也可以写作 if e not in mdict:
        mdict[e] = 1
    else:                             #出现过
        mdict[e] += 1
for key,value in mdict.items():
    print (key,value)
```

接下来，给出一些对字典做修改的例子，代码见<程序：对字典的修改>。

```
#<程序：对字典的修改>
#代码 1
di = {'fruit':['apple','banana']}
di['fruit'].append('orange')
print(di)                             #{'fruit': ['apple', 'banana', 'orange']}
#代码 2
D = {'name':'Python','price':40}
D['price'] = 70
print(D)                              #{'name': 'Python', 'price': 70}
del D['price']
print(D)                              #{'name': 'Python'}
#代码 3
D = {'name':'Python','price':40}
print(D.pop('price'))
print(D)                              #{'name': 'Python'}
#代码 4
D = {'name':'Python','price':40}
D1 = {'author':'Dr.Li'}
D.update(D1)
print(D)                              #{'name': 'Python', 'price': 40, 'author': 'Dr.Li'}
```

从上面的代码可以看出，与列表相同，字典也是可变的。除了在字典中添加元素外，还可以修改、删除字典中某个键对应的值，但需要注意的是，字典的 update 方法并不是单纯地更新某一个键对应的值，而是合并两个字典中所有不同的键。有的同学可能会问：“前面所讲的关系表格，每行是一个记录，我也可以用一个列表将每行当作一个子列表组织起来。那么对比使用列表结构或者使用字典结构这两种方法，哪一种更好呢?”为了方便大家理解，我们先给出列表与字典结构的差异，然后通过两个小测试来具体比对一下。

谈谈列表和字典结构的差异

(1) 列表是序列，字典不是序列。

(2) 序列是使用索引方式获取元素的，而字典是使用键来获取元素的。序列元素的插入是和索引相关的，而字典元素的插入是和键相关的。因此列表有 append()函数、分片功能等，而字典没有。

(3) 列表是一种通用的数据结构，它里面的元素可以千变万化。列表可以被视为有序

列的一组内容,可见其功能的广泛;而字典则不然,它是键值这种结构的独特组合。所以,字典对于寻找某一个键的记录,会有快速的方式来实现,这种实现方式叫作哈希(Hash)方式,有兴趣的读者可以自行了解。

下面来看两个测试程序(见<程序:使用字典查找元素的时间>和<程序:使用列表查找元素的时间>)。

```
#<程序:使用字典查找元素的时间>
import time
k = 50000
D = {}
for i in range(k, -1, -1): D[i] = i
start = time.clock()
for i in range(k):
    if k + 1 in D: print("Something wrong");break
elapsed = time.clock() - start
print("使用字典用时:",elapsed)
```

```
#<程序:使用列表查找元素的时间>
#import time
#k = 50000              #与前面程序放在一起
L = []
for i in range(k, -1, -1):
    L.append([i,i])
start = time.clock()
for i in range(k):
    if [k + 1,k + 1] in L: print("Something wrong");break
elapsed = time.clock() - start
print("使用列表用时:",elapsed)
```

两个程序的输出结果分别如下:

使用字典用时:0.005061647999999974。

使用列表用时:63.124664228。

虽然每台计算机所用的输出时间不同,但是可见字典的搜寻要比列表的 in 快速许多。此程序是将 50 000～0 的 50 001 个元素分别放入字典和列表中,然后遍历其中的所有元素,判断是否有 50 001 在这些元素中。程序利用了 time 库所提供的 clock()函数来进行计时,首先对这部分相关知识做一个讲解。

(1) import time。import 是 Python 中用来实现模块引用的语句,这里引入了 time 模块,所以在程序中就可以使用 time 模块中与时间有关的函数。

(2) time.clock()。它是 time 模块中特有的函数,用来返回程序运行时的实际时间。

程序中为什么要两次调用该函数呢?因为第一次调用时是在循环开始之前,则记录的就是开始的时间;第二次调用时是在循环之后,则记录的就是结束时间,两个时间之差才是我们所需要的循环执行时间。注意,如果使用 Python 3.8 版本或以后,time.clock()函数不再使用,用 time.perf_counter()函数来取代。

通过比较程序,可以发现使用字典的方式要比列表所花费的时间少很多,原因就是在字典中查找元素使用的是哈希方式,会更加快速,而列表使用的则是从头遍历列表的方式。

3.3 关于 Python 数据类型的注意事项

前面详细讲解了列表、字符串、字典、元组等数据类型，但在 Python 中，对这些数据结构的使用还有很多需要特别注意的地方，稍有不慎就会使得整个代码出错，并且可能是一些平时并没有在意的细节引起的。下面将分成可变与不可变类型和参数的传递问题这两点具体来讲。

3.3.1 可变与不可变类型的讨论

在讲解数据类型时，我们提到过可变(mutable)与不可变(immutable)这两个词。其中列表、字典是可变的，字符串、元组、数值是不可变的。可变是说，可以直接对该变量本身进行修改；不可变是说，不能直接对该变量本身进行修改，若需要改变该变量时，就只能重新分配一段空间存放新的值。本节重点以列表和字符串来举例，讲解可变与不可变类型。

首先需要了解 Python 中数据的存储方式。当写下 L=[1,2]这一语句的时候，其实 Python 开辟了两块空间，真实的数据[1,2]存放在其中一块空间，而变量 L 在另一块空间，可以把 L 看作一个装东西的容器，该容器(变量 L)只是保存了存放数据[1,2]的地址(也称作 L 中存放了指向存放数据[1,2]空间的指针)，如图 3-5 所示。这样就可以通过变量 L 找到真实的数据。由图 3-5 也可以看出，其实变量 L 可以存放各种类型的数据(可以是数字、列表、字符串、字典，等等)。所以，Python 与其他编程语言有一点不同之处在于，Python 的变量中存放的是指针，而不是真实的值！指针的大小是一个 word(在 64 位系统中，一个 word 的大小为 64bit)。也就是说，Python 中所有的变量都是以同一种形式来存储的，即指针指向真实值的形式。

图 3-5　数据存放示意图

了解了 Python 中变量与数据的存储形式，再来回顾一下前面所讲的局部变量与全局变量。我们可以根据存储形式重新定义一下这两个名词：局部变量(也包括函数的参数变量)的容器都是在函数内部的；全局变量的容器是在所有函数的外部的。那么如果一个函数中共有 k 个局部变量，则该函数中就应该有 k 个 word 大小的指针。有了上述知识作为基础，我们再详细讲解可变与不可变类型。

1. 不可变类型

对于像字符串这种不可变的数据类型，所有改变字符串中元素的操作都是产生一个新的字符串值，而不是在原有值的基础上直接做修改。例如，要把字符串 str="I like Dr. Sha"变成"U like Dr. Sha"。很容易想到一种错误的改法：str[0]= 'U'，但实际上，不可变的数据类型是不允许这样操作的。但是字符串可以分片，所以正确的解法为：str='U'+str[1:len(str)]。'U'+str[1:len(str)]会产生一个新的字符串并存放在一段新的空间内，当将'U'+str[1:len(str)]又一次赋值给 str 时，str 容器会丢弃原来存放的地址而重新存放新的

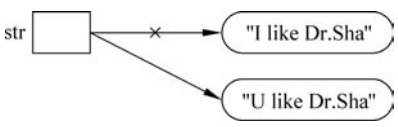


图 3-6 修改字符串示意图

地址，如图 3-6 所示。当然，对于上面的例子，也可以使用字符串的内置函数 replace()来处理：str. replace('I', 'U')。当然这种方法同样不会在原来值的基础上做改变，也是会产生一个新的值并存放在另一块空间，如果仅仅写下 str. replace('I', 'U')这样一句代码，那么在后续操作过程中是无法找到"U like Dr. Sha"这句话的，因为没有一个变量可以指引到它，所以应该这样写：str=str. replace('I', 'U')。这样就可以通过 str 变量找到新改变的值，当然也可以重新给变量命名，如 k=str. replace('I', 'U')。

总结一下分片和“+”号，不管是可变还是不可变的序列类型：

(1) 分片必定产生新的序列；

(2) “+”号在等号右边，必定产生新的序列，然后将新的序列地址赋给等号左边的变量。

例如，设 L=[1,2]，当用 L=L+[3]来给列表增加元素时，首先，对于 L+[3]部分，并不会在原来的数据上直接添加元素，而是产生了一个新的列表值[1,2,3]并保存在另一块空间内；其次，当将 L+[3]再一次赋值给 L 的时候，L 这一容器将会丢弃原来[1,2]的地址而存放[1,2,3]的地址。

2. 可变类型

讲完不可变类型，再来看看可变类型。对于像列表这类可变的类型，某些操作是可以直接在原数据的基础上直接改变的。例如，列表的 append 操作，可以说 append()函数是列表、字典这类可变类型数据结构的一种专有函数，通过 append()函数的方式来添加元素是在原有值的基础上直接进行改变，而不是重新产生一个新的值。例如，L=[1,2]，当使用 L. append(3)时，原列表值[1,2]本身直接变成[1,2,3]，L 列表保存的地址没有改变，如图 3-7 所示。

对于可变类型来说，在赋值的时候经常容易出错。相信细心的同学已经注意到，根据前面讲过的一些知识，对于一个已有的列表 A，L=A 和 L=A[:]这两种方式都会使 L 列表和 A 列表有相同的值，但这两种赋值方式是否有区别？下面通过一个例子来具体解释。

假设 A=[1,2]，当使用 L=A 时，L=A 操作创建了一个新的容器 L，但 L 中存放的地址和 A 是完全一样的，即 A 和 L 这两个变量同时指向一个值。因此 L 和 A 有一个共同列表[1,2]。如图 3-8(a)所示。当使用 L=A[:]时，在创建了一个新的容器 L 的同时也将原有的数值复制了一份存放在一块新的空间中，L 存放的是这个新的地址，而不是与 A 中一样的地址，如图 3-8(b)所示。

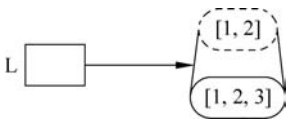
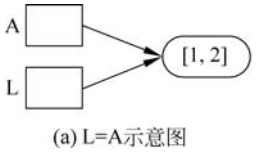
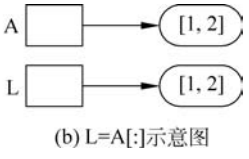


图 3-7 append 操作示意图



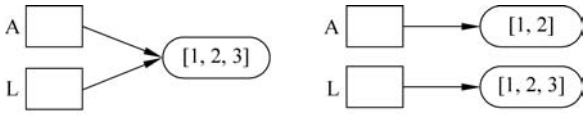
(a) L=A示意图



(b) L=A[:]示意图

图 3-8 不同赋值方式的示意图

了解了这两种赋值方式的不同，可以进一步思考：这两种不同的赋值方式在编程上是否会有很大的影响？答案是肯定的。沿用上面的例子，在要对 L 做 L.append(3)操作时，会得到不同的结果。首先对于 L＝A 的赋值方式，由于 L 和 A 其实指向的是同一个值，那么对 L 的改变也就是对 A 的改变，所以 L.append(3)操作后，A 和 L 的值全都变为了[1,2,3]，如图 3-9(a)所示。而对于 L＝A[:]的方式，由于 L 和 A 其实是各自独立的，所以 L.append(3)操作后只会改变 L 中的值，而 A 的值不会变动，如图 3-9(b)所示。



(a) L=A的修改示意图　　(b) L=A[:]的修改示意图

图 3-9　不同赋值方式在修改时的对比图

这里建议大家养成一个好的编程习惯，尽量用 L＝A[:]这种方式赋值，以避免程序中出现不必要的错误。

对列表进行添加元素的方法总结

向列表中添加新元素有 3 种常用的方法，分别为：L＝L＋[i]，L.append(i)，L＋＝[i]，下面总结一下这 3 种方法的差别：

(1) L＝L＋[i]在每次执行时都会将原列表复制一次，L 指向新列表，并在新列表中加入新元素。

(2) L.append(i)只是将新元素直接添加到原列表中，不会产生新列表。

(3) L＋＝[i]的执行效果和 L.append(i)类似，也是在原列表中直接添加元素，不会复制原列表。

其实对于第三种 A＋＝B 的添加方法，其专业术语叫作增强赋值语句，A＋＝B、A－＝B、A＊＝B 这类语句都是增强赋值语句。虽然 Python 的增强赋值语句是从 C 语言借鉴过来的，但其有自己的独特之处：对于不可变变量来说，A＋＝B 其实就等价于 A＝A＋B；但是对于可变变量来说，A＋＝B 是直接在原值的基础上做修改。Python 相比其他编程语言还有一种独特的赋值语句："A,B＝B,A"，该赋值语句可以直接实现两个变量值的交换功能。

Python 中可以通过 id()函数来查看列表存储的地址，如果语句执行前后列表的地址不同，那么就说明原列表被复制，此时的 L 指向新列表的地址；反之，则列表没有被复制，L 仍指向原列表。

同学们可以尝试运行代码<程序：L＋＝[i]和 L＝L＋[i]的讨论>，验证 L＋＝[i]和 L＝L＋[i]执行后是否产生了新列表。

通过对比打印出的 id 结果，可以发现 L＋＝[i]执行之后列表地址没有改变，说明 L＋＝[i]不会对原列表进行复制。因为列表是可变类型的，所以可以在原列表的基础上进行改变。而执行 L＝L＋[i]之后，L 的地址改变了，原列表被复制产生了新列表，L 指向新列表。

对于不可变类型的数据，比如 int 型变量 a，在进行 a＋＝1 的操作之后，可以发现 a 指向了一个新的地址。因为对不可变类型进行修改时，并不是在原有地址的数据中进行修改，而是将变量重新指向一个新的地址。

