第3章

函　　数

函数是组织好的，可重复使用的，用来实现单一或相关联功能的代码段。函数能提高应用的模块性和代码的重复利用率。通过前面章节的学习，我们已经了解了很多 Python 内置函数，通过使用这些内置函数可给编程带来很多便利，提高开发程序的效率。除了使用 Python 内置函数，也可以根据实际需要定义符合我们要求的函数，这称为用户自定义函数。

3.1　为什么要用函数

通过前面章节的学习，我们已经能够编写一些简单的 Python 程序了。但如果程序的功能比较多，规模比较大，把所有的代码都写在一个程序文件里，就会使文件中的程序变得庞杂，使人们阅读和维护程序变得困难。此外，有时程序中要多次实现某一功能，就要多次重复编写实现此功能的程序代码，这会使程序冗长，下面通过举例来进一步说明这个问题。

假如需要计算三个长方形的面积和周长，这三个长方形的长和宽分别是 18 和 12、27 和 14、32 和 26。如果创建一个程序来对这三个长方形求面积和周长，可能编写如下所示的代码：

```
length=18
width=12
area=length*width
perimeter=2*length+2*width
print('长为%d、宽为%d的长方形的面积为%d，周长为%d'%(length, width, area,
perimeter))
length=27
width=14
area=length2*width2
perimeter=2*length+2*width
print('长为%d、宽为%d的长方形的面积为%d，周长为%d'%(length, width, area,
perimeter))
length=32
width=26
```

```
area=length * width
perimeter=2 * length+2 * width
print('长为%d, 宽为%d 的长方形的面积为%d, 周长为%d'%(length, width, area,
perimeter))
```

上述代码在 IDLE 中运行的结果如下：

```
长为 18、宽为 12 的长方形的面积为 216, 周长为 60
长为 27、宽为 14 的长方形的面积为 378, 周长为 82
长为 32、宽为 26 的长方形的面积为 832, 周长为 116
```

从上述三段代码可以看出，这三段代码除了开始和结束的数字不同，其他都非常相似。这三段基本相同的代码是否能够只写一次呢？对于这样的问题，我们可以使用函数来解决，使计算长方形面积和周长的这段代码得以重用。上面的代码使用函数后，可简化成下面所示的代码：

```
1. def rectangle(length,width):
2.     area=length * width
3.     perimeter=2 * length+2 * width
4.     print('长%d、宽%d 的长方形面积为%d,周长为%d'%(length,width,area, perimeter))
5. def main():
6.     rectangle(18,12)
7.     rectangle(27,14)
8.     rectangle(32,26)
9. main()     #调用 main()函数
```

上述代码在 IDLE 中运行的结果与前面三段代码运行的结果相同。

在第 1 行～第 4 行定义了带两个参数 length 和 width 的 rectangle()函数。第 5 行～第 8 行定义了 main()函数，它通过 rectangle(18,12)、rectangle(27,14)和 rectangle(32,26)调用 rectangle()函数分别计算长和宽分别是 18 和 12、27 和 14、32 和 26 的长方形的面积和周长。第 9 行调用了 main()函数。

从本质意义上来说，函数是用来完成一定的功能的。函数可看作是实现特定功能的小方法或小程序。函数可简单地理解成：你编写了一些语句，为了方便重复使用这些语句，把这些语句组合在一起，给它起一个名字，使用时只要调用这个名字，就可以实现这些语句的功能。另外，每次使用函数时可以提供不同的参数作为输入，以便对不同的数据进行处理；函数处理后，还可以将相应的结果反馈给我们。在前面章节中，我们已经学习了像 range(a,b)、int(x)和 abs(x)这样的函数。当调用 range(a,b)函数时，系统就会执行该函数里的语句并返回结果。

3.2　怎样定义函数


怎样定义函数

在 Python 中，程序中用到的所有函数必须“先定义，后使用”。例如，想用 rectangle()函数去求长方形的面积和周长，必须事先按 Python

函数规范对它进行定义，指定函数的名称、参数、函数实现的功能、函数的返回值。在 Python 中定义函数的语法格式如下：

```
def 函数名([参数列表]):
    '''注释'''
    函数体
```

在 Python 中使用 def 关键字来定义函数，定义函数时需要注意以下几个事项。

(1) 函数代码块以 def 关键词开头，代表定义函数。

(2) def 之后是函数名，由用户自己指定，def 和函数名中间至少要敲一个空格。

(3) 函数名后跟括号，括号后要加冒号，括号内用于定义函数参数，称为形式参数，简称形参，参数是可选的，函数可以没有参数。如果函数有多个参数，参数之间用逗号隔开。参数就像一个占位符，当调用函数时，就会将一个值传递给参数，这个值被称为实际参数或实参。在 Python 中，函数形参不需要声明其类型。

(4) 函数体，指定函数应当完成什么操作，是由语句组成，要有缩进。

(5) 如果函数执行完之后有返回值，称为带返回值的函数，函数也可以没有返回值。带有返回值的函数，需要使用以关键字 return 开头的返回语句来返回一个值，执行 return 语句意味着函数执行的终止。函数返回值的类型由 return 后要返回的表达式的值的类型决定，表达式的值是整型，函数返回值的类型就是整型；表达式的值是字符串，函数返回值的类型就是字符串类型。

