第3章

函　数

学习目标

- 熟练掌握自定义函数的设计和使用。
- 深入理解各类参数，熟悉参数传递过程。
- 掌握 lambda 表达式。

本章先介绍函数的定义与调用，接着介绍形式参数、实际参数、函数的返回、位置参数与关键参数、默认参数、个数可变的参数等相关概念和用法，再介绍参数与返回值类型的注解、lambda 表达式和函数式编程中常用的类与内置函数。

3.1　函数的定义

函数是为实现一个特定功能而组合在一起并赋予一个名字（函数名）的语句集，可以被别的程序或函数本身通过函数名来引用，也可以用来定义可重用代码，组织和简化代码。

函数定义的语法格式如下：

```
def 函数名(形式参数):
    函数体
```

函数通过 def 关键字定义，包括函数名称、形式参数、函数体。函数名是标识符，命名必须符合 Python 标识符的规定；形式参数，简称为形参，写在一对圆括号里面。形参是可选的，即函数可以包含参数，也可以不包含参数，多个形参之间用逗号隔开。即使没有形参，这对圆括号也不能省略。该行以冒号结束。函数体是语句序列，左端必须缩进一些空格。

一些函数可能只完成要求的操作而无返回值，而另一些函数可能需要返回一个计算结果给调用者。如果函数有返回值，则使用关键字 return 来返回一个值。执行 return 语句的同时意味着函数的终止。

【例 3.1】 定义一个函数，其功能是求正整数的阶乘，并利用该函数求解 6!、16! 和 26! 的结果。

程序源代码如下：

```
#example3_1.py
#coding = utf - 8
def jc(n):                          #函数定义
    s = 1
    for i in range(1,n + 1):
        s * = i
    return s

#主程序
i = 6
k = jc(i)
print(str(i) + "!= ",k)
i = 16
k = jc(i)
print(str(i) + "!= ",k)
i = 26
k = jc(i)
print(str(i) + "!= ",k)
```

程序 example3_1.py 的运行结果：

```
6!= 720
16!= 20922789888000
26!= 403291461126605635584000000
```

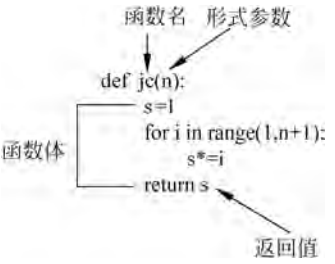

图 3.1 jc()函数的定义图解

这里定义了一个名为 jc 的函数，它有一个形式参数 n，函数返回 s 的值，即 n 的阶乘值。图 3.1 解释了这个函数的定义。

函数必须先定义再调用。否则，在调用时会得到函数名没有定义的错误提示。该程序从上往下执行，遇到 def 定义的内容先跳过，从非 def 定义的地方开始执行。这里非 def 定义的部分通常被称为主程序。

3.2 函数的调用

函数的定义是通过参数和函数体决定函数能做什么，并没有被执行。而函数一旦被定义，就可以在程序的任何地方被调用。当调用一个函数时，程序控制权就会转移到被调

用的函数上，真正执行该函数；执行完函数后，被调用的函数就会将程序控制权交还给调用者。

下面通过例 3.1 详细描述函数的调用过程。

在例 3.1 中，从主程序开始执行。执行主程序中的第一条语句，将 6 赋值给变量 i，然后执行主程序中的第二条语句，调用函数 jc(i)。当 jc(i)函数被调用时，变量 i 的值被传递到形参 n，程序控制权转移到 jc(i)函数，然后就开始执行 jc(i)函数。当 jc(i)函数的 return 语句被执行后，jc(i)函数将计算结果返回给调用者，并将程序的控制权转移给调用者主程序。回到主程序后，jc(i)函数的返回值赋值给变量 k。接下来执行主程序中的第三条语句，打印出结果。然后继续执行主程序的第四条语句，将 16 赋值给变量 i ……(后续调用与前面一致，不再重复)。图 3.2 解释了 jc(i)函数的调用过程。