```
#<程序：L+=[i]和 L=L+[i]的讨论>
>>> L=[0,1,2,3]
>>> id(L)
2149974408392
>>> L+=[4]
>>> id(L)
2149974408392
>>> L=L+[5]
>>> id(L)
2149974409352
>>> a=0
>>> id(a)
1960169248
>>> a+=1
>>> id(a)
1960169280
```

练习题 3.3.1 说出<程序：反转一个列表中的元素 1>中打印的结果。

```
#<程序：反转一个列表中的元素 1>
a=[1,2,3,4,5]
b=a
b.reverse()          #反转列表
print("b=",b)        #b= [5, 4, 3, 2, 1]
print("a=",a)        #a= [5, 4, 3, 2, 1]
```

【答案】 打印的结果为：b=[5，4，3，2，1]和 a=[5，4，3，2，1]。可以看到，程序中使用了 b=a 这种赋值操作，使 b 和 a 有一个相同的列表，reverse()这类列表专有函数是直接在原有的对象上操作的，所以当我们把 b 通过 b.reverse()改变后，会同时修改 a。

练习题 3.3.2 说出<程序：反转一个列表中的元素 2>中打印的结果。

```
#<程序：反转一个列表中的元素 2>
a=[1,2,3,4,5]
b=a[:]
b.reverse()          #反转列表
print("b=",b)        #b= [5, 4, 3, 2, 1]
print("a=",a)        #a= [1, 2, 3, 4, 5]
```

【答案】 打印的结果为：b=[5，4，3，2，1]和 a=[1，2，3，4，5]。当使用 b=a[:]时，b 会独自拥有一个与 a 的值完全相同的列表，此时对 b 列表的操作不会对 a 产生影响。

兰　兰： 我知道元组是不可变的数据类型，元组里面有列表，例如，T=([1,2],0)，那么 T[0]能否被改动？

沙老师： 你这个问题比较狡猾，说能被改动也不完全对，说不能改动也不对。简单来说，T[0]能被列表的专有函数改动，也就是说，可以在原有列表上面改动。但是 T[0]不能通过产生新列表的方式改动。即，元组的顶层结构是不能改变的。下面将详细解释。

如图 3-10 所示为元组 T 的结构。元组作为一个不可变类型是指该图圆框中所标注的部分为不可变的。即当 T=([1,2],0)时，T 的顶层结构不可以改变，元组中元素指向的地址空间不可以改变。例如，T[0]=T[0]+3 在元组 T 中的操作是不允许的，因为这个操作要生成新的 T[0]=[1,2,3]，T 要指向新的地址空间，那么就改变了 T 的顶层结构。但由于 T[0]作为列表是可变类型，所以我们可以在 T[0]的原有列表上做改动，当使用它的专有函数时，不会产生一个新列表，也就不会改动元组的顶层结构。

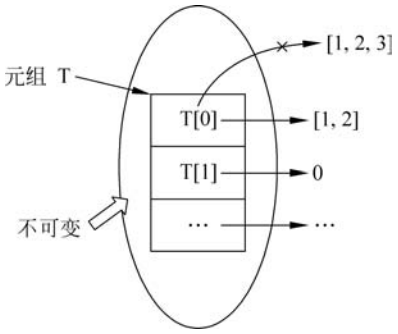


图 3-10　元组的结构

3.3.2　参数的传递问题

函数的重要性相信大家已经清楚了，那么在编写函数的时候我们经常需要进行参数变量的传递。这里先给出定义，什么是参数变量？参数变量就是定义函数时，括号内所定义的变量。注意，参数变量也是局部变量，所以它也有相应的在函数内的容器，里面存放对应真实值的指针。或许大家在前面的编程练习中遇到过下面这样的问题，见<程序：参数传递问题举例>。

```
#<程序：参数传递问题举例>
def fun(L):
    L = L + [4]
A = [1,2,3]; fun(A)
print(A)              #输出结果仍然为[1,2,3]
```

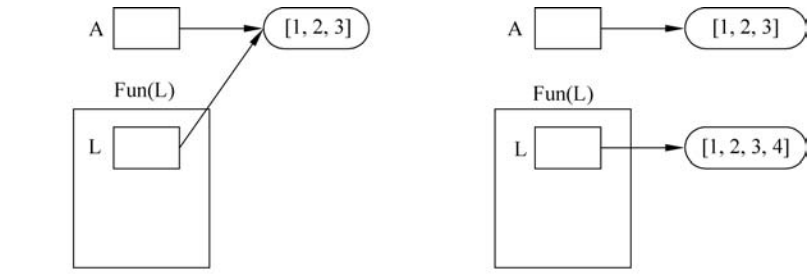
我们本来是希望通过函数 fun()来给列表添加一个元素 4，使得最终 A=[1,2,3,4]，但为什么执行 fun()函数之后 A 并没有被改变呢？或许大家还遇到过另一种情况：我不想在执行函数过后传递进去的变量被改变，但实际执行函数过后列表参数却被改变了！这又是怎么回事？

Python 在进行函数调用时，假如函数的参数变量叫作 L，调用函数时所传的变量为 A，那么参数的传递就相当于是 L=A(无论变量是可变类型还是不可变类型，也无论变量是整数、浮点数、字符串还是列表，都是如此)。即参数传递时传递的是指针，所以变量 L 与 A 指向同一个地址(如图 3-11(a)所示，A 与 L 指向的是同一个地址)。

对于<程序：参数传递问题举例>，其中 fun()函数中的 L=L+[4]，是对“=”右边的列表 L 使用“+”号操作，会产生一个新列表，然后将它赋值给“=”左边的参数变量 L，所以 L 现在指向一个新的列表，和 A 所指的列表脱钩了。在函数外的 A 还是指向[1,2,3]，没有变动，如图 3-11(b)所示。

练习题 3.3.3　请同学们利用 id()函数分别检验：

(1) 定义变量 A="abcd"，定义函数 def my_fun(L)：return L，调用 my_fun(A)时变量 L 和 A 的地址是否相同；



(a) 参数传递指向同一个地址　　(b) L=L+[4]后示意图

图 3-11　参数的传递

(2) 验证<程序：参数传递问题举例>中变量 L 和 A 的地址是否相同。

了解了参数传递的基本原理后，再结合前面所讲的“可变与不可变类型的讨论”，相信大家可以理解 Python 在参数传递的过程中会出现一些似乎奇怪的现象：当参数为可变类型变量时，函数内部对参数所做的操作可能会修改传进来的变量的值；而当参数为不可变类型变量时，函数内部对参数所做的操作当然是不会改变原变量的值的。这里我们特别用一节的篇幅来讲解像列表这样的可变类型作为参数时需要注意的问题，希望大家在编程的时候多加注意！

假设有列表 L 是函数定义的参数变量，列表 A 是调用函数时外部所传递的变量，则可以总结出以下两点规律：

(1) 当整个列表 L 出现在函数内部的“=”左边时(排除“+=”的形式)，所做的更改并不会影响传递进来的原列表 A 的值。因为函数内产生新列表，并且 L 会指向新的列表。

(2) 当在列表 L 上直接更改时，例如，L 的某个元素出现在“=”左边，或者 L 使用了列表的内置函数或者“+=”时，所做的更改将会影响传递进来的原列表 A 的值。因为函数内是在原列表上做修改的。

接下来对这两点规律展开详细解释。

(1) 当整个列表 L 出现在函数内部的“=”左边时(排除“+=”的形式)，所做的更改并不会影响传递进来的原列表 A 的值。因为函数内产生新列表，并且 L 会指向新的列表。

执行<程序：列表作参数举例 1>，最后一行 print(L)得到的结果是什么？有了前面的讲解，我们可以知道，L 的初始值为[1,2,3]，L 作为参数传入 ex1 函数中，其本质是通过指针传递的，即函数外的 L 与函数内的 L 指向的是同一个地址。函数内，语句 L=[0]产生新列表[0]，其赋值操作给参数变量 L 并不会影响传递进来的外界变量 L，故在外层打印 L 的结果仍为[1,2,3]。

```
#<程序：列表作参数举例 1>
def ex1(L):
    L = [0]              #整个列表出现在"="左侧
L = [1,2,3]
ex1(L)
print(L)                 #[1,2,3]
```

(2) 当在列表 L 上直接更改时，例如，L 的某个元素出现在“=”左边，或者 L 使用了列表的内置函数或者“+=”时，所做的更改将会影响传递进来的原列表 A 的值。因为函数内

是在原列表上做修改的。

首先看一个例子，<程序：列表作参数举例 2 >的 add()函数实现了对列表中的每个值进行加 1 的操作，并返回结果。同时我们打印 X，它的值是多少？

A＝add(X)执行完后，函数返回值 A 为[2,3,4]，而根据规则，X 作为参数传递到 add()函数中(在函数中列表名为 L)，记得参数传递时是指针的复制，也就是 L＝X，参数变量 L 指向了 X 所指的列表。列表 L 的单个元素所做的改变也会同时影响传递进来的原来列表 X 的值，所以 add()函数执行后，X 的值也被改变了，X＝[2,3,4]。我们用画图的方式来做一个具体的讲解，见图 3-12。