(6) 在定义函数时，开头部分的注释通常描述函数的功能和参数的相关说明，但这些注释并不是定义函数时必需的，可以使用内置函数 help()来查看函数开头部分的注释内容。

下面定义一个找出两个数中较小的函数。这个函数被命名为 min，它有两个参数：num1 和 num2，函数返回这两个数中较小的那个。图 3-1 解释了函数的组件及函数的调用。

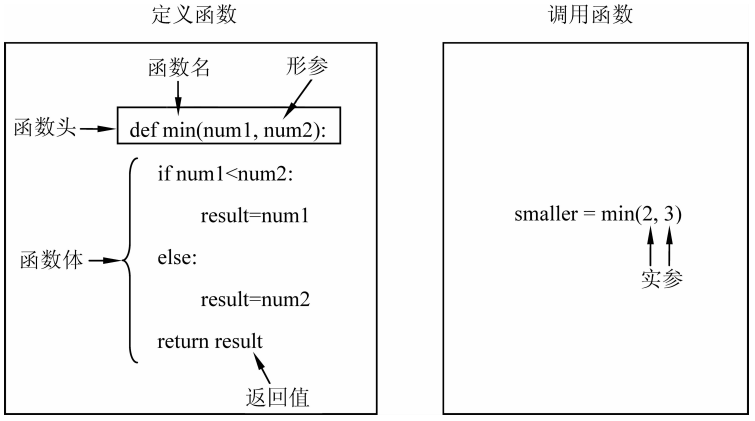

图 3-1　函数的组件及函数的调用

Python 允许嵌套定义函数，即在一个函数中定义另外一个函数。内层函数可以访问

外层函数中定义的变量，但不能重新赋值，内层函数的局部命名空间不能包含外层函数定义的变量。嵌套函数定义举例如下：

```
def f1():                    #定义函数 f1
   m=3                       #定义变量 m=3
   def f2():                 #在 f1 内定义函数 f2
      n=4                    #定义局部变量 n=4
      print(m+n)
   f2()                      #在 f1 函数内调用函数 f2
f1()                         #调用 f1 函数
```

上述程序代码在 IDLE 中运行的结果如下：

```
7
```

3.3　函数调用

在函数定义中，定义了函数的功能，即定义了函数要执行的操作。要使函数发挥功能，必须调用函数，调用函数的程序被称为调用者。调用函数的方式是**函数名(实参列表)**，实参列表中的参数个数要与形参个数相同，参数类型也要一致。当程序调用一个函数时，程序的控制权就会转移到被调用的函数上。当执行完函数的返回值语句或执行到函数结束时，被调用函数就会将程序控制权交还给调用者。根据函数是否有返回值，函数调用有两种方式，即带有返回值的函数调用和不带返回值的函数调用。

3.3.1　带返回值的函数调用

对这种函数的调用通常当作一个值处理，如下所示。

```
smaller=min(2, 3)     #这里的 min()函数指的是图 3-1 里面定义的函数
```

smaller＝min(2,3)语句表示调用 min(2,3)，并将函数的返回值赋值给变量 smaller。另外一个把函数当作值处理的调用函数的例子为

```
print(min(2, 3))
```

这条语句将调用函数 min(2,3)后的返回值输出。

【例 3-1】　简单的函数调用。(3-1.py)

```
def fun():                   #定义函数
   print('简单的函数调用 1')
   return  '简单的函数调用 2'
a=fun()                      #调用函数 fun
print(a)
```

3-1.py 在 IDLE 中运行的结果如下：

```
简单的函数调用 1
```

```
简单的函数调用 2
```

注意：即使函数没有参数，调用函数时也必须在函数名后面加上括号，只有见到这个括号，才会根据函数名从内存中找到函数体，然后执行它。

【例 3-2】 函数的执行顺序。(3-2.py)

```
def fun():
  print('第一个 fun()函数')
def fun():
  print('第二个 fun()函数')
fun()
```

3-2.py 在 IDLE 中运行的结果如下：

```
第二个 fun()函数
```

从上述执行结果可以看出，fun()调用函数时执行的是第二个 fun()函数，下面的 fun()函数将上面的 fun()函数覆盖掉了，也就是说程序中如果有多个同函数名、同参数的函数，调用函数时只有最近的函数发挥作用。

在 Python 中，**一个函数可以返回多个值**。下面的程序定义了一个输入两个数并以升序返回这两个数的函数。

```
>>>def sortA(num1, num2):
    if num1<num2:
        return num1,num2
    else:
        return num2, num1
>>>n1, n2=sortA(2, 5)
>>>print('n1 是', n1, '\nn2 是', n2)
n1 是 2
n2 是 5
```

sortA()函数返回两个值，当它被调用时，需要用两个变量同时接收函数返回的两个值。

【例 3-3】 包含程序主要功能的名为 main 的函数。(TestSum.py)

下面的程序文件用于求两个整数之间的整数和。

```
1. def sum(num1, num2):                                  #定义 sum()函数
2.     result=0
3.     for i in range(num1, num2+1):
4.         result+=i
5.     return result
6. def main():                                           #定义 main()函数
7.     print("Sum from 1 to 10 is", sum(1, 10))          #调用 sum()函数
8.     print("Sum from 11 to 20 is", sum(11, 20))        #调用 sum()函数
9.     print("Sum from 21 to 30 is", sum(21, 30))        #调用 sum()函数
```

```
10. main()                                          #调用 main()函数
```