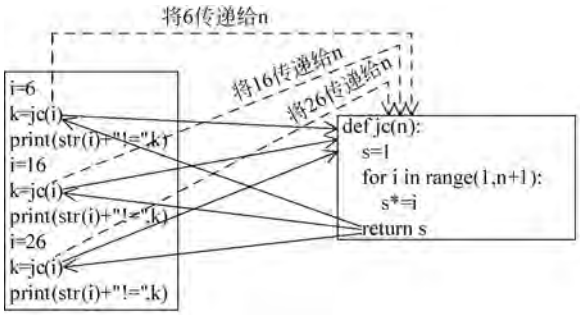

图 3.2 jc(i)函数的调用过程

3.3 形参与实参

在函数定义中，函数名后面圆括号中列出的参数称为形式参数，简称形参，如例 3.1 中 jc(n)函数的 n。如果形参的个数超过 1 个，各参数之间用逗号隔开。在定义函数时，函数的形参不代表任何具体的值，只有在函数调用时，才会有具体的值赋给形参。调用函数时传入的参数称为实际参数，简称实参，如例 3.1 中调用 jc(n)函数时使用 jc(i)传入的变量 i。

3.4 函数的返回

函数的执行结果通过返回语句 return 返回给调用者。函数体中不一定有表示返回的 return 语句。函数调用时的参数传递实现了从函数外部向函数内部输入数据，而函数的返回则解决了函数向外部输出信息的问题。如果一个函数的定义中没有 return 语句，系统将自动在函数体的末尾插入 return None 语句。

Python 语言提供了一条 return 语句用于从函数返回值，格式如下：

```
def 函数名(形式参数):
    …
    return <表达式 1>, … ,<表达式 n>
```

当一个函数需要返回多个值时，在 return 语句之后跟上多个需要返回的表达式或变量，这些表达式的值和变量将共同构成一个元组返回给调用者，所以返回的始终是一个对象。

3.5 位置参数与关键参数

当调用函数时，需要将实参传递给形参。参数传递时有两种方式：以位置参数形式赋值和以关键参数形式赋值。以位置参数形式赋值是指按照函数定义中形参的排列顺序来传递，以关键参数形式赋值是指按照参数赋值的形式来传递。

当使用位置参数时，实参和形参在顺序、个数和类型上必须一一匹配。例 3.1 中，调用带参数的函数时使用位置参数。

在函数调用中，也可以通过“变量名＝值”的“键-值”形式将实参传递给形参，使得参数可以不按顺序来传递，让函数参数的传递更加清晰、易用。采用这种方式传递的参数称为关键参数（也称关键字参数）。

【例 3.2】 编写一个函数，有三个形参，其中两个传递字符分别作为开始字符和结束字符，打印出两个字符之间的所有字符，每行打印的字符个数由第三个形参指定。

程序源代码如下：

```
#example3_2.py
#coding = utf - 8
def printChars(ch1, ch2, number):
    count = 0
    for i in range(ord(ch1), ord(ch2) + 1):
        count += 1
        if count % number!= 0:
            print(" % 4s" % chr(i),end = ' ')
        else:
            print(" % 4s" % chr(i))

#主程序
printChars("!","9",10)                          #以位置参数形式传递
print()
printChars(number = 10,ch2 = "9",ch1 = "!")     #以关键参数形式传递
print()
printChars("!",number = 10,ch2 = "9")           #位置参数和关键参数混合使用
```

程序 example3_2.py 的运行结果如下：

```
!   "   #   $   %   &   '   (   )   *
+   ,   -   .   /   0   1   2   3   4
5   6   7   8   9
!   "   #   $   %   &   '   (   )   *
+   ,   -   .   /   0   1   2   3   4
5   6   7   8   9
!   "   #   $   %   &   '   (   )   *
+   ,   -   .   /   0   1   2   3   4
5   6   7   8   9
```