```
#<程序：列表作参数举例 2 >
def add(L):
    for i in range(0,len(L)):
        L[i] = L[i] + 1
    return (L)
X = [1,2,3]
A = add(X)              #此时 A = [2,3,4]
print(X)                #[2,3,4]
```

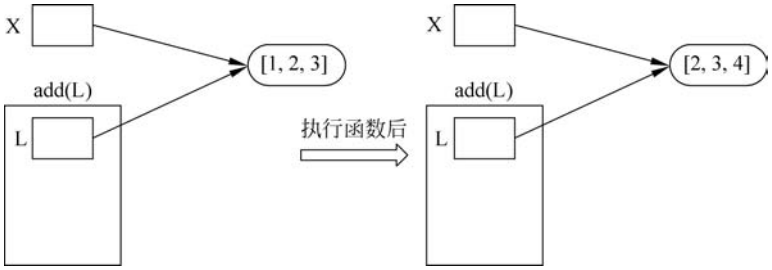


图 3-12　<程序：列表作参数举例 2 >示意图

然而我们需要认真思考一下，像<程序：列表作参数举例 2 >中的 add()这样的函数好吗？是完美函数吗？显然不是，因为一个完美函数既不会受到外界的干扰，也不会影响到外界的环境。add(L)将函数外的变量 X 的值改变了，而实际上我们并不希望改变 X 的值，这就需要对上面的程序做修改，应养成良好的编程习惯，在函数内部增加 L＝L[:]语句，使得进入参数内后，参数变量 L 与外界的 X 指向两个不同的地址的列表。见<程序：列表作参数举例 2_修改>，此时再对 L 操作，就不会影响到函数外的 X。

```
#<程序：列表作参数举例 2_修改>
def add1(L):
    L = L[:]            #注意，执行该语句后 L 指向的是复制的列表
    for i in range(0,len(L)):
        L[i] = L[i] + 1
    return (L)
X = [1,2,3]
A = add1(X)             #此时 A = [2,3,4]
print(X)                #[1,2,3]
```

写函数时,要把函数当作一个黑匣子,在黑匣子里面不应该对函数外的变量的值有所改变。在传递函数参数时,如果要传递的参数为像列表这种可变数据类型,为了要保护原来列表 L 的内容,建议函数一开始就建立一个全新副本,利用 L_new=L[:](或 L=L[:])方式,然后函数的操作都在这个复制的新列表上进行。

再来看一个例子,见<程序:列表作参数举例 3>。

```
#<程序:列表作参数举例 3>
def ex4(L):
    L.append(16)
    return L
X=[1,2,3]
A=ex4(X)              #此时 A=[1,2,3,16]
print(X)              #[1,2,3,16]
```

根据规则,<程序:列表作参数举例 3>在传入变量 X 的时候,X 与 L 指向的是同一个块地址。在函数 ex4()中,L 使用了列表的内置函数 append(),由于该函数同样会直接修改 L 中的值,而不是产生一个新的列表存放在新的空间中,所以对 L 的改变也就是对 X 的改变,所以最终 X 的值也被修改成[1,2,3,16]。

在函数内修改输入参数列表的内容,有时候并不是程序设计者所预期的。比较好的方式是通过 return 将新的列表返回,而不改变参数列表的内容。即使要改变原来列表的值,也可以先返回后再赋值改变。例如<程序:列表作参数举例 2_修改>中,可以将 A=add1(X)改为 X=add1(X),这样 X 就指向了新的值。下面做一些练习。

练习题 3.3.4 执行下面的<程序:列表作参数练习 2>后,函数体 swap 外面的 L1 和 L2 有没有被交换?

```
#<程序:列表作参数练习 2>
def swap(L1, L2):
    L1, L2 = L2, L1
L1=[1,2,3]
L2=[100]
swap(L1,L2)
print("Do they swap?",L1,L2)
```

【解题思路】 L1 的初始值为[1,2,3],L2 的初始值为[100],L1、L2 作为参数传入函数 swap()后,在函数体内部做了 L1, L2=L2, L1 的赋值操作,由于是整个列表出现在“=”左侧,符合规则(1),所以完全可以将 swap()函数内部的变量 L1 和 L2 当成局部变量,所以实际上外界的 L1 和 L2 并没有做交换。通过画图来加以解释(见图 3-13)。列表 L1 和 L2 作为参数传入函数时,函数体内部的局部变量 L1 和 L2 会指向和函数体外部的 L1 和 L2 相同的列表,swap 内部执行 L1, L2=L2, L1 后,相当于执行了 temp=L1; L1=L2; L2=temp,局部变量 L1 和 L2 所指向的值确实交换了,但是没有影响到外面的 L1 和 L2。

练习题 3.3.5 在练习题 3.3.4 的基础上,在 swap()函数中增加 L1[0]=9,<程序:swap 程序 1>执行后,函数体 swap 外面的 L1、L2 的值分别为多少?

【解题思路】 根据上一个练习题的分析,执行完 L1, L2=L2, L1 后,函数 swap()内的

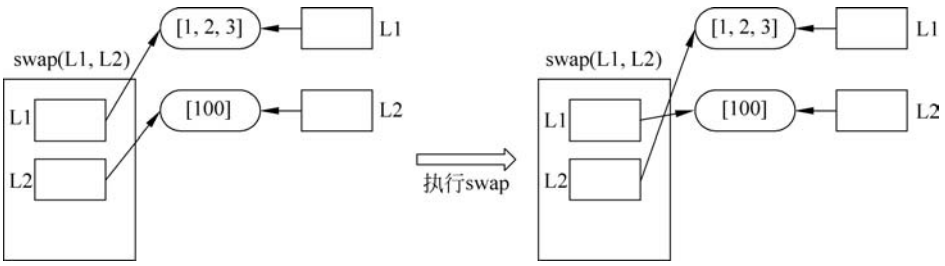


图 3-13　练习题 3.3.4 图解

局部变量 L1、L2 指向的值会交换，执行 L1[0]＝9，操作如图 3-14 所示，所以函数体外面的 L1＝[1,2,3]，L2＝[9]。

```
#<程序：swap 程序 1>
def swap(L1, L2):
    L1, L2 = L2, L1
    L1[0] = 9
L1 = [1,2,3]
L2 = [100]
swap(L1,L2)
print(L1,L2)
```

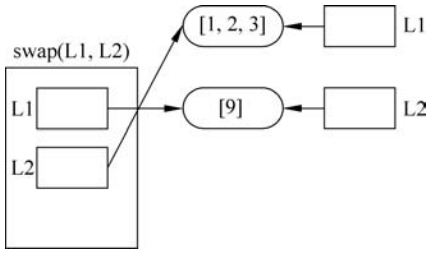


图 3-14　练习题 3.3.5 图解

练习题 3.3.6　对练习 3.3.4 的程序如何修改，才能交换函数外面 L1、L2 的值？

【解题思路】　通过上面的分析可知，在 swap()函数内部做 L1，L2＝L2，L1 不能影响外部 L1，L2 的值，有的同学可能会想，可以通过函数将 L1、L2 列表的内容交换，这样 L1、L2 的值就改变了，代码如<程序：swap 程序 2>所示。

```
#<程序：swap 程序 2>
def swap_for_fun(L1, L2):              #歪路，不可取
    T1 = L1[:]
    d1 = len(L1)
    d2 = len(L2)
    for e in L2:
        L1.append(e)
    for e in T1:
        L2.append(e)
    for e in L1[:d1]:
        L1.remove(e)
```

```
        for e in L2[:d2]:
            L2.remove(e)
L1 = [1,2,3]
L2 = [100]
swap_for_fun(L1,L2)
print(L1,L2)
```

但这种方式相当笨拙,不建议使用。其实,改变函数体外部的 L1、L2 有更好的方式,可以通过 return 传回新的值,对函数外面的 L1、L2 重新赋值的方式改变,代码如<程序:swap 程序 3>所示。

```
#<程序:swap 程序 3>
def swap(L1,L2):                    #用 return 传回新的值,这是正道
    L1, L2 = L2, L1
    return L1, L2
L1 = [1,2,3]
L2 = [100]
L1, L2 = swap(L1,L2)
```

3.3.3 默认参数的传递问题(可选)

当列表 L 作为默认值参数时会比较特别。我们在这里给大家展开讲解。由于列表作为默认参数在编程中使用较少,本节作为选择性学习内容,可以跳过。

一般而言,我们认为默认参数是不会随着函数的多次执行而改变的。但在 Python 中,当参数 L 的默认值为列表时,如果对作为默认参数的列表没有慎重使用,就会改变此参数的默认值。所以对于列表作为默认参数的使用要特别注意,建议大家在使用此列表时不要在原列表上做改动。因为,当调用函数没有给予参数 L 值时,则 L=默认值变量(默认值变量是个隐藏的局部变量,存放指针指向其默认值),所以当默认值变量所指的列表被改变时,就会被记住,下次使用时就会使用新的默认值。

下面通过几个例子来做详细的解释。首先,请大家思考,执行<程序:默认参数的诡异 1>,会打印出什么结果?

```
#<程序:默认参数的诡异 1>
def append_1(L = []):
    L.append(1)
    return(L)
print(append_1())                   #[1]
print(append_1())                   #[1,1]
print(append_1([2]))                #[2,1] 因为调用时没有使用默认参数
print(append_1())                   #[1,1,1]
```

对于<程序:默认参数的诡异 1>,L 是一个默认参数,函数隐藏的默认值变量仍旧有一个自己的容器,初始状态下指向一个默认值“[]”,当我们前两次调用 append_1 函数时,由于

没有给参数 L 赋值，则 L 会等于默认值变量，也就是 L 和默认值变量指向同一个默认值列表，所以每次 L 被改变都会使得默认值被改变并且被记住，即默认值变量先后由[ ]变为[1]，再变为[1,1]。直到第三次调用时，因为对 L 有赋值，所以 L 指向了另一块空间[2]，然后根据 append 操作，由[2]变为了[2,1]，而默认值变量仍为[1,1]。当最后一次调用函数时，由于仍旧使用的是默认值变量，所以函数在上次记住的默认值基础上做修改，故结果为[1,1,1]，此时默认值变量也为[1,1,1]。整个过程如图 3-15 所示。

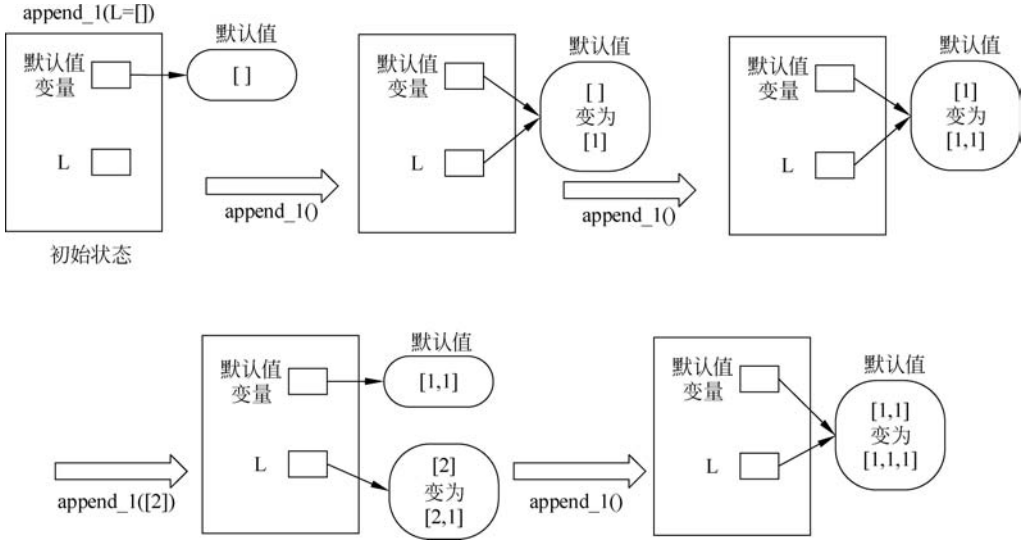


图 3-15　<程序：默认参数的诡异 1 >示意图

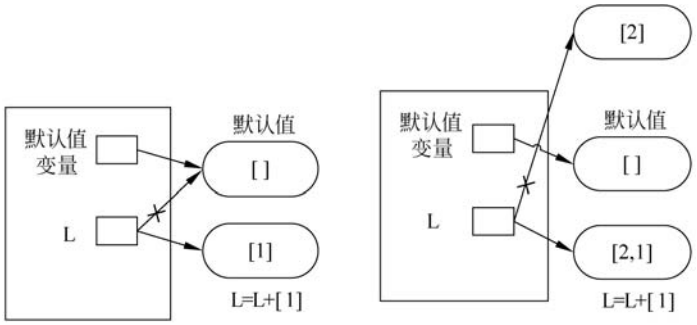
练习题 3.3.7　请同学们再次利用 id()函数，将其添加到程序中，验证<程序：默认参数的诡异 1 >中每次调用函数时 L 的地址是否有改变，是否与图 3-15 的说明一致。

下面稍微修改一下上一个程序中的代码，得到<程序：默认参数的诡异 2 >，请大家思考，会打印出什么结果。