TestSum. py 在 IDLE 中运行的结果如下：

```
Sum from 1 to 10 is 55
Sum from 11 to 20 is 155
Sum from 21 to 30 is 255
```

这个程序文件包含 sum()函数和 main()函数，在 Python 中 main()函数也可以写成其他任何合适的标识符。程序文件在第 10 行调用 main()函数。习惯上，程序里通常定义一个包含程序主要功能的名为 main()的函数。

这个程序的执行流程：解释器从 TestSum. py 文件的第一行开始一行一行地读取程序语句，读到第 1 行的函数头时，将函数头以及函数体(第 1～5 行)存储在内存中。然后，解释器将 main()函数的定义(第 6～9 行)读取到内存。最后，解释器读取到第 10 行时，调用 main()函数，main()函数中的语句被执行。程序的控制权转移到 main()函数，main()函数中的三条 print 输出语句分别调用 sum()函数求出 1～10、11～20、21～30 的整数和并将计算结果输出。TestSum. py 中函数调用的流程图如图 3-2 所示。

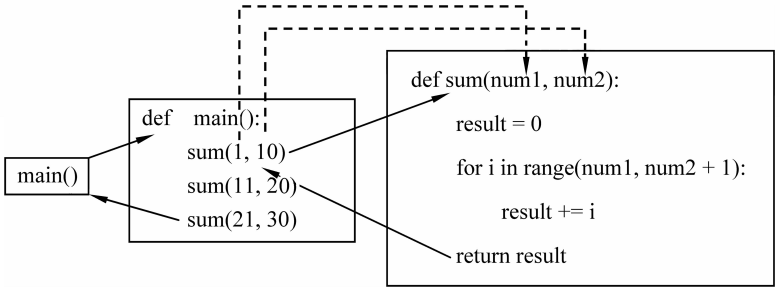

图 3-2　TestSum. py 中函数调用的流程图

注意：这里的 main()函数定义在 sum()函数之后，但也可以定义在 sum()函数之前。在 Python 中，函数在内存中被调用，在调用某个函数之前，该函数必须已经调入内存，否则会出现函数未被定义的错误。也就是说，在 Python 中不允许前向引用，即在函数定义之前，不允许调用该函数，下面进一步举例说明。

```
print(printhello())          #在函数 printhello()定义之前调用该函数
def printhello():            #定义 printhello()函数
    print('hello')
```

上述代码在 IDLE 中运行，出现运行错误，具体错误如下：

```
Traceback (most recent call last):
  File "C:\Users\cao\Desktop\test FunctionCall.py", line 1, in <module>
    print(printhello())
NameError: name 'printhello' is not defined
```

3.3.2 不带返回值的函数调用

如果函数没有返回值，对函数的调用是通过将函数调用当作一条语句来实现的，如下面含有一个形式参数的输出字符串的函数的调用。

```
>>>def printStr(str1):
  "打印任何传入的字符串"
  print(str1)

>>>printStr('hello world')     #调用函数 printStr(),将'hello world'传递给形参
hello world
```

另外也可将要执行的程序保存成 file. py 文件，打开 cmd，将路径切换到 file. py 文件所在的文件夹，在命令提示符后输入 file. py，按 Enter 键就可执行。

3.4 函数参数传递

在 Python 中，数字、元组和字符串对象是不可更改的对象，而列表、字典对象是可以修改的对象。Python 中一切都是对象，严格意义上说，调用函数时的参数传递不能说是值传递或引用传递，应该说是传可变对象或传不可变对象。因此，函数调用时传递的参数类型分为可变类型和不可变类型。

不可变类型：若 a 是数字、字符串、元组这三种类型中的一种，则函数调用 fun(a)时，传递的只是 a 的值，在 fun(a)内部修改 a 的值，只是修改另一个复制的对象，不会影响 a 本身。

可变类型：若 a 是列表、字典这两种类型中的一种，则函数调用 fun(a)时，传递的是 a 所指的对象，在 fun(a)内部修改 a 的值，fun(a)外部的 a 也会受影响。例如：

```
>>>b=2
>>>def changeInt(x):
    x=2*x
    print(x)
>>>changeInt(b)
4
>>>b
2                        #changeInt(b)外的 b 没发生变化

>>>c=[1, 2, 3]
>>>def changeList(x):
    x.append([4, 5, 6])
    print(x)
>>>changeList(c)
[1, 2, 3, [4, 5, 6]]
>>>c
```

```
[1, 2, 3, [4, 5, 6]]    #changeList(c)外的 c 发生了变化
```

3.5　函数参数的类型

函数的作用在于它处理参数的能力，当调用函数时，需要将实参传递给形参。函数参数的使用可以分为两个方面：一是函数形参是如何定义的，二是函数在调用时实参是如何传递给形参的。在 Python 中，定义函数时不需要指定参数的类型，形参的类型完全由调用者传递的实参本身的类型来决定。函数形参的表现形式主要有位置参数、关键字参数、默认值参数、可变长度参数和序列解包参数。

3.5.1　位置参数

位置参数函数的定义方式为 functionName(参数 1，参数 2，…)。调用位置参数形式的函数时，是根据函数定义的参数位置来传递参数的，也就是说在给函数传参数时，按照顺序，依次传值，要求实参和形参的个数必须一致。下面举例说明：

```
>>>def print_person(name, sex):
    sex_dict={1:'先生',2:'女士'}
    print('来人的姓名是%s,性别是%s'%(name,sex_dict[sex]))
```