在 printChars()函数中,ch1、ch2 表示两个字符,number 表示每行打印字符的个数。在主程序中,使用 printChars("!","9",10)表示输出字符! 到字符 9 之间的字符,每行打印 10 个字符。在该语句中,按照参数位置顺序将字符"!"传递给 ch1,将字符"9"传递给 ch2,将 10 传递给 number。函数调用 printChars(number=10,ch2="9",ch1="!")中,采用关键参数形式,将实参 10 传递给形参 number,将实参字符"9"传递给形参 ch2,将实参字符串"!"传递给形参 ch1。函数调用 printChars("!",number=10,ch2="9")中,第一个参数采用位置参数形式进行传递,后两个参数采用关键参数形式进行传递。

3.6 默认参数

函数定义时,形参可以设置默认值,这种形参通常称为默认参数。如果在调用函数时不为这些参数提供值,这些参数就使用默认值;如果在调用时有实参,则将实参的值传递给形参,形参定义的默认值将被忽略。具有默认参数值的函数定义格式如下:

```
def 函数名(非默认参数, 形参名 = 默认值, …):
        函数体
```

函数定义时,形式参数中非默认参数与默认参数可以并存,但非默认参数之前不能有默认参数。

【例 3.3】 默认参数应用实例。分析函数调用及程序的运行结果。

程序源代码如下:

```
#example3_3.py
#coding = utf - 8
#函数定义
def sayHello(s = "Hello!",n = 2,m = 1):
    for i in range(n):
        print(s * m)

#主程序
#形参没有赋予新值,均取默认值
sayHello()
print()

#按照顺序依次赋值给形参
sayHello("Ha!",3,4)
print()

#按照顺序,形参中 s 赋予新值"Ha!"
#n 没有赋新值,取默认值
#通过关键参数形式,为形参 m 赋予新值 3
sayHello("Ha!",m = 3)
```

程序 example3_3.py 的运行结果如下:

```
>>>
```

```
RESTART: d:\test\example3_3.py
Hello!
Hello!

Ha!Ha!Ha!Ha!
Ha!Ha!Ha!Ha!
Ha!Ha!Ha!Ha!

Ha!Ha!Ha!
Ha!Ha!Ha!
>>>
```

在该函数的定义中有三个参数——s、n 和 m，s 的默认值是字符串“Hello!”，n 的默认值是 2，m 的默认值是 1。

在主程序中，第 1 个 sayHello()调用语句没有提供实参值，所以程序就将默认值“Hello!”赋给 s，将默认值 2 赋给 n，将默认值 1 赋给 m，运行结果就是打印出两行字符串“Hello!”。

调用 sayHello("Ha!",3,4)时，这三个参数均是按位置赋值的，字符串“Ha!”赋给 s，3 赋给 n，4 赋给 m，运行结果就是打印出三行字符串“Ha! Ha! Ha! Ha!”，行数由 n 决定，字符串“Ha!”的重复次数由 m 决定。

调用 sayHello("Ha!",m=3)时，以位置参数形式将字符串“Ha!”赋给形参 s；没有提供实参值赋给 n，则将默认值赋给 n；以关键参数形式将整数 3 赋值给形参 m。打印出两行字符串“Ha! Ha! Ha!”。采用关键参数形式来传递实参可以跳过一些默认参数的赋值。这里采用关键参数形式为形参 m 重新赋予新值，直接跳过了默认参数 n 的赋值，此时 n 取默认值。

3.7 个数可变的参数

当需要接收不定个数参数时，形参以元组或字典等组合对象形式收集不定个数的实参。实参也可以以序列、字典等组合对象形式，为形参中的多个参数分配值。实参和形参也可以均为组合对象，从而可以实现不定个数参数的传递。


扫码观看

3.7.1 以组合对象为形参接收多个实参

在前面的函数介绍中，我们知道一个形参只能接收一个实参的值。其实在 Python 中，函数可以接收不定个数的参数，即用户可以给函数提供可变个数的实参。这可以通过在形参前面添加标识符（一个星号 * 或两个星号 ** ）来实现。