```
#<程序：默认参数的诡异 2 >
def add_1(L = [ ]):
    L = L + [1]
    return(L)
print(add_1())          #[1]
print(add_1())          #[1]
print(add_1([2]))       #[2,1]
print(add_1())          #[1]
```

<程序：默认参数的诡异 2 >中，L=[]为 add_1 函数的默认参数，在函数内，执行语句 L=L+[1]后会指向产生的新列表，L 不再指向默认值列表了。故对参数 L 的更新不会影响到默认值列表，所以无论执行多少次 add_1()，打印的结果都是[1]，见图 3-16(a)。而当执行 add_1([2])时，没有使用默认参数，故直接将[2]添加元素 1 即可，故打印结果为[2,1]，见图 3-16(b)。

由于我们在使用默认参数时，希望其值不会被随意改变，所以当默认参数的值是一个列



(a) add_1()示意图　　(b) add_1([2])示意图

图 3-16　两次执行结果

表时，我们可以首先把它复制成一个新的列表，然后再对这个新列表做操作。

3.4 深入探讨列表的常用操作与开销

前面的章节中很多知识点都是关于列表的，由此也可以看出列表的强大之处。在实际编程中，列表也确实是最常用的一个数据结构。列表作为一个可变类型，为我们的编程提供了很多方便，但就在我们可以随意修改列表的同时也很容易出错。本节将主要解释列表的常用操作中所需要注意的关键问题以及时间开销，然后再介绍生成列表时可以使用的一些技巧，从而使大家更好地理解和使用列表。

3.4.1 添加列表元素的讨论

我们在写程序时经常需要向列表中添加元素，前面也已经讲解过，向列表添加元素主要有两种常见的方法(假设待添加的元素为变量 i)：一种是 L.append(i)，另一种是 L=L+[i]。我们曾经对比过这两种添加方式是否会改变列表本身，与此同时，这两种不同的方法在时间开销上也是有很大差异的，我们不可不知。首先来看一下<程序：append()和 L+[i]在时间开销方面的对比>，然后再具体比较造成这种时间差异的原因。

```
#<程序：append()和 L+[i]在时间开销方面的对比>
import time
def test_time(k):
    print("***** k = ",k)
    L = []
    start = time.clock()
    for i in range(k):
        L.append(i)
    elapsed = time.clock() - start
```

```
        print("使用 append 花时间: ",elapsed)

        start = time.clock()
        for i in range(k):
            L = L + [i]
        elapsed = time.clock() - start
        print("使用 L = L + [i]花时间: ",elapsed)

test_time(5000)
test_time(10000)
test_time(20000)
test_time(40000)
```

程序的运行结果可能如下(因计算机速度差异而会不同):

```
***** k = 5000
使用 append 花时间: 0.00041552276091237107
使用 L = L + [i]花时间: 0.1295196264525024
***** k = 10000
使用 append 花时间: 0.0007775500765575816
使用 L = L + [i]花时间: 0.5707438386154383
***** k = 20000
使用 append 花时间: 0.001415452159365449
使用 L = L + [i]花时间: 2.776138045618552
***** k = 40000
使用 append 花时间: 0.003287481723685204
使用 L = L + [i]花时间: 10.821593001054048
```

通过这个程序可以发现此程序中 append()的时间开销远小于 L=L+[i],循环次数越多,它在时间上的优势越明显。这是因为 L=L+[i]在每次执行时都会将原列表复制一次产生一个新列表,每次的复制过程都会产生大量的时间开销。而 append()只是将新元素直接添加到原列表中,不会产生新列表,因此能够在时间开销方面优于 L=L+[i]。

3.4.2　删除列表元素的讨论

第 2 章讨论了在使用 for 循环时要特别注意列表被改动的问题,在学习了列表的专有方法之后,我们结合循环来讨论一下 remove()的使用以及 remove()带来的时间开销。

1. 列表被改变是危险的

我们知道 for 循环要特别注意列表被改动的问题,其实,while 也要特别注意这些情况,我们同样来看一个程序,该程序也是删除列表中所有为 0 的元素。请问同学们,该程序为什么是错误的? 如<程序:删除列表中为 0 的元素_2(错误)>所示。

```
#<程序：删除列表中为 0 的元素_2(错误)>
L = [0,0,1,2,3,0,1]
i = 0
while i < len(L):
    if L[i] == 0: L.remove(0)
    i += 1
print(L)            #输出结果为[1, 2, 3, 0, 1]
```

在这个程序中，当 i=0 时，L[0]是列表中第一个值为 0 的元素。用 remove()删除第一个 0 后，L 被更新为 L=[0,1,2,3,0,1]。第二次循环时，i=1，这时 L[1]的值为 1，即直接跳过了原列表中的第二个 0。接着一直循环，直到找到原列表中第三个 0 为止，通过 L.remove(0)删除了原列表中的第二个 0，此时 L 又被更新为 L=[1,2,3,0,1]。当再想进入下一次循环时，i=5，len(L)=5，不满足循环条件，故直接退出循环，因此也不会再进行 remove()操作。但是此时原列表中的第三个 0 仍然存在，所以程序运行的结果是错误的。

关于上面的程序应该如何修改，其实至少有 3 种方法，这里讨论一下。很多同学最先想到的应该是使用“while(0 in L):L.remove(0)”的方法，这样的写法看上去非常简单。这个解决办法是正确的，我们建议在长列表循环中应该尽量减少使用 remove()，除了它每次只会去掉一个元素外，另外它也有可能产生无谓的耗时，后面将会详细介绍。代码如<程序：删除列表中为 0 的元素改进 1>所示。

```
#<程序：删除列表中为 0 的元素改进 1>
L = [0,0,1,2,3,0,1]
i = 0
while 0 in L:
    L.remove(0)
print(L)
```

第二种方法，就是使用索引来删除列表中为 0 的元素。使用这种方法，程序可以清楚地知道在每轮循环中，应该遍历列表 L 的哪一个元素，直到遍历到 L 的最后一个，表示 L 的每个元素都已经被判断过。注意，当我们删除一个列表中的元素后，列表中所有元素的索引都会减小 1，这时，我们不再对索引进行加 1 操作。代码如<程序：删除列表中为 0 的元素改进 2>所示。

```
#<程序：删除列表中为 0 的元素改进 2>
L = [0,0,1,2,3,0,1]
i = 0
while i < len(L):
    if L[i] == 0: L.remove(0)
    else: i += 1
print(L)
```

第三种方法，是建立一个新的列表 L1，每次循环都将 L 中不为 0 的元素放入列表 L1，直到将不为 0 的元素全部放入 L1 中为止。代码如<程序：删除列表中为 0 的元素改进 3>所示。

```
#<程序：删除列表中为 0 的元素改进 3>
L = [0,0,1,2,3,0,1]
L1 = []
for e in L:
    if e!= 0: L1.append(e)
print(L1)
```

兰　兰：那这 3 个程序有什么区别呢？哪个程序会比较好？

沙老师：这 3 个程序的执行时间有的快有的慢。造成执行时间的不同是什么原因呢？我们仔细探讨一下，这是很有意思的。

2. 使用不同方法删除列表元素的执行时间

在“删除列表中为 0 的元素”的例子中，我们使用了 3 种方法，这些方法的不同在哪里呢？

为了要体现它们的不同，我们先组织一个列表 L，L 由 1 000 000 个 1 和 1000 个 0 组成。请将列表 L 中所有 0 去掉。这里用了上述 3 种方法来实现，程序如下所示。分别执行以下程序得到这 3 种方法的运行时间，从时间上判断 3 种方法的优劣，同学们也可以在自己的计算机上动手实践一下。

```
#<程序：将列表中的 0 去掉方法 1>
import time                              #引入 time 模块
start = time.clock()
L = [1 for i in range(1000000)] + [0 for i in range(1000)]
while 0 in L:
    L.remove(0)
elapsed = time.clock() - start
print("方法一：")
print("用 while 0 in L, remove 后列表长度 = ",len(L),"花时间：", elapsed)
```

```
#<程序：将列表中的 0 去掉方法 2>
import time                              #引入 time 模块
start = time.clock()
L = [1 for i in range(1000000)] + [0 for i in range(1000)]
i = 0
while i < len(L):
    if L[i] == 0: L.remove(0)
    else: i += 1
elapsed = time.clock() - start
print("方法二")
print("一个个遍历再 remove 后，列表长度 = ",len(L),"花时间：", elapsed)
```

```
#<程序：将列表中的 0 去掉方法 3>
import time                              #引入 time 模块
start = time.clock()                     #程序开始运行时的时间
```

```
L = [1 for i in range(1000000)] + [0 for i in range(1000)]
L1 = []
for e in L:
    if e!= 0: L1.append(e)
elapsed = time.clock() - start                    # 循环结束消耗的时间
print("方法三")
print("新列表 L1 的长度: ",len(L1), "花时间: ", elapsed)
```

程序中“L=[1 for i in range(1000000)]+[0 for i in range(1000)]”语句表示列表 L 由 1 000 000 个 1 和 1000 个 0 组成,这种列表的表示方式会在 3.4.3 节中详细讲解。

下面就来探讨一下程序的运行结果。同学们的实验结果怎么样呢?在我的计算机里,方法一需要 20s 左右完成;方法二需要 10s 左右完成;而方法三在 1s 之内就能够快速完成。你们知道是为什么吗?

方法二比方法三慢的原因是每执行一次 L.remove(0),就要从头遍历 L,直到找到第一个 0 为止。所以,每一次 remove()都要重复遍历 L 中前面的 1 000 000 个 1。方法二中要执行 1000 次 remove(),那么就会重复遍历 1000 次 L 中前 1 000 000 个 1,导致程序执行时间变得很长。

方法一是最慢的。因为它不仅和方法二一样,每次 remove()都要重复遍历,而且 while 0 in L 也要重复检查前面的 1 000 000 个 1,这样又要重复检查至少 1000 次,导致方法一程序执行时间差不多是方法二的两倍长。

上述例子对不同的方法的时间进行比较,我们可以了解到,while i in L 和 remove 带给程序的时间花销是很大的。我们在编写程序处理长序列时应该注意。

练习题 3.4.1 在上面的问题中,如果 L=[1 for i in range(1000000)]+[0 for i in range(10000)],也就是 0 的个数从 1000 变成 10 000,请预测 3 个程序的大约执行时间,为什么?

【解题思路】 执行完程序可以发现,第一个程序的执行时间变为了之前的 10 倍,200s 左右。第二个程序也是之前时间的 10 倍,100s 左右。第三个程序的执行时间基本没有差别,依旧在 0.15s 左右。

为什么会有这样的差别?因为在第二个程序中,程序的执行时间是列表的长度加上 0 的个数乘以在 0 前面的 1 的个数。而第一个程序的执行时间至少是第二个程序的两倍。在第三个程序中,程序的执行时间和列表的长度成正比。而 0 的个数从 1000 变为 10 000,相对于列表中还有的 1 000 000 个 1,对列表的长度影响很小,所以时间基本没有变化。

通过对在列表中添加或删除元素的讨论,我们知道应该在循环内谨慎使用 append()或者 remove(),因为使用它们时会改变列表中元素的个数。相应地,每个元素对应的索引也会发生改变,在后续的执行中容易忽视这些变化而造成错误。同样,while 循环中也可能会由于使用 append()或者 remove()而造成同样的问题。而且,remove()方法所带来的开销也是不容忽视的。

因此,最好是在保持原有列表的基础上,另外产生一个新的列表。这种问题在以后的编程中会经常出现,希望同学们可以活学活用,避免类似的错误。

3.4.3　生成列表的一些技巧

本节介绍使用列表时的一些小技巧，从而使编程更加方便。

1. 列表推演表达式

列表推演表达式(List Comprehensive Expression)可以说是一种轻量级的循环，可以用于创建新的列表。比如<程序：列表推演表达式举例_1>。