上面定义的 print_person(name，sex)函数中，name 和 sex 这两个参数都是位置参数，调用的时候，传入的两个值按照顺序，依次赋值给 name 和 sex。

```
>>>print_person('李明', 1)              #必须包括两个实参,第一个是姓名,第二个是性别
来人的姓名是李明,性别是先生
```

通过 print_person('李明',1)调用该函数，则'李明'传递给 name，1 传递给 sex，实参与形参的含义要相对应，即不能颠倒'李明'和 1 的顺序。

3.5.2　关键字参数

关键字参数主要指调用函数时的参数传递方式，关键字参数用于函数调用，通过“键-值”形式加以指定。使用关键字参数调用函数时，是按参数名字传递实参值，关键字参数的顺序可以和形参顺序不一致，不影响参数值的传递结果，避免了用户需要牢记参数位置和顺序的麻烦。例如：

```
>>>print_person(name='李明', sex=1)  #name、sex 为定义函数时函数的形参名
来人的姓名是李明,性别是先生
>>>print_person(sex=1,name='李明')
来人的姓名是李明,性别是先生
```

3.5.3　默认值参数

在定义函数时，Python 支持默认值参数，即在定义函数时为形参设置默认值。在调

用设置了默认值参数的函数时，可以通过显示赋值来替换其默认值，如果没有给设置了默认值的形参传递实参，这个形参就将使用函数定义时设置的默认值。定义带有默认值参数的函数的语法格式如下：

```
def functionName(…, 参数名=默认值):
    函数体
```

可以使用“函数名.__defaults__”查看函数所有默认值参数的当前值，其返回值为一个元组，其中的元素依次表示每个默认值参数的当前值，带默认值参数的函数举例如下：

```
>>>def add(x, y=5):                    #定义带默认值参数的函数，y 为默认值参数，默认值为 5
    return(x+y)

>>>add.__defaults__
(5,)
```

通过 add(6,8)调用该函数，表示将 6 传递给 x，8 传递给 y，y 不再使用默认值 5。此外，add(9)这个形式也是可以的，表示将 9 传递给 x，y 取默认值 5。

```
>>>add(6, 8)
14
>>>add(9)
14
```

注意：*在定义带有默认值参数的函数时，默认值参数必须出现在函数形参列表的最右端，其任何一个默认值参数右边都不能再出现非默认值参数。*

3.5.4 可变长度参数

定义带有可变长度参数的函数的语法格式如下：

```
functionName(arg1, *tupleArg, **dictArg))
```

tupleArg 和 dictArg 称为可变长度参数。tupleArg 前面的 * 表示这个参数是一个元组参数，用来接收任意多个实参并将其放在一个元组中。dictArg 前面的**表示这个参数是字典参数(“键:值”对参数)，用来接收类似于关键字参数一样显示赋值形式的多个实参并将其放入字典中。可以把 tupleArg、dictArg 看成两个默认参数，调用带有可变长度参数的函数时，多余的非关键字参数放在元组参数 tupleArg 中，多余的关键字参数放字典参数 dictArg 中。

下面的程序演示了第一种形式的可变长度参数的用法，即无论调用该函数时传递了多少个实参，统统将其放入元组中。

```
>>>def f(*x):
    print(x)
>>>f(1,2,3)
(1, 2, 3)
```

下面的代码演示了第二种形式可变长度参数的用法，即在调用该函数时自动接收关键字参数形式的实参，将其转换为“键:值”对放入字典中。

```
>>>def f(**x):
    print(x)
>>>f(x='Java',y='C',z='Python')
{'x': 'Java', 'y': 'C', 'z': 'Python'}
>>>f(a=1,b=3,c=5)
{'a': 1, 'b': 3, 'c': 5}
```

下面的代码演示了几种不同形式参数的混合使用：

```
>>>def varLength(arg1, *tupleArg, **dictArg):
    print("arg1=", arg1)
    print("tupleArg=", tupleArg)
    print("dictArg=", dictArg)
>>>varLength("Python")
arg1=Python       #表明函数定义中的 arg1 是位置参数
tupleArg=()       #表明函数定义中的 tupleArg 的数据类型是元组
dictArg={}        #表明函数定义中的 dictArg 的数据类型是字典
>>>varLength('hello world','Python',a=1)
arg1=hello world
tupleArg=('Python',)
dictArg={'a': 1}
>>>varLength('hello world','Python','C',a=1,b=2)
arg1=hello world
tupleArg=('Python', 'C')
dictArg={'a': 1, 'b': 2}
```

3.5.5　序列解包参数

序列解包参数主要指调用函数时参数的传递方式，与函数定义无关。使用序列解包参数调用的函数通常是一个位置参数函数。序列解包参数由一个 * 和序列连接而成，Python 解释器自动将序列解包成多个元素，并一一传递给各个位置参数。

创建列表、元组、集合、字典以及其他可迭代对象，称为“序列打包”，因为值被“打包到序列中”。序列解包是指将多个值的序列解开，然后放到变量的序列中。下面用序列解包的方法将一个元组的三个元素同时赋给三个变量，注意变量的数量和序列元素的数量必须一样多。

```
>>>x, y, z=(1,2,3)                          #元组解包赋值
>>>print('x:%d, y:%d, z:%d'%(x, y, z))
x:1, y:2, z:3
>>>list1=['春', '夏', '秋', '冬']           #list1 中有 4 个元素
>>>Spring, Summer, Autumn, Winter=list1     #列表解包赋值
>>>print(Spring, Summer, Autumn, Winter)
```

```
春 夏 秋 冬
```