1. 将多个以位置参数形式传递的实参收集为形参中的元组

在函数定义的形参前面加一个星号 * ，则该参数将接收不定个数的、以位置参数传递的实参，构成一个元组。

【例 3.4】 编写一个函数，接收任意个数的参数并打印出来。

程序源代码如下：

```
#example3_4.py
```

```
#coding = utf - 8
#函数定义
def all_1( * args):
    print(args)

#主程序
all_1()
all_1("a")
all_1("a",2)
all_1("a",2,"b")
#all_1(x = "a",y = 2)        #这里不能以关键参数的形参传递
```

程序 example 3_4.py 的运行结果：

```
()
('a',)
('a', 2)
('a', 2, 'b')
```

在函数 all_1()的定义中，形参 args 前面有一个星号标识符 *，表明形参 args 可以接收不定个数的、以位置参数形式传递的实参。主程序中调用 all_1()，没有传递实参，形参 args 得到一个空的元组；主程序中调用 all_1("a")，传递一个参数给 args，结果以元组的形式输出('a',)；主程序中调用 all_1("a",2)，传递两个参数给 args，结果也是以元组的形式输出('a',2)；主程序中调用 all_1("a",2,"b")，传递三个参数给 args，结果还是以元组的形式输出('a',2,'b')。从这个示例中可以看出，不管传递几个参数到 args，都是将接收的所有参数按照次序组合到一个元组上。

example3_4.py 主程序的最后一行被注释掉了，否则将出现以下错误信息：

```
Traceback (most recent call last):
  File "D:\example3_4.py", line 12, in < module >
    all_1(x = "a",y = 2)
TypeError: all_1() got an unexpected keyword argument 'x'
```

因为加一个星号 * 的形参不接收以关键参数形式传递的实参。不定个数的以关键参数形式传递的实参可以被以两个星号 ** 为前缀的形参接收，并收集为字典形式。

以一个星号 * 为前缀的形参可以和其他普通形参联合使用，这时一般将以星号 * 为标识符的形参放在形参列表的后面，普通形参放在前面。

2. 将多个以关键参数形式传递的实参收集为形参中的字典

前面已经提到，以一个星号 * 为前缀的形式参数不接收以关键参数形式传递的实参值。在 Python 的函数形参中还提供了一种在参数名前面加两个星号 ** 的方式。这时，函数调用者须以关键参数的形式为其赋值，以两个星号 ** 为前缀的形参得到一个以关键参数中变量名为 key，右边表达式值为 value 的字典。

【例 3.5】 以“ ** ”为前缀的可变长度参数使用案例。

程序源代码如下：

```
#example3_5.py
#coding = utf - 8
#函数定义
def all_4( ** args):
    print(args)

#主程序
all_4(x = "a",y = "b",z = 2)
all_4(m = 3,n = 4)
all_4()
```

程序 example3_5.py 的运行结果如下：

```
{'x': 'a', 'y': 'b', 'z': 2}
{'m': 3, 'n': 4}
{}
```

在函数 all_4()的定义中，参数 args 前面有两个星号 **，表明该形参 args 可以将不定个数的、以关键参数形式给出的实参收集起来转换为一个字典。在主程序中第一次调用该函数时，以关键参数形式将三个参数传递给 args，输出的结果是一个字典；第二次调用该函数时，以关键参数形式将两个参数传递给 args，输出的结果还是一个字典；第三次调用时，没有传递实参，形参得到一个空字典。

以两个星号 ** 为前缀的不定个数参数、以一个星号 * 为前缀的不定个数参数和普通参数在函数定义中可以混合使用。这时，普通参数放在最前面，其次是以一个星号 * 为前缀的不定个数参数，最后是以两个星号 ** 为前缀的不定个数参数。


扫码观看

3.7.2 以组合对象为实参给多个形参分配参数

函数定义时的形参为单变量时，实参可以是以一个星号 * 为前缀的序列变量，将此序列中的元素分配给相应的形参；实参也可以是以两个星号 ** 为前缀的字典变量，根据字典中的 key 和形参变量名的对应关系，将字典中的 value 传递给相应的形参。