```
#<程序：列表推演表达式举例_1>
L = [i for i in range(10)]
L          #输出为[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

上例中一个短短的语句就生成了0～9的一个序列，我们并不需要再像以前一样，单独写一个函数去生成该序列，是不是非常方便！注意，这个表达式中没有任何的标点符号。

再来看<程序：列表推演表达式举例_2>。

```
#<程序：列表推演表达式举例_2>
import random
L = [random.randint(1,100) for i in range(10)]
L          #输出为[74, 5, 42, 54, 71, 55, 67, 96, 100, 11]
```

这个例子中，我们利用列表推演表达式生成了一个有10个元素的列表，列表中的每一个元素都是1～100的某个随机数。

再看另一个例子，假设已经有一个矩阵M=[[1,4,7],[2,5,8],[3,6,9]](关于矩阵的相关知识将在下一部分讲解)，那么可以进行如下操作。

```
#<程序：列表推演表达式举例_3>
M = [[1,4,7],[2,5,8],[3,6,9]]                    #M矩阵3行3列
col0 = [row[0] for row in M]                     #得到矩阵的第0列的元素
print(col0)                                      #[1,2,3]
col_new = [row[0] * 2 for row in M]              #得到矩阵的第0列的元素,同时乘以2
print(col_new)                                   #[2,4,6]
filter = [row[0] for row in M if row[0] % 2 == 1]  #筛选出第0列中为奇数的元素
print(filter)                                    #[1,3]
```

通过<程序：列表推演表达式举例_3>，可以看到列表推演表达式的方便之处。其实可以这样理解列表推演表达式的原理：for循环会形成一个序列，然后通过筛选产生满足条件的新序列。当然，列表推演表达式还可以更复杂一些，代码如<程序：列表推演表达式举例_4>所示。

```
#<程序：列表推演表达式举例_4>
R = ["%d + %d" % (x,y) for x in range(4) for y in range(2)]
R      #输出为['0 + 0', '0 + 1', '1 + 0', '1 + 1', '2 + 0', '2 + 1', '3 + 0', '3 + 1']
```

在上面例子中,我们在列表推演表达式中写了嵌套 for 循环语句,生成了列表 R。

在平时的编程中,或许大家会经常需要用到在某一序列内随机挑选元素,生成一个长度为 num 的列表,以便后续使用。此时可以利用列表推演表达式的方式单独写一个小函数,方便使用。代码见<程序:列表推演表达式举例_5>。

```
#<程序:列表推演表达式举例_5>
def random_list(options,num):
    return [random.choice(options) for i in range(num)]
L = random_list(range(1,12),5)              #[8, 8, 8, 7, 1]
S = random_list("abcd",8)                   #['c', 'c', 'c', 'c', 'd', 'a', 'd', 'b']
```

<程序:列表推演表达式举例_5>中 random_list()函数用于从 options 变量中随机挑选元素组成长度为 num 的列表,其中 random.choice(options)函数是 random 的内置函数,用于在 options 中随机选取一个元素。有了这个函数之后,我们每次就可以通过简单的传参得到想要的列表。

2. 生成矩阵

矩阵也是编程时会经常用到的一个组织数据的方式。常用的有一维矩阵和二维矩阵,下面来看看可以怎样生成矩阵。

一维矩阵也就是我们平时所见的,如[1,2,3]、[aa,bb,cc]等都是一维矩阵。假如我们想要生成一个一维矩阵,长度为 100,每个元素都为 0,那么有两种生成方式:第一种为[0]*100;第二种为[0 for i in range(100)]。

对于二维矩阵,其形式为[[ ],[ ],[ ]…]。现在假设有一个二维矩阵为 A=[[1,2,3],[4,5,6]],实际上可以将其看成如下形式,其中,A[0][0]=1、A[0][1]=2、A[0][2]=3、A[1][0]=4、A[1][1]=5、A[1][2]=6。

$$A=\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix}$$

根据一维矩阵的生成方式,二维矩阵也有两种生成方式。

(1) A=[[1,2]]*3,则得到二维矩阵 A=[[1,2],[1,2],[1,2]]。但这种方式有一个致命的问题:当执行 A[0][0]=5 的时候,矩阵会变成 A=[[5,2],[5,2],[5,2]]。也就是说,我们以这种方式生成 3 行 2 列的矩阵 A 时,其实每行的数据指向的是同一个[1,2]的地址,所以改变其中一个值,其他行也会跟着改变,如图 3-17 所示。所以不推荐使用这种方式。

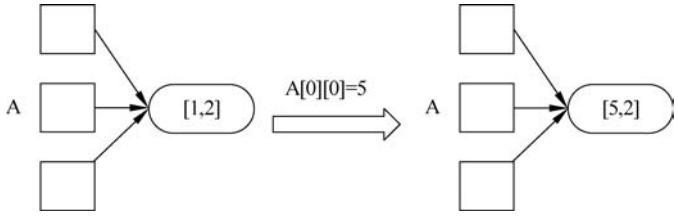


图 3-17　方式(1)生成二维矩阵示意图

(2) 推荐使用列表推演表达式的方式生成二维矩阵。A=[[1,2] for i in range(3)],会得到矩阵 A=[[1,2],[1,2],[1,2]]。即使执行 A[0][0]=5,也只是会改变相应位置的元素,矩阵会变成 A=[[5,2],[1,2],[1,2]],如图 3-18 所示。

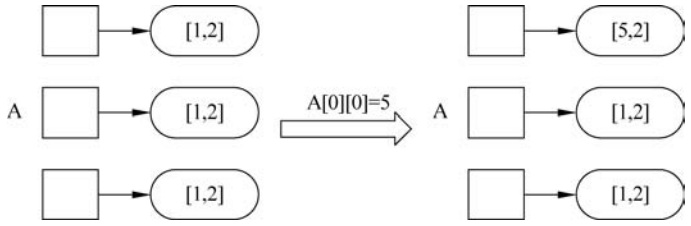


图 3-18　方式(2)生成二维矩阵示意图

练习题 3.4.2　如何使用列表推演表达式的方式产生一个 8×8 的二维矩阵,并且该矩阵中的所有元素都为 0。

【答案】　推演表达式如下:

```
L = [[0 for i in range(8)] for i in range(8)]
```

3.5　输入输出、文件操作与异常处理

我们在编程过程中经常需要与用户进行交互,即输入输出(Input/Output,I/O)。同时在很多情况下需要去读取文件中的内容,进行一系列处理后再将信息输出到文件中,即文件相关的操作。因此,I/O 与文件操作也是编程中重要的一部分。本节将分成 4 点来分别进行讨论:输入、输出、文件操作和异常处理。

3.5.1　输入

输入是用户与程序交互的一个重要阶段,如果没有处理好输入,那么整个程序都将出错。Python 提供了一个 input()函数,用来获得用户输入的数据,返回的是字符串。我们将输入问题进一步分成两点来详细讲解:一是类型转换;二是输入合法性检查。

1. 类型转换

通过 input()函数获得的信息是字符串,那么如何处理这些得到的字符串,变成我们需要的数据类型呢?下面首先介绍几个常用数据类型的转换函数。

1) 转换为数值类型的常用方法

我们知道,数值类型可以分为整数类型和浮点数类型。将字符串类型转换成相应的数值类型需要调用相应的转换函数。例如,int()函数可以将字符串转换为整数,float()函数可以将字符串转化为浮点数,比如 str="123",那么 int(str)的返回值为 123;如果 str=

"123.45",那么 float(str)的返回值为 123.45。然而,int()函数和 float()函数在进行数据类型转换时,对参数有一定的要求,如果传入的参数不能被转换,函数会报错,我们在传参时需要特别注意。

(1) 对于 int()函数,当参数为浮点数时,将向下取整。例如,int(3.7)的返回值为 3。对于 float()函数,当参数为整数时,就会返回浮点数。例如,float(3)的返回值为 3.0。

(2) 对于 int()和 float()函数,当参数为字符串时,可以用+、-号表示正负值。

2) eval()函数的妙用

eval(str)函数很强大,它可以将字符串 str 当成有效的表达式来求值并返回计算结果,相当于 str 去掉字符串的引号后,在 Python 中被执行。我们可以用该函数来计算字符串中有效的表达式,并返回结果。<程序:利用 eval 函数做数学计算>中用到了 eval()函数,功能是将 str 变成算术表达式来执行,实现了对“2+2”与“2^2”的计算。

```
#<程序:利用 eval 函数做数学计算>
eval('2 + 2')                #输出:4
eval('pow(2,2)')             #输出:4
```

eval()的功能不局限于此,其实该函数也可以实现将字符串转换成相应的对象,如 list、tuple、dict 和 string 之间的转换。例如<程序:字符串转换成列表方案一>、<程序:字符串转换成字典>。

```
#<程序:字符串转换成列表方案一>
a = '[[1,2], [3,4], [5,6], [7,8], [9,0]]'
type(a)                      #输出:<class 'str'>
a = eval(a)
type(a)                      #输出:<class 'list'>
a                            #输出:[[1, 2], [3, 4], [5, 6], [7, 8], [9, 0]]
```

```
#<程序:字符串转换成字典>
a = "{1: 'a',2: 'b'}"
type(a)                      #输出:<class 'str'>
a = eval(a)
type(a)                      #输出:<class 'dict'>
a                            #输出:{1: 'a', 2: 'b'}
```

3) 字符串如何转换为列表

字符串转换为列表也是十分常用的一个操作,如果希望将字符串的每个字符作为一个元素保存在一个列表中,则可以使用 list()函数,比如 str="123, 45",list(str)的返回值为['1', '2','3',',','','4','5']。注意逗号“,”和空白“ ”都当作一个字符。

如果希望将字符串分开,那么可以使用字符串专用方法 split。例如,字符串 str=“123, 45”,注意,str 中 45 前面有一个空格。将 str 以“,”分割,使用 L=str.split(“,”)便可实现。其返回值是一个列表[“123”,“45”],需要注意的是,得到的列表中每个元素都是字符串类型,空格仍然在字符串"45"里面。如果要得到整数类型的列表,还需要将字符串转换为数值,例如,使用如下语句:L=[int(e) for e in L]可将 L=[“123”,“45”]转换为单纯的整

数列表 L=[123,45]。

假如将字符串如 S="1,2,3,4"变为整数列表，除了使用上面的 split 的方式外，还可以使用前面介绍过的 eval()函数，我们知道，该函数可以实现将字符串转成相应的对象，因此需要将字符串 S 插入“[]”中，一种方式是"["+S+"]"，另一种方式是利用格式化语句处理 S 得到 A="[%s]"%S 得到'[1,2,3,4]'。然后再通过 eval()函数转换，即 eval(A)，eval()会自动将字符串 A 转换为列表。程序如下所示：

```
#<程序：字符串转换成列表方案二>
S = input("Enter 1,2, , , :")                    #Enter: 1,2,3,4
L = eval("[%s]" % S)                             #L = [1,2,3,4]
```