如果变量个数和元素的个数不匹配，就会出现错误：

```
>>>Spring, Summer, Autumn=list1          #变量的个数小于 list1 中元素的个数
Traceback(most recent call last):
  File "<pyshell#79>", line 1, in <module>
    Spring, Summer, Autumn=list1
ValueError: too many values to unpack (expected 3)
>>>dict1={"one":1,"two":2,"three":3}
>>>x,y,z=dict1                          #字典解包默认的是解包字典的键
>>>print(x,y,x)
one two one
>>>x1,y1,z1=dict1.items()          #用字典对象的 items()方法解包字典的"键:值"对
>>>print(x1,y1,x1)
('one', 1) ('two', 2) ('one', 1)
```

下面举例说明调用函数时的序列解包参数的用法：

```
>>>def print1(x, y, z):
    print(x, y, z)
>>>tuple1=('姓名', '性别', '籍贯')
>>>print1(*tuple1)     #调用函数时，*将 tuple1 解开成 3 个元素并分别赋给 x、y、z
姓名 性别 籍贯
>>>print(*[1, 2, 3])   #调用 print 函数，将列表[1, 2, 3]解包输出
1 2 3
>>>range(*(1,6))       #将(1,6)解包成 range 函数的两个参数
range(1, 6)
```

3.6 函数模块化

在程序中定义函数可以用来减少冗余的代码并提高代码的可重用性。但当程序中的代码逐渐变得庞大时，你可能想要把它分成几个文件，以便能够更简单地维护。同时，你希望在一个文件中写的代码能够被其他文件所重用，这时应该使用模块。在 Python 中，一个.py 文件就构成一个模块。可以把多个模块，即多个.py 文件，放在同一个文件夹中，构成一个包(Package)。

在 Python 中，可以将函数的定义放在一个模块中，然后，将模块导入到其他程序中，这些程序就可以使用模块中定义的函数。通常，一个模块可以包含多个函数，但同一个模块中的函数名不允许相同。例如 random、math 都是定义在 Python 库里的模块，这样，它们可以被导入到任何一个 Python 程序中，被这个 Python 程序使用。

```
>>>import platform          #将整个 platform 模块导入
>>>s=platform.platform()    #使用 platform 的 platform()方法查看操作平台信息
>>>print(s)
```

```
Windows-7-6.1.7601-SP1      #计算机不同,输出的信息可能不同
>>>import time as t          #导入模块 time,并将模块 time 重命名为 t
>>>t.ctime()                 #获取当前的时间
'Sat Feb  3 22:37:10 2018'
>>>from math import sqrt     #把 math 模块里的函数 sqrt 导入到当前模块里
>>>sqrt(4)      #这时可以直接调用 sqrt()函数求 4 的平方根,而不用再使用 math.sqrt()
2.0
```

下面编写一个求两个整数的最小公倍数的函数 lcm(x, y),并将其放在 LCMFunction.py 模块中。

```
def lcm(x, y):
   #获取最大的数
   if x>y:
       greater=x
   else:
       greater=y
   while(True):
       if((greater %x==0) and (greater %y==0)):
           lcm=greater
           break
       greater+=1
   return lcm
```

现在,编写一个独立的程序使用 lcm()函数,如下面的程序 TestLCMFunction.py 所示。

```
from LCMFunction import lcm          #导入 lcm()函数
num1=eval(input('请输入第一个整数: '))
num2=eval(input('请输入第二个整数: '))
print(num1,'和',num2,'的最小公倍数是', lcm(num1,num2))
===========RESTART: D:/Python/TestLCMFunction.py===========
请输入第一个整数: 24
请输入第二个整数: 54
24 和 54 的最小公倍数是 216
```

第一行从模块 LCMFunction 中导入 lcm()函数,这样,就可以在程序中调用 lcm()函数(第 4 行)。也可以使用下面的语句导入它:

```
import LCMFunction
```

如果使用这条语句,必须使用 LCMFunction. lcm()才能调用函数 lcm()。

将求最小公倍数的代码封装在函数 lcm()中,并将函数 lcm()封装在模块中,从这样的程序组织方式中可以看到模块化具备以下几个优点。

(1) 它将计算最小公倍数的代码与其他代码分隔开,使程序的逻辑更加清晰、程序的可读性更强,大大提高了代码的可维护性。

(2) 编写代码不必从零开始。当一个模块编写完毕,就可以被其程序引用。我们在编写程序的时候,也经常引用其他模块,包括 Python 内置的模块和来自第三方的模块。

