Python语言及数据分析

王学军

wangxuejun@stdu.edu.cn

22-03

第3章 函数

第3章 函数

- 3.1 为什么要用函数
- 3.2 怎样定义函数
- 3.3 函数调用
- 3.4 函数参数传递
- 3.5 函数参数的类型
- 3.6 函数模块化
- 3.7 lambda表达式
- 3.8 变量的作用域
- 3.9 函数的递归调用
- 3.10 常用内置函数

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。
- 通过前面几章的学习，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我们的编程带来很多便利，提高了我们开发程序的效率。
- 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，这被叫做用户自定义函数。

第3章 函数

- ✓ 3.1 为什么要用函数
- ☐ 3.2 怎样定义函数
- ☐