2. 输入合法性检查

在了解了输入信息常用的类型转换后，我们来看下面的<程序：对用户输入的两个数相加>，程序实现了对用户输入的两个数求和，用户输入的数会先用字符串 a 和 b 表示，再用 float()函数转化成浮点型相加，并输出计算结果。

```
#<程序：对用户输入的两个数相加>
def sum():
    a = input("请输入第一个数字：")              #等待用户输入第一个数字
    b = input("请输入第二个数字：")              #等待用户输入第二个数字
    c = float(a) + float(b)                      #将 a、b 转换为浮点型相加
    return c
```

sum()函数实现对输入的两个数相加的操作，但是这个函数写得严谨吗？并不是所有的用户都会“小心”，让输入什么就输入什么。比如上述程序中，若用户不小心输入错误，在输入第二个数字时输入的是“3.1.24”，这并不是一个数字，我们称这种输入为非法输入，然而代码并没有对这种非法输入做任何处理，所以这个函数就会报错，不能继续运行，显然是不合理的。所以在需要用户输入信息的时候，一个必不可少的步骤就是对用户的输入做合法性检查。

我们对用户的输入信息进行检查一般使用 while 循环，<程序：对用户输入的合法性检查>写出了常用的输入合法性检查的“模板”，该模板利用 while(True)来循环判断输入是否合法，若合法，则直接终止循环；若非法，则继续提示用户输入，直到合法为止。

```
#<程序：对用户输入的合法性检查>
while(True):
    s = input("请输入:")
    if s 合法: break
    else:print("输入不合法,需重新输入")
```

有了对用户信息的合法性检查的模板后，请大家思考如何检查用户输入的信息是否为数字？有的同学可能会说，Python 中字符串有一个自带的函数叫作 isdigit()，可以判断字符串是否只由数字组成。当输入是一般正整数时，这个函数是有用的。

但是各位可以测试,会发现该函数只有在字符串表示没有正负符号做前缀的整数时,返回 True,而如果字符串中的数字有正负号做前缀或者字符串表示的是浮点数时则无法识别出来。所以我们需要根据实际的情况自定义合法性检测函数,如<程序:检查字符串是不是数字>是我们自己实现的检测函数。函数 isnum()判断输入的字符串是不是数字,如果是则返回 True;否则返回 False。该函数实现的思路如下:若字符串 S 中只有一个字符,则该字符只能为无符号整数;若 S 中的字符个数大于一个,则 S 的第一个字符可以为+、-或者整数。若 S 中的一个字符为符号+、-,则第二个字符只能是数字;若 S 中的一个字符为整数,则第二个字符可以是“.”或者整数,S 中只能出现一个“.”号,一旦后面的判断中出现“.”则直接判断 S 中“.”号之后的字符必须都是数字。

```
#<程序:检查字符串是不是数字>
def isnum(S):
    if len(S)<=1:return S.isdigit()                    #S 只有一个字符,只能为整数
    for i in range(0,len(S)):
        if i==0 :                                      #第一个字符只能为"+""-"或整数
            if not (S[0]=="+" or S[0]=="-" or S[0].isdigit()):
                return False
        if i==1:
            if S[0]=="+" or S[0]=="-":                 #若第一个字符为"+""-"
                if not S[1].isdigit():return False     #则第二个字符只能为整数
            else:                                      #若第一个字符是整数,第二个字符可以为"."或整数
                if S[1]=="."":break
                if not (S[1].isdigit()):return False
        if i>1:                                        #第二个字符之后的字符只能为"."或整数
            if S[i]=="."":break
            if not S[i].isdigit():return False
    if i==len(S)-1:return True
    return S[i+1:].isdigit()
```

这样我们便可以对<程序:对用户输入的两个数相加>做如下改进,利用合法性检查的结构,调用我们自定义的合法性检查函数 isnum(),对用户输入进行检查,代码见<程序:对用户输入的两个数相加改进>。

```
#<程序:对用户输入的两个数相加改进>
def sum():
    while(True):
        a=input("请输入第一个数字:")                    #等待用户输入第一个数字
        if isnum(a):break
        else:print("输入不合法,需重新输入")
    while(True):
        b=input("请输入第二个数字:")                    #等待用户输入第二个数字
        if isnum(b):break
        else:print("输入不合法,需重新输入")
    c=float(a)+ float(b)                               #将 a、b 转换为浮点数相加
    return c
```

3.5.2　输出

在前面的代码示例中，多次用到了 print()函数，相信大家对于它的使用已经很熟悉了。下面进一步介绍 print()函数的完整形式，在 Python 中，print()的完整格式为：print(objects, sep,end,file,flush)，其中 objects 就是需要输出的那些内容，后面 4 个为关键字参数。

1. sep 关键字参数

在输出的字符串之间插入指定字符串，默认是空格，例如：

```
#<程序：sep 参数举例>
print("a","b","c")                    #输出：a b c
print("a","b","c",sep = " ** ")       #输出：a ** b ** c
```

2. end 关键字参数

在 print 输出语句的结尾加上指定字符串，默认是换行(\n)，而如果不想换行，则 end=''；但需要注意的是，如果不换行，那么 print 语句不会马上打印出当前需要打印的全部信息，直到需要换行打印的时候才会将这一行信息全部打印出来。

大家可以自行尝试，end 的值不止可以是空字符，还可以是其他字符，比如：

```
print("aaa",end = '@')            #输入这一语句以后，屏幕上不会打印出任何信息
print("qq.com")
最终输出为：aaa@qq.com
```

大家可以尝试以下代码，可以对 end 这一关键字有更清晰的理解。

```
print("aaa",end = '@')
for i in range(1000000000):
    a = 1
print("qq.com")
```

最终输出依然为：aaa@qq.com，但是大家可以看出在打印的时候，会先打印 aaa@，然后再等待打印“qq.com”的语句执行，最终将两个 print 中的信息输出为一行。

最后两个参数 file 和 flush 将在 3.5.3 节进行讲解。

3.5.3　文件操作

大多数程序都遵循：输入→处理→输出的模型，程序首先输入数据，然后按照要求进行处理，最后输出处理结果，通过前面的学习，我们已经熟悉如何使用 input()接收用户数据、print()输出处理结果了。但是如果想将信息保存下来，就需要将程序结果输出到文件，将来需要时从文件中输入信息。本节介绍如何打开文件、对文件读写以及关闭文件。

1. 打开文件

Python 提供了文件对象，并内置了 open()函数来获取一个文件对象。open()函数的使用：file_object = open(path,mode)。其中，file_object 是调用 open()函数后得到的文件对象，成功打开文件后，file_object 这个变量将一直代表这个文件，参数 path 是一个字符串，代表要打开文件的路径，一个完整的文件路径格式应该是这样的："盘符:/文件夹名/…/文件夹名/文件名"。mode 是打开文件的模式，常用的模式如表 3-6 所示。

表 3-6　打开文件时的常用模式

打开模式	解　　释
r	以只读方式打开：只允许对文件进行读操作，不允许写操作(默认方式)
w	以写方式打开：文件不为空时清空文件，文件不存在时新建文件
a	追加模式：文件存在则在写入时将内容添加到末尾
+	可读写模式，可添加到其他模式中使用

例如，要打开 F 盘下的 file1.txt 文件进行读取操作，需要使用 r 模式，实现如下：f = open("F:/file1.txt",'r')。之后对该文件的操作只需对得到的文件对象 f 使用文件对象提供的方法即可。这里要注意的是，我们打开文件的模式为 r，文件不存在时会报错。那么是不是文件不存在时，打开文件都会报错呢？其实不是，这与打开模式有关，我们可以试一下使用 w 模式打开一个不存在的文件时，程序不但不会报错，还会创建这个文件。表 3-7 给出了常用文件模式的一些使用细节。

表 3-7　文件打开模式使用细节

打开模式	简述	若欲操作的文件不存在	是否清空原有内容	注
r	只读	打开失败	否	默认打开方式，只能读取文件
w	只写	新建	是	打开时会清空文件
a		新建	否	只能在尾部写入
r+	读写	打开失败	否	写入时会覆盖原有位置内容
w+		新建	是	打开时会清空文件
a+		新建	否	只能在尾部写入

在使用 w 模式打开文件时，一定要特别注意，Python 返回文件对象时，会清空该文件！当我们打开文件后，获得了文件对象，就可以用文件对象提供的方法对文件进行操作了。表 3-8 给出了文件对象提供的常用方法，参数中的[]符号表示括号中的值可以传递，也可以不传递。

表 3-8　文件对象常用方法

字	方　　法	作用/返回	参数
1	f.close()	关闭文件：用 open()打开文件后使用 close 关闭	无
2	f.read([count])	读出文件：读出 count 字节。如果没有参数，则读取整个文件	[count]
3	f.readline()	读出一行信息，保存于 list：每读完一行，移至下一行开头	无
4	f.readlines()	读出所有行，保存在字符串列表中	无
5	f.truncate([size])	截取文件，使文件的大小为 size	[size]
6	f.write(string)	把 string 字符串写入文件	一个字符串
7	f.writelines(list)	把 list 中的字符串写入文件，是连续写入文件，没有换行	字符串 list

2. 读写文件

下面以对文件 file1.txt 的操作为例介绍文件读写的相关知识，该文件位于 F 盘，内容如下：

```
1 this is a test file
2 Python can easily read files
3 10 5 19 20 37
```

1）读文件——read()、readline()和 readlines()

read()函数是按字节（一个字符算一字节）读取，若不设置参数，会全部读取出来。注意，read()函数会读取出换行符'\n'。

```
#<程序：读取文件>
>>> f = open("F:/file1.txt",'r')
>>> f.read()                          #读出所有的内容
'1 this is a test file\n2 Python can easily read files\n3 10 5 19 20 37'
>>> f.close()
```

readline()函数用于在文件中读取一整行，如果文件中只有一行，则读取结果如<程序：读取文件 1>所示，注意，readline()函数同样会读取出换行符。

```
#<程序：读取文件 1>
>>> f = open("F:/file1.txt",'r')
>>> f.readline()
'1 this is a test file\n'          #输出内容，换行符也一并读取
>>> f.close()
```

如果文件中有多行，那么 readlines()函数会将读出的所有行保存在字符串列表中。见<程序：读取文件 2>。

```
#<程序：读取文件 2>
f = open("F:/file1.txt",'r')
>>> f.readlines()                     #将文件内容以列表的形式存放
['1 this is a test file\n', '2 Python can easily read files\n', '3 10 5 19 20 37']
                                      #输出内容
>>> f.close()
```

实例 1：读取文件内容

在打开文件 file1.txt 后，若想要读取该文件的内容，并打印出来，程序实现如下：

```
#<程序：读取文件>
f = open("F:/file1.txt",'r')
fls = f.readlines()
for line in fls:
    line = line.strip(); print (line)
f.close()
```

使用 readlines 方法后,返回一个 list,该 list 的每个元素为文件的一行信息。需要注意的是,文件的每行信息中其实都包括了最后的换行符"\n",readlines()函数会将换行符也读取出来,如果不对读取的信息做处理,同时再用 print()函数输出,因为 print()函数默认是换行的,所以最终的输出结果就是屏幕上输出的每行信息之间会空两行。所以可以对读取的每行字符串进行处理,通常需要使用 strip 方法将头尾的空白和换行符号等去掉。

2) 写文件——write()、writelines()和 print()