(3) 使用模块还可以避免函数名和变量名冲突。相同名字的函数和变量完全可以分别存在不同的模块中。

3.7 lambda 表达式


lambda 表达式

Python 使用 lambda 表达式来创建匿名函数,即没有函数名字的临时使用的小函数。lambda 表达式的主体是**一个表达式**,而不是一个代码块,但在表达式中可以调用其他函数,并支持默认值参数和关键字参数,表达式的计算结果相当于函数的返回值。lambda 表达式拥有自己的名字空间,且不能访问自有参数列表之外或全局名字空间里的参数。可以直接把 lambda 定义的函数赋值给一个变量,用变量名来表示 lambda 表达式所创建的匿名函数。

lambda 表达式的语法格式如下:

```
lambda [参数 1 [,参数 2,…,参数 n]]:表达式
```

可以看出 lambda 表达式的一般形式:关键字 lambda 后面敲一个空格,后跟一个或多个参数,紧跟一个冒号,之后是一个表达式。冒号前是参数,冒号后是返回值。lambda 表达式返回一个值。

单个参数的 lambda 表达式:

```
>>>g=lambda x:x*2
>>>g(3)
6
```

多个参数的 lambda 表达式:

```
>>>f=lambda x,y,z:x+y+z          #定义一个 lambda 表达式,求三个数的和
>>>f(1,2,3)
6
#创建带有默认值参数的 lambda 表达式
>>>h=lambda x, y=2, z=3 : x+y+z
>>>print(h(1, z=4, y=5))
10
```

3.7.1 lambda 和 def 的区别

(1) def 创建的函数是有名称的,而 lambda 创建的函数是匿名函数。

(2) lambda 会返回一个函数对象,但这个对象不会赋给一个标识符;而 def 则会把函数对象赋值给一个标识符,这个标识符就是定义函数时的函数名,下面举例说明。

```
>>>def f(x,y):
    return x+y
```

```
>>>a=f
>>>a(1,2)
3
```

(3) lambda 只是一个表达式,而 def 则是一个语句块。

正是由于 lambda 只是一个表达式,它可以直接作为 Python 列表或 Python 字典的成员,例如:

```
info=[lambda x: x*2, lambda y: y*3]
```

在这个地方没有办法用 def 语句直接代替,因为 def 是语句,不是表达式,不能嵌套在里面。lambda 表达式中":"后只能有一个表达式,包含 return 返回语句的 def 可以放在 lambda 表达式中":"后面,不包含 return 返回语句的不能放在 lambda 表达式中":"后面。因此,像 if 或 for 或 print 这种语句就不能用于 lambda 中,lambda 一般只用来定义简单的函数。

```
>>>def multiply(x,y):
    return x*y
>>>f=lambda x,y:multiply(x,y)
>>>f(3,4)
12
```

lambda 表达式常用来编写带有行为的列表或字典,例如:

```
>>>L=[(lambda x: x**2),
    (lambda x: x**3),
    (lambda x: x**4)]
>>>print(L[0](2), L[1](2), L[2](2))
4 8 16
```

列表 L 中的三个元素都是 lambda 表达式,每个表达式是一个匿名函数,一个匿名函数表达一个行为,下面是带有行为的字典举例。

```
>>>D={'f1':(lambda x, y: x+y),
    'f2':(lambda x, y: x-y),
    'f3':(lambda x, y: x * y)}
>>>print(D['f1'](5, 2),D['f2'](5, 2),D['f3'](5, 2))
7 3 10
```

lambda 表达式可以嵌套使用,但是从可读性的角度来说,应尽量避免使用嵌套的 lambda 表达式。

map()函数可以将 lambda 表达式映射到一个序列上,将 lambda 表达式依次作用到序列的每个元素上。

map()函数接收两个参数:一个是函数,另一个是序列。map()将传入的函数依次作用到序列的每个元素上,并以 map 对象的形式返回作用后的结果。

```
>>>def f(x):
```

```
    return x * 2
>>>L=[1, 2, 3, 4, 5]
>>>list(map(f, L))
[2, 4, 6, 8, 10]
>>>list(map((lambda x: x+5),L))          #对列表 L 中的每个元素加 5
[6, 7, 8, 9, 10]
>>>list(map(str,[1,2,3,4,5,6,7,8,9]))    #将一个整型列表转换成字符串类型的列表
['1', '2', '3', '4', '5', '6', '7', '8', '9']
```

lambda 表达式可以用在列表对象的 sort()方法中。

```
>>>import random
>>>data=list(range(0, 20, 2))
>>>data
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
>>>random.shuffle(data)
>>>data
[2, 12, 10, 6, 16, 18, 14, 0, 4, 8]
>>>data.sort(key=lambda x: x)            #使用 lambda 表达式指定排序规则
>>>data
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
>>>data.sort(key=lambda x: -x)           #使用 lambda 表达式指定排序规则
>>>data
[18, 16, 14, 12, 10, 8, 6, 4, 2, 0]
#使用 lambda 表达式指定排序规则,将数字转换成字符串后,按字符串的长度来排序
>>>data.sort(key=lambda x: len(str(x)))
>>>data
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
>>>data.sort(key=lambda x: len(str(x)), reverse=True)
>>>data
[10, 12, 14, 16, 18, 0, 2, 4, 6, 8]
```

(4) lambda 表达式“:”后面,只能有一个表达式,返回一个值,而 def 则可以在 return 后面有多个表达式,返回多个值。

```
>>>def function(x):
    return x+1,x * 2,x * * 2
>>>print(function(3))
(4, 6, 9)
>>>(a, b, c)=function(3)       #通过元组接收返回值,并存放在不同的变量里
>>>print(a,b,c)
4 6 9
```

function 函数返回三个值,当它被调用时,需要三个变量同时接收函数返回的三个值。

3.7.2　自由变量对 lambda 表达式的影响

在 Python 中，函数是一个对象，与整数、字符串等对象有很多相似之处，例如，可以作为其他函数的参数，Python 中的函数还可以携带自由变量。通过下面的例子来分析 Python 函数在执行时是如何确定自由变量的值的。

```
>>>i=1
>>>def f(j):
    return i+j
>>>print(f(2))
3
>>>i=5
>>>print(f(2))
7
```