【例 3.6】 本例为以单变量为形参、以序列和字典为实参传递参数的案例，试分析程序输出结果并解释原因。

程序源代码如下：

```
#example3_6.py
#coding = utf - 8
#函数定义,形参为单变量参数
def snn3(x,y,z):
    return x + y + z

#主程序
aa = [1,2,3]                          #列表
print(snn3( * aa))                    #实参为列表变量(以" * "为前缀)
bb = (6,2,3)                          #元组
print(snn3( * bb))                    #实参为元组变量(以" * "为前缀)
ss = "abc"
```

```
print(snn3( * ss))                                    #实参为字符串变量(以" * "为前缀)
cc = [8,9]
print(snn3(7, * cc))                                  #实参为单变量 + 序列(以 * 为前缀)
print(snn3( * cc,20))
d1 = {"x":1,"y":2,"z":3}
print(snn3( ** d1))                                   #实参为字典变量(以 ** 为前缀)
d2 = {"y":2,"z":3}
print(snn3(1, ** d2))
d3 = {"x":1,"y":2}
#以 ** 为前缀的实参之后的普通参数以关键参数的形式传递
print(snn3( ** d3,z = 3))
```

程序 example3_6.py 的运行结果如下：

```
6
11
abc
24
37
6
6
6
```

在如上示例中，函数 snn3()中的形参是三个单变量，返回值为这三个变量的和，而在主程序中调用时，aa 是一个列表，也就是说用序列作实参时，要在序列前加 * ，而且序列的元素个数与 snn3()中的形参个数相同，aa 中的元素正好也是 3 个，这样调用时就写成 snn3(* aa)，输出结果 6 就是 aa 列表中三个元素的和。如果主程序中写成 snn3(aa)，则程序会出现这样的错误：snn3() takes exactly 3 arguments (1 given)，因为这时调用时将列表 aa 作为一个整体传递给形参 x，而形参 y 和 z 没有值，因此出错。而用 * aa，则能把实参的元素分解给各个形参，把列表 aa 中的元素 1 分解给 x、2 分解给 y、3 分解给 z，snn3()接收了三个参数。

bb 是一个元组，ss 是一个字符串。这两个变量与 aa 一样，也是序列作实参，调用时要在序列前加 * 。

cc 也是一个列表，但是只有两个元素，主程序中通过 snn3(7, * cc)或 snn3(* cc,20)均可调用。按照位置分别进行参数分配。

d1、d2 和 d3 均为字典。作为实参为单变量形参传递并分配值时，实参变量名前面加两个星号 ** 。如果这类实参变量名后面还有普通参数需要传递，则须采用关键参数形式。

3.7.3 形参和实参均为组合类型


扫码观看

当形参和实参均为序列时，可以通过在形参和实参前均添加一个星号 * 来实现参数传递。当形参和实参均为字典时，可以通过在形参和实参前均添加两个星号 ** 来实现参数传递。

【例 3.7】 形参和实参均为序列或形参和实参均为字典，分别添加 * 和 ** 作为形参

和实参的前缀来实现参数传递。

程序源代码如下：

```
#example3_7.py
#coding = utf - 8
#序列作为参数的函数定义
def snn1( * args):
    print(args)
    for i in args:
        print(i,end = ' ')
    print()

#字典作为参数的函数定义
def snn2( ** args):
    print(args)
    s = 0
    for i in args.keys():
        s += args[i]
    return s

#主程序
print("snn1:")
aa = [1,2,3]                                    #列表(序列)
snn1( * aa)
snn1( * [4,5])                                  #列表(序列)
bb = (6,2,3,1)                                  #元组(序列)
snn1( * bb)
snn1( * 'abc')                                  #字符串(序列)
print()
print("snn2:")
cc = {'x': 1, 'y': 2, 'c': 3}                   #字典
print(snn2( ** cc))
print(snn2( ** {'aa': 1, 'bb': 2 ,'cc': 4, 'dd': 5, 'ee': 6}))     #字典
```