函数 write()的参数是一个字符串,通过该函数可以向文件中写入一行内容,见<程序:通过 write 函数写入一行>。

```
#<程序:通过 write 函数写入一行>
f = open("F:/newfile.txt",'w')
f.write("我喜欢使用 python 编程")
f.close()
```

打开 file.txt,可以看到写入的内容:

```
我喜欢使用 python 编程
```

注意,write()函数不会在写入的文本末尾添加换行符,因此如果写入多行时没有指定换行符,那么文件看起来可能不是我们所希望的那样,写入的两行内容会挤到一起,比如下面的例子:

```
#<程序:试图通过 write 函数写入多行>
f = open("F:/newfile.txt",'w')
f.write("我喜欢使用 python 编程")
f.write("python 有很多优点")
f.close()
```

程序的执行结果如下:

```
我喜欢使用 python 编程 python 有很多优点      #file.txt 文件内容
```

所以要让每个字符串都独占一行,需要在 write()语句中包含换行符,如<程序:通过 write 函数写入多行>所示。

```
#<程序:通过 write 函数写入多行>
f = open("F:/newfile.txt",'w')
f.write("我喜欢使用 python 编程\npython 有很多优点")
f.close()
#file.txt 文件内容如下
我喜欢使用 python 编程
python 有很多优点
```

writelines()函数可以把列表中的字符串写入文件,注意是连续写入文件,没有换行。见<程序:writelines 函数的使用>。

```
#<程序：writelines 函数的使用 >
L = ["abc","def"]
f = open("F:/newfile.txt",'w')
f.writelines(L)
f.close()

abcdef                          #file.txt 文件内容
```

3.5.1 节介绍了 print()函数，也提到了 print()函数的完整形式 print(objects,sep,end,file,flush)。print()函数不仅能够将信息输出到屏幕上，还可以通过传参的方式让 print()函数将信息输出到文件中。3.5.2 节介绍了 objects、sep、end 这 3 个参数，这里接着讲解 file 和 flush 参数。file 参数用于将文本输入某些对象中，可以是文件(注意文件的路径要写对)，也可以是数据流，等等，默认是输出到屏幕(即 sys.stdout)。见<程序：print 到文件中>，我们打开了文件 f，接着将字符 a 输出到文件 f 中。

```
#<程序：print 到文件中>
f = open("F:/newfile.txt",'r+')
print("a",file=f)              #将"a"输出到文件中
f.close()
```

flush 参数表示是否立刻将输出语句输入参数 file 指向的对象中，其值只能是 True 或 False，默认为 False。例如，如果只写了如下两行代码：

```
>>> f = open('abc.txt', 'w')
>>> print('a',file=f)
```

那么执行这两句之后可以看到 abc.txt 文件这时为空，里面并没有内容，只有执行 f.close()之后才会将内容写进文件中。而如果将语句改为：

```
>>> print('a',file=f,flush=True)
```

则执行完这一句之后就会看到文件里立刻出现了字符 a。

实例 2：将信息写入文件

实例 2 要将文件 file1.txt 中首字符为 3 的行中每个数字加起来，不包括 3，即将“10 5 19 20 37”相加；然后，将结果写入文件末尾。

分析：在利用 readlines()函数将文件中的每行字符串都存储到列表中后，需要遍历列表的每个元素，每个元素也就是文件中的一行，看哪一行是以 3 开头的，为此，可以用 split()函数将每行字符串按空格分解为每个元素不包含空格的 list。然后判断 list[0]是不是字符 3。然后需要计算该 list 从 1 号元素开始的所有元素的和。最后，需要将结果写回文件，所以，文件的打开方式应为“r+”。该程序的实现如下：

```
#<程序：读取文件，计算并写回>
f = open("F:/file1.txt",'r+');fls = f.readlines()
for line in fls:
    line = line.strip();lstr = line.split()
```

```
        if lstr[0] == '3':
            res = 0
            for e in lstr[1:]:
                res += int(e)
    f.write('\n4 ' + str(res));f.close()
```

需要注意的是,用 readlines()读取文件以及 split 分割字符串后,每个元素均为字符串。所以,要进行加法计算,首先需要将字符串转换为 int 类型。而在写入文件的时候,需要将 int 类型的 res 转换为字符串类型,执行程序后,文件 file1.txt 中的内容如下:

```
1 this is a test file
2 Python can easily read files
3 10 5 19 20 37
4 91
```

3. 关闭文件

我们将关闭文件单独列出来讲解,就是希望重点强调:在对文件操作完成时,不要忘记收尾工作,即关闭文件。在进行文件操作时,首先需要使用 open()打开文件,每次对文件操作完成后,不要遗忘 close()操作,将已经打开并操作完成的文件关闭。养成这个习惯可以避免程序出现很多奇怪的错误(bug)。事实上,每个进程打开文件的数量是有限的,每次系统打开文件后会占用一个文件描述符,而关闭文件时会释放这个文件描述符,以便系统打开其他文件。

3.5.4 异常处理

在 3.5.3 节中,我们所讲解的文件打开操作都是假设文件一定可以成功打开。然而,在实际应用中,很可能会出现无法打开文件的情况,比如找不到要打开的文件或者文件已经损坏无法打开等情况。如果文件无法打开,而我们又没有考虑到这种异常情况,仍然强行让程序继续执行,所导致的后果将难以想象,所以这个时候异常处理显得尤为重要。有了异常处理,就可以避免程序在运行中出现严重问题,从而保证代码的正确执行。

1. 什么是异常

以打开文件为例,先来看一下什么被认为是异常。在打开文件时,一种常见的问题就是找不到文件,比如要打开的文件在别的地方、文件名不对或者文件根本不存在。假设在计算机的 F 盘下有一个文件 file.txt,可以通过 f = open("F:/file.txt",'r')的方式打开,而如果把文件名写错了,像下面这种情况就会出错(见<程序:程序出现异常示例 1>),这种情况就算作一个异常。

```
#<程序:程序出现异常示例 1>
f = open("F:/file.txt",'r')            #文件名为 file1.txt,我们却写成 file.txt
#输出如下错误:
Traceback (most recent call last):
```

```
  File "<pyshell#8>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'F:/fil.txt'
```

相信大家在前面编写程序时经常会看到类似的错误，比如直接输出一个未定义的变量a时，程序也会出错，这同样算作一个异常。

```
#<程序：程序出现异常示例2>
>>> print(a)            #变量a未定义
#屏幕输出如下错误：
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    print(a)
NameError: name 'a' is not defined'
```

示例中的FileNotFoundError、NameError都是异常！也就是说，其实异常并非只有打开文件出错这一种，还包括调用未定义的变量、除法运算中除数为0等很多情况都算作异常，并且每种异常都有自己独特的名字。处理异常的模型是固定的，只要我们学会了运用该模型，那么当发生各种各样的异常问题时就可以迎刃而解。本节以一些常见的异常名称为例来讲解如何处理异常。

2. 用try语句处理异常

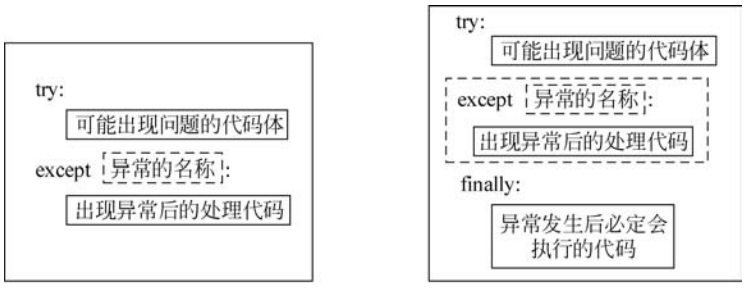
在Python中，异常算是一个事件，若该事件在程序执行的过程中发生，将会影响程序的正常执行。当编写的程序中发生了让Python不知所措的错误时，Python都会创建一个异常对象，比如<程序：程序出现异常示例1>中文件找不到时，FileNotFoundError就是一个异常对象，如果我们未对异常进行处理，那么程序将停止，并显示一个traceback(可以认为是错误的跟踪报告)，其中包含有关异常的报告。如果我们编写了处理异常的代码，捕捉到了这些错误并进行处理，程序就可以正常地继续执行，那么应该怎么捕获和处理异常呢？

异常捕获可以使用try语句来实现，任何出现在try语句范围内的异常都会被及时捕获到。try语句的形式有很多，这里介绍两种比较常见的形式：try-except和try-finally。其模型如图3-19(a)和(b)所示，其中虚线的方框表示这一部分可有可无。可以这样理解try语句的意思，当我们认为某一段代码可能出现问题时，就将其放到try语句的里面，意味着尝试去执行这段代码，如果确实发现这段代码有问题(若写了异常的名称，则还需要判断是否与我们所写的异常的名称相符)，则不去执行那段代码，直接说明异常，执行后续操作；而如果这段代码没有问题，则跳过except的语句块部分，继续执行后续操作。需要提醒大家的是，不同的异常有自己独特的名称，如果需要写在except语句后面添加异常名称，那么请正确书写对应的名称，这样才能做相应的处理异常操作。下面分别介绍一下这两种形式。

1) try-except模型

try-except模型执行方式如下：尝试执行try语句内可能出现问题的代码，如果发现确实出现了我们所写的异常，则执行except部分的处理代码，然后正常执行后面的代码；否则直接执行try语句中的代码段，然后正常执行后面的代码。

首先来看FileNotFoundError这一文件名称异常。在<程序：程序出现异常示例1>



(a) try-except模型　　(b) try-finally模型

图 3-19　异常捕获形式

的例子中，文件名字输入错误，导致文件找不到，在 traceback 中，最后一行报告了 FileNotFoundError 异常，这是 Python 找不到要打开的文件时创建的异常。在这个示例中，错误是函数 open()导致的，因此要处理这个错误，需要将 open 文件的语句放入 try 语句里面。

```
#<程序：try-except 示例 1>
try:
    f = open("F:/file.txt",'r')          #文件名为 file1.txt,我们却写成 file.txt
except FileNotFoundError:
    print("文件找不到!!")
```

该程序最终会输出：

```
文件找不到!!
```

在<程序：try-except 示例 1>中，try 代码块引发了 FileNotFoundError 异常，Python 会找出与该错误匹配的 except 代码块，并运行其中的代码。最终的结果是显示一条友好的错误信息——“文件找不到!!”，而不是上面的 traceback 信息，让人不太理解。当然使用异常处理的好处是除了比较友好之外，还有其他的方面，比如我们写好了代码(但是没有做异常处理)给用户使用，如果用户怀有恶意，故意让程序出现异常，然后根据看到的 traceback 中的信息，就可以推测出一些重要的代码，由此就可针对我们的代码进行攻击。

再来看另一个异常：NameError，这个异常表示变量名错误，可能正在访问一个未声明的变量。在<程序：try-except 示例 2>中，如果 file1.txt 确实存在，那么 open()函数正常返回文件对象，但异常却发生在成功打开文件后的 print(a)语句上，此时 Python 将直接跳到 except NameError，并输出提示。

```
#<程序：try-except 示例 2>
try:
    f = open("F:/file1.txt",'r')          #文件存在
    print(a)
except FileNotFoundError:
    print("文件找不到!!")
except NameError:
    print("变量未定义!!")
```

该程序最终会输出：

```
变量未定义!!
```

当然，如果你真的无法确定要对哪一类异常进行处理，只是希望一旦 try 语句块出错，就给用户一个“看得懂”的提醒，也可以这么做：

```
…
except:
    print("出错了!")
…
```

练习题 3.5.1　编写程序，输入一个字符串作为表达式，例如，字符串"2＋3"，用 eval 求该字符串的值。假如字符串有除以 0，则使用 try-except 处理异常。