可见，当定义函数 f()时，Python 不会记录函数 f()里面的自由变量 i 对应什么对象，只会告诉函数 f()你有一个自由变量，它的名字叫 i。接着，当函数 f()被调用执行时，Python 告诉函数 f()：①空间上，你需要在你被定义时的外层命名空间(也称为作用域)里面去查找 i 对应的对象，这里将这个外层命名空间记为 S；②时间上，在函数 f()运行时，S 里面的 i 对应的对象。上面例子中的 i=5 之后，f(2)随之返回 7，恰好反映了这一点。再看下面类似的例子。

```
>>>fTest=map(lambda i:(lambda j: i * * j), range(1,6))
>>>print([f(2) for f in fTest])
[1, 4, 9, 16, 25]
```

在上面例子中，fTest 是一个行为列表，里面的每个元素是一个 lambda 表达式，每个表达式中的 i 值通过 map()函数映射确定下来，执行 print([f(2) for f in fTest])语句时，f()依次选取 fTest 中的 lambda 表达式并将 2 传递给选取的 lambda 表达式中的 j，所以输出结果为[1,4,9,16,25]。再如下面的例子。

```
>>>fs=[lambda j:i * j for i in range(6)]
#fs 中的每个元素相当于是含有参数 j 和自由变量 i 的函数
>>>print([f(2) for f in fs])
[10, 10, 10, 10, 10, 10]
```

之所以会出现[10,10,10,10,10,10]这样的输出结果，是因为列表 fs 中的每个函数在定义时其包含的自由变量 i 都是循环变量，因此，列表中的每个函数被调用执行时，其自由变量 i 都是对应循环结束 i 所指对象值 5。

3.8　变量的作用域

变量起作用的代码范围称为变量的作用域。在 Python 中，使用一个变量时并不需要预先声明它，但在真正使用它之前，它必须被绑定到某个内存对象(也即变量被定义、赋

值),变量名绑定将在当前作用域中引入新的变量,同时屏蔽外层作用域中的同名变量。

3.8.1 变量的局部作用域

在函数内部定义的变量称为局部变量,局部变量起作用的范围是函数内部,称为局部作用域。也就是说局部变量的作用域从创建变量的地方开始,直到包含该变量的函数结束为止。当函数运行结束后,在该函数内部定义的局部变量被自动删除而不可再访问。

(1) 函数内部的变量名 x 如果是第一次出现,且在赋值符号“=”左边,那么就可以认为在函数内部定义了一个局部变量 x。在这种情况下,不论全局变量名中是否有变量名 x,函数中使用的 x 都是局部变量。例如:

```
1. num=1
2. def func():
3.     num=2
4.     print(num)
5. func()
```

输出结果是 2,说明函数 func()中定义的局部变量 num 覆盖全局变量 num。

(2) 函数内部的变量名如果是第一次出现,且出现在赋值符号“=”后面,且在之前已被定义为全局变量,则这里将引用全局变量。例如:

```
num=10
def func():
  x=num+10
  print(x)
func()
```

运行上述程序代码,输出结果是 20。

(3) 函数中使用某个变量时,如果该变量名既有全局变量也有局部变量,则默认使用局部变量。例如:

```
num=10                  #全局变量
def func():
  num=20                #局部变量
  x=num * 10            #此处的 num 为局部变量
  print(x)
func()
```

运行上述程序代码,输出结果是 200。

(4) 有些情况需要在函数内部使用全局变量,这时可以使用 global 关键字来声明变量的作用域为全局。如果需要在函数内部改变全局变量的值,需要在函数内部使用 global 关键字来声明。

```
num=100
def func():
  global num            #声明 num 是全局变量
```

```
    num=200              #修改 num 全局变量的值
    print('在函数内输出 num: ', num)
func()
print('在函数外输出 num: ',num)
```

上述程序代码在 IDLE 中运行的结果如下：

```
在函数内输出 num: 200
在函数外输出 num: 200
```

num 在函数内外都输出 200。这说明函数中的变量名 num 被定义为全局变量，并被赋值为 200。

3.8.2　变量的全局作用域

不属于任何函数的变量一般为全局变量，它们在所有的函数之外创建，可以被所有的函数访问，也即模块层次中定义的变量，每一个模块都是一个全局作用域。也就是说，在模块文件顶层声明的变量具有全局作用域，模块的全局变量就像是一个模块对象的属性。

注意：全局作用域的范围仅限于单个模块文件内。

```
name='Jack'              #全局变量，具有全局作用域
def f1():
    age=18               #局部变量
    print(age, name)
def f2():
    age=19               #局部变量
    print(age, name)
f1()
f2()
```