程序 example3_7.py 的运行结果如下：

```
snn1:
(1, 2, 3)
1 2 3
(4, 5)
4 5
(6, 2, 3, 1)
6 2 3 1
('a', 'b', 'c')
a b c

snn2:
{'x': 1, 'y': 2, 'c': 3}
6
{'aa': 1, 'bb': 2, 'cc': 4, 'dd': 5, 'ee': 6}
18
```

实际上，当实参与形参类型相同时，参数可以直接传递，实参和形参均不需要添加任何星号为前缀。例 3.7 的程序可以改为以下实现形式：

```
#example3_7_another.py
#coding = utf - 8
#序列作为参数的函数定义
def snn1(args):
    print(args)
    for i in args:
        print(i,end = ' ')
    print()

#字典作为参数的函数定义
def snn2(args):
    print(args)
    s = 0
    for i in args.keys():
        s += args[i]
    return s

#主程序
print("snn1:")
aa = [1,2,3]                                    #列表(序列)
snn1(aa)
snn1([4,5])                                     #列表(序列)
bb = (6,2,3,1)                                  #元组(序列)
snn1(bb)
snn1('abc')                                     #字符串(序列)
print()
print("snn2:")
cc = {'x': 1, 'y': 2, 'c': 3}                   #字典
print(snn2(cc))
print(snn2({'aa': 1, 'bb': 2 ,'cc': 4, 'dd': 5, 'ee': 6}))      #字典
```

程序 example3_7_another.py 运行结果：

```
snn1:
[1, 2, 3]
1 2 3
[4, 5]
4 5
(6, 2, 3, 1)
6 2 3 1
abc
a b c

snn2:
{'x': 1, 'y': 2, 'c': 3}
6
{'aa': 1, 'bb': 2, 'cc': 4, 'dd': 5, 'ee': 6}
18
```

3.8 参数与返回值类型注解

Python是一种动态语言，在声明一个参数时不需要指定类型，并且函数的返回值也没有类型的定义。在编写程序时，参数可以接收任意值，运行时可能就会因为类型不匹配而出现错误。函数没有指明返回值类型，有时只有运行后才能确定类型，这可能导致调用程序运行时出错。为了使函数调用者明确函数的参数类型和返回值类型，引入了类型注解的概念，可以为函数参数和函数返回值标注类型。这种类型的注解不是强制执行的，在定义函数时也可以没有。本章前面自定义的函数中没有使用类型注解。类型注解的引入便于一些集成开发环境（Integrated Development Environment，IDE）可以在程序运行前进行校验，发现一些隐藏的类型不匹配错误。即使使用了类型注解，目前大部分编辑器仍然不做类型检查。

在参数名后面添加一个英文冒号，并在冒号后面写出类型名称，即可实现参数类型注解。如果该参数有默认值，则默认值的赋值号放在参数类型名称后面。如果要为函数注解返回值类型，则在参数列表的右侧圆括号后面添加“->类型”。

以下代码定义函数f(x)，用类型注解指明参数x的类型为int，用函数返回值类型注解指明函数返回值类型为int。另外指明参数x的默认值为5。

```
def f(x : int = 5) -> int :
    return x + 1
```

在一些IDE中，要求实参必须是形参指定的类型，并且返回值必须与指定的类型一致。但目前大部分IDE并不做检查，因此实参不是整数也可以调用f(x)，如f(1.1)。

3.9 lambda表达式

lambda表达式又称lambda函数，是一个匿名函数，比def格式的函数定义简单很多。lambda表达式可以接收任意多个参数，但只返回一个表达式的值。lambda中不能包含多个表达式。

lambda表达式的定义格式为：

```
lambda 形式参数 : 表达式
```

其中形式参数可以有多个，它们之间用逗号隔开。表达式只有一个。返回表达式的计算结果。

以下例子中赋值号左边的变量相当于给lambda函数定义了一个函数名。可以将此变量名作为函数名来调用该lambda表达式。

```
>>> f = lambda x,y : x + y
>>> f(5,10)
15
>>>
```