【答案】　代码见<程序：处理除数有 0 的异常>。

```
#<程序：处理除数有 0 的异常>
try:
    s = input("请输入一个数学表达式：")
    print(eval(s))
except ZeroDivisionError:
    print("除数不可以为 0!")
```

2）try-finally 模型

为了实现在程序出现异常后，仍继续执行必要的收尾工作，比如在程序崩溃前，保存用户文档，Python 引入 try-finally 语句的处理模型，该模型中的 except 部分是可有可无的。try-finally 模型执行方式如下：尝试执行 try 语句内可能出现问题的代码，如果发现确实出现了异常，则执行 except 部分的处理代码，然后必须执行 finally 部分的代码，再去执行后面其他的代码；如果没有发现异常，则直接执行 try 语句中的代码段，跳过 except 部分，但仍旧要执行 finally 部分的代码，再去执行后面其他的代码。先看一个例子，见<程序：try-finally 示例 1>。

```
#<程序：try-finally 示例 1>
try:
    f = open("F:/file1.txt",'r')          #文件存在
    print(a)
except NameError:
    print("变量未定义!!")
finally:
    f.close()
```

该程序由于 try 语句前面没有定义变量 a，所以仍旧会输出：

```
变量未定义!!
```

该例中如果 try 语句块中没有出现错误，则会跳过 except 语句执行 finally 语句块的内

容,即关闭文件;如果出现异常,则会先执行 except 中的语句,再执行 finally 语句块中的内容。也就是说,无论异常是否被捕获,最后都会将文件关闭。

对于 try-finally 模型还有一点需要说明的是,其中的 except 语句部分是可有可无的,如果不写 except 语句部分,则如果发生异常,会输出 Traceback 信息,然后执行 finally 部分的语句。所以还可以写出<程序:try-finally 示例 2>中的 try-finally 语句。

```
#<程序:try-finally 示例 2>
try:
    f = open("F:/file.txt",'w')
    print(a)
finally:
    f.close()
```

那么 Python 通常还可能抛出哪些异常呢?这里给大家做个总结,见表 3-9,以后遇到这样的异常就不会感到陌生了。

表 3-9 常见异常描述

异 常	描 述
NameError	尝试访问一个不存在的变量
ZeroDivisionError	除数为 0
SyntaxError	语法错误
IndexError	索引超出序列范围
KeyError	请求一个不存在的字典关键字
OSError	操作系统产生的异常,就像打开一个不存在的文件会引发 FileNotFoundError,它就是 OSError 的子类
AttributeError	尝试访问未知的对象属性
TypeError	不同类型间的无效操作,比如 1+'1'

3. 对资源进行访问时还可以用 with 语句处理异常

在第 2 部分中介绍了用 try 语句来处理异常,有时我们需要知道异常的名字,才能准确写明 except 部分,告知用户是什么地方出错,并且对于像打开文件这种操作,为了保证安全性,还会用到 finally 语句来强制关闭文件(不管是否捕获到异常)。每次总是这样写难免会显得很烦琐。为此,Python 还提供了 with 语句:with 语句适用于对资源进行访问的场合,以确保不管使用过程中是否发生异常都会执行必要的“清理”操作,释放资源,比如文件使用后自动关闭、线程中锁的自动获取和释放等。with 语句的形式如下:

```
with 对资源的操作语句[as target(s)]:
    正常函数代码段
```

所以对于处理打开文件这种操作时产生的异常,可以用 with 语句来处理,它会自动调用 close 方法,见<程序:with 语句使用示例>。

```
#<程序：with 语句使用示例>
with open("F:/file.txt",'w') as f:
    f.write("Hello world!")
```

这种写法和使用 try-finally 关闭文件的效果一样，它会自动调用 close 方法，可以发现，这种写法使代码简洁了很多。

习题

习题 3.1　请利用所学知识，将下面的“不完美函数”改写成完美函数。

```
#<程序："不完美函数">
res
def add(a,b):
    a = a*b; res = a+b
add(2,3)
print("最终结果为：",res)
```

习题 3.2　改写本章 3.1.2 节<程序：参数与返回值举例>中的 find 函数，使其可以实现新的功能：查找序列中是否有字符 f，若有，则返回 True 与一个列表，列表中记录所有字符 f 所在的索引；若无，则返回 False 与空列表。

例如，对于'abeffestffe'；返回 True，[3,4,8,9]。

例如，对于[23,4,6,'e']；返回 False，[]。

习题 3.3　下面这个程序，将会输出什么？在 g-func()函数中哪些是局部变量？

```
#<程序：局部变量与全局变量举例>
b, c = 2, 4
def g_func(d):
    global a ; a = d*c
g_func(b) ; print(a)
```

习题 3.4　局部与全局变量练习。请分析<程序：四则运算例子>的执行过程，并说明输出结果。

```
#<程序：四则运算例子>
def do_div(a, b):
    c = a/b              #a、b、c 都是 do_div()中的局部变量
    print (c); return c
def do_mul(a, b):
    global c ; c = a*b#a, b 是 do_mul()的局部变量，c 是全局变量
    print (c) ; return c
def do_sub(a, b):
```

```
    c = a - b                    # a、b、c 都是 do_sub()中的局部变量
    c = do_mul(c, c)
    c = do_div(c, 2)
    print (c); return c
def do_add(a, b):                # 参数 a 和 b 是 do_add()中的局部变量
    global c
    c = a + b                    # 全局变量 c,修改了 c 的值
    c = do_sub(c, 1)             # 再次修改了全局变量 c 的值
    print (c)
# 所有函数外先执行:
a = 3                            # 全局变量 a
b = 2                            # 全局变量 b
c = 1                            # 全局变量 c
do_add(a, b)                     # 全局变量 a 和 b 作为参数传递给 do_add()
print (c)                        # 全局变量 c
```

习题 3.5 修改习题 3.4 中的<程序：四则运算例子>,去掉 do_add()中的 global c 语句,分析程序将会输出什么。

习题 3.6 嵌套函数中局部与全局变量的练习。分析<程序：嵌套函数局部与全局变量练习>,每个变量分别是局部变量还是全局变量？并说明打印结果。

```
#<程序：嵌套函数局部与全局变量练习>
a = 1;b = 2
def fun(x):
    def F():
        global a ; a = x + y + b
        return a
    y = 12 ; x = x + 2 ; a = F()
fun(b)
print("Finally, a is: %d and b is: %d" %(a,b))
```

习题 3.7 假设一个列表为 L,则 L.reverse()和 L[-1:-1-len(L):-1]的差别在哪里？

习题 3.8 假设一个列表为 L,我们知道 L.remove(x)是除去 L 中第一个值为 x 的元素,那么要除去 L 中所有是 x 的元素,要怎么办？

习题 3.9 如何用 L.insert(i,x)实现 L.append(x)？

习题 3.10 利用 for 循环将一个字符串列表双重倒转。给定一个字符串列表,将整个序列倒转,同时每个字符串元素也要倒转,输出倒转后的列表。

比如 L=['It is','very very','funny','!'],则完全倒转的结果为 L_new=['!','ynnuf','yrev yrev','si tI']。

习题 3.11 输入一个字符串,内容是个带小数的实数,如 123.45,输出是两个整数变量 x 和 y,x 是整数部分 123,y 是小数部分 45。可以用 split()函数来完成。

习题 3.12 字典字符串练习 1。实现一个函数,该函数功能为：删除字符串中出现次数最少的字符,若多个字符出现次数一样,则都删除。输出删除这些单词后的字符串,字符串中其他字符保持原来的顺序。

输入：字符串只包含小写英文字母，不考虑非法输入，输入的字符串长度小于或等于20字节（如abcdd）。

输出：删除字符串中出现次数最少的字符后的字符串（如dd）。

习题 3.13　字典字符串练习2。实现一个函数，该函数功能为：假设一篇文章已经存储于一个字符串S中，统计S中每个单词出现的次数（注意单词后面的标点符号问题）。

习题 3.14　参数传递问题练习。现有一个Sum()函数，该函数可以求得输入的数字列表L中所有偶数的和，程序如<程序：参数传递问题练习>所示。请分析该程序，原列表L是否被修改？说明打印结果是什么。请尝试用多种方法修改程序，使得原列表L不会被修改。

```
#<程序：参数传递问题练习>
def Sum(L):
    mysum = 0;i = len(L) - 1
    while i >= 0:
        if L[i] % 2 == 0:
            mysum += L.pop(i);
        i = i - 1;
    return mysum
L = [2,2,3,4,5];mysum = Sum(L)
print(L,mysum)
```

习题 3.15　默认参数练习。请分析<程序：默认参数练习>中的代码，说明每次的打印结果是什么，当前的默认参数是什么。

```
#<程序：默认参数练习>
def append_1(L = [1,2]):
    if L[0] % 2 == 1: L.append(0)
    else: L.append(5)
    return(L)
print(append_1())
print(append_1([2]))
print(append_1([3]))
print(append_1())
```

习题 3.16　用列表推演表达式生成九九乘法表，每个元素都是一个计算式子。使得输出的列表为：L=['1*1=1','1*2=2','1*3=3','1*4=4','1*5=5','1*6=6','1*7=7','1*8=8','1*9=9','2*2=4','2*3=6','2*4=8','2*5=10','2*6=12','2*7=14','2*8=16','2*9=18','3*3=9','3*4=12','3*5=15','3*6=18','3*7=21','3*8=24','3*9=27','4*4=16','4*5=20','4*6=24','4*7=28','4*8=32','4*9=36','5*5=25','5*6=30','5*7=35','5*8=40','5*9=45','6*6=36','6*7=42','6*8=48','6*9=54','7*7=49','7*8=56','7*9=63','8*8=64','8*9=72','9*9=81']。

习题 3.17　用列表推演表达式完成如下功能：给定一个字符串text，里面存放的是一小段文本。请利用列表推演表达式获取文本中所有单词的第1个字符。text="My house

is full of flowers"。

习题 3.18 文件操作练习。请生成九九乘法表,并按照表的形式输出到文件中,格式如下:

```
1*1=1
1*2=2  2*2=4
1*3=3  2*3=6  3*3=9
1*4=4  2*4=8  3*4=12  4*4=16…
```

习题 3.19 从文件中以字典的形式读取数据,名字作为 key,年龄作为 value。文件中的内容如下,以制表符('\t')分割数据。

```
name      age
Aaron     34
Abraham   23
Andy      56
Benson    41
```

然后输出到另一个文件中,并添加行号。格式如下,依旧以制表符分割数据。

```
1    Aaron     34
2    Abraham   23
3    Andy      56
4    Benson    41
```

习题 3.20 异常处理练习。假设输入一组任意长度的列表,我们要对该列表中第 10 个元素进行加 1 操作,请利用 try-except 模型自己实现一个异常处理,可以捕获 IndexError 异常。

第4章 探究递归求解的思维方式

引 言

用递归(recursion)思维解决问题是计算机科学中最美的部分之一。递归的基本概念就是将大问题分解成“同质”的小问题,然后用小问题的解组合成大问题的解。很多问题通过递归的方式求解,程序都会变得简洁而优美,更加快速而准确,所以对递归思维的熟练运用是编程修炼的重中之重。

本章首先通过几个简单的例子带领大家理解递归求解问题的思维方式;其次,使用递归思维重温之前章节的例题,让大家熟悉如何通过编写递归程序来解题;再次,分别使用非递归与递归的方式实现列表和字符串的专有方法,在对比中认识到递归思维以及递归程序的优点;最后,以排序为例,分别用递归方式实现4种不同的排序算法,让大家在对比中进一步熟悉递归思维,也从中认识到算法有优劣之分,应尽可能地设计优化的算法解决问题。