上述程序代码在 IDLE 中运行的结果如下：

```
18 Jack
19 Jack
```

特殊说明：列表、字典可修改，但不能重新赋值，如果需要重新赋值，需要在函数内部使用 global 声明全局变量。

```
name=['Chinese','Math']              #全局变量
name1=['Java','Python']              #全局变量
name2=['C','C++']                    #全局变量
def f1():
    name.append('English')           #列表的 append()方法可改变外部全局变量的值
    print('函数内 name: %s'%name)
    name1=['Physics','Chemistry']    #重新赋值无法改变外部全局变量的值
    print('函数内 name1: %s'%name1)
    global name2  #如果需重新给全局变量 name2 赋值，需使用 global 声明全局变量
```

```
    name2='123'
    print('函数内 name2: %s'%name2)
f1()
print('函数外输出 name: %s'%name)
print('函数外输出 name1: %s'%name1)
print('函数外输出 name2: %s'%name2)
```

上述程序代码在 IDLE 中运行的结果如下：

```
函数内 name:['Chinese', 'Math', 'English']
函数内 name1:['Physics', 'Chemistry']
函数内 name2: 123
函数外输出 name:['Chinese', 'Math', 'English']
函数外输出 name1:['Java', 'Python']
函数外输出 name2: 123
```

3.8.3 变量的嵌套作用域

嵌套作用域也包含在函数中，嵌套作用域和局部作用域是相对的，嵌套作用域相对于更上层的函数而言也是局部作用域。与局部作用域的区别在于，对一个函数而言，局部作用域是定义在此函数内部的局部作用域，而嵌套作用域是定义在此函数的上一层父级函数的局部作用域。

嵌套作用域应用的示例代码如下：

```
x=5
def test1():
    x=10
    def test2():
        print(x)
    test2()
    print(x)

test1()
```

上述程序代码在 IDLE 中运行的结果如下：

```
10
10
```

从上面的例子可以看见，test1()和 test2()函数里面的 print 语句都是打印 test1()函数里面定义的 x，而没有涉及函数外的 x。其查找 x 的过程：调用 test1()函数，依次从上到下执行其里面的语句，执行到 test2()，进入 test2()内部执行，执行到 print(x)语句时，要解析 x，先在 test2()内部搜索，结果没有搜到，然后在 test2()的父级函数 test1()内搜索，搜索到 x=10，于是搜索在此处停止，变量的含义就定位到该处，即 x 的值是 10，执行 print(x)输出 10。print(x)执行后，其后面不再有 test2()函数的语句，于是控制权重新回

到 test1()，然后执行 test2()下面的语句 print(x)，在 test1()里搜到 x=10，于是搜索在此处停止，变量的含义就定位到该处，即 x 的值是 10，执行 print(x)输出 10。

由变量的局部作用域、变量的全局作用域和变量的嵌套作用域的介绍可知搜索变量名的优先级：局部作用域→嵌套作用域→全局作用域。也就是说，变量名解析机制是：在局部找不到，便会去局部外的局部找，再找不到就会去全局找。

3.9　函数的递归调用

在调用一个函数的过程中又出现直接或间接地调用该函数本身，称为函数的递归调用。递归函数就是一个调用自己的函数。递归常用来解决结构相似的问题。所谓结构相似，是指构成原问题的子问题与原问题在结构上相似，可以用类似的方法求解。具体地，整个问题的求解可以分为两部分：第一部分是一些特殊情况(也称为最简单的情况)，有直接的解法；第二部分与原问题相似，但比原问题的规模小，并且依赖第一部分的结果。每次递归调用都会简化原始问题，让它不断地接近最简单的情况，直至它变成最简单的情况。实际上，递归是把一个大问题转化成一个或几个小问题，再把这些小问题进一步分解成更小的小问题，直至每个小问题都可以直接解决。因此，递归有两个基本要素。

(1) 边界条件：确定递归到何时终止，也称为递归出口。

(2) 递归模式：大问题是如何分解为小问题的，也称为递归体。

递归函数只有具备了这两个要素，才能在有限次计算后得出结果。

许多数学函数都是使用递归来定义的，如数字 n 的阶乘 $n!$可以按下面的递归方式进行定义：

$$n! = \begin{cases} n! = 1 & (n = 0) \\ n \times (n-1)! & (n > 0) \end{cases}$$

对于给定的 n 如何求 $n!$呢？

求 $n!$可以用递推方法，即从 1 开始，乘以 2，再乘以 3……一直到乘以 n。这种方法容易理解，也容易实现。递推法的特点是从一个已知的事实(如 1!=1)出发，按一定规律推出下一个事实(如 2!=2×1!)，再从这个新的已知的事实出发，再向下推出一个新的事实(3!=3×2!)，直到推出 $n!=n\times(n-1)!$。

求 $n!$ 也可以用递归方法，即假设已知$(n-1)!$，使用 $n!=n\times(n-1)!$就可以立即得到 $n!$。这样，计算 $n!$的问题就简化为计算$(n-1)!$。当计算$(n-1)!$时，可以递归地应用这个思路直到 n 递减为 0。

假定计算 $n!$的函数是 factorial(n)。如果 $n=1$ 调用这个函数，立即就能返回它的结果，这种不需要继续递归就能知道结果的情况称为基础情况或终止条件。如果 $n>1$ 调用这个函数，它会把这个问题简化为计算 $n-1$ 的阶乘的子问题。这个子问题与原问题本质上是一样的，具有相同的计算特点，但比原问题更容易计算，计算规模更小。

计算 $n!$ 的函数 factorial(n)可简单地描述如下：

```
def factorial(n):
    if n==0:
```