3.10 函数式编程的常用类与函数

map、reduce()和 filter 是函数式编程中常用的类与函数。这里先将利用初始化参数创建类的对象看作函数的调用,但两者有本质的区别。请读者在学完第 4 章后再来区分两者的区别。

1. filter 类

创建对象的格式:filter(function or None,iterable)

功能:把一个带有一个参数的函数 function 作用到一个可迭代(iterable)对象上,返回一个 filter 对象,filter 对象中的元素由可迭代(iterable)对象中使得函数 function 返回值为 True 的那些元素组成;如果指定函数为 None,则返回可迭代(iterable)对象中等价于 True 的元素。filter 对象是一个迭代器(iterator)对象。

```
>>> aa = [5,6, - 9, - 56, - 309,206]
>>> def func(x):                          #定义函数,x 为奇数时返回 True,为偶数时返回 False
        return x % 2!= 0

>>> bb = filter(func,aa)
>>> type(bb)                              #bb 是一个 filter 对象
<class 'filter'>
>>> list(bb)
[5, - 9, - 309]
>>> dd = [6,True,1,0,False]
>>> ee = filter(None,dd)                  #指定函数为 None
>>> list(ee)
[6, True, 1]
```

2. reduce()函数

函数调用格式:reduce(function,iterable[,initializer])

其中,参数 function 函数是有两个参数的函数,iterable 是需要迭代计算的可迭代对象,initializer 是计算的初始化参数,是可选参数。

reduce()函数对一个可迭代对象中的所有数据进行如下操作:用作为 reduce 参数的两参数函数 function 先对初始值 initializer 和可迭代对象 iterable 中的第 1 个元素进行 function 函数定义的相关计算;然后用得到的结果再与可迭代对象 iterable 中的第 2 个元素进行相关计算;直到遍历计算完可迭代对象 iterable 中的所有元素,最后得到一个结果。如果参数中没有初始值 initializer,则直接从可迭代对象 iterable 中的第 1 个和第 2 个元素开始计算,然后依次将结果和下一个对象进行计算,直到遍历计算完可迭代对象 iterable 中的所有元素,最后得到一个结果。

从 python 3 开始,reduce()函数移到了 functools 模块,使用之前需要先导入。例如:

```
>>> def add(x,y):
```

```
    return x + y

>>> from functools import reduce
>>> reduce(add,(1,2,3,4))          #使用两参数函数 add()
10
>>> reduce(lambda x,y:x + y,(1,2,3,4))    #使用两参数的 lambda 表达式
10
>>> reduce(lambda x,y:x + y,(1,2,3,4),5)  #初始值为 5
15
>>>
```

3. map 类

创建对象的格式：`map(func, * iterables)`

功能：把一个函数 func 依次作用到可迭代(iterable)对象的每个元素上，返回一个 map 对象。参数 * iterables 前的 * 表示 iterables 接收不定个数的可迭代对象，其个数由函数 func 中的参数个数决定。map 对象是一个迭代器(iterator)对象。例如：

```
>>> aa = ['1','5.6','7.8','9']
>>> bb1 = map(float,aa)
>>> bb1
< map object at 0x0000000002F76400 >
>>> list(bb1)
[1.0, 5.6, 7.8, 9.0]
>>> list(map(str,range(5)))
['0', '1', '2', '3', '4']
```

习题 3

1. 编写一个可判断一个数是否为素数的函数，然后调用该函数来判断从键盘输入的数是否为素数。素数也称质数，是指只能被 1 和它本身整除的自然数。

2. 编写一个可求出一个数除了 1 和自身以外的因子的函数。从键盘输入一个数，调用该函数输出除了 1 和它自身以外的所有因子。

3. 编写一个可判断一个数是否为水仙花数的函数。然后调用该函数打印出 1000 以内的所有水仙花数。水仙花数是指一个 n 位自然数(n≥3)，它的每个位上的数字的 n 次幂之和等于它本身。例如：$1^3 + 5^3 + 3^3 = 153$，则 153 是水仙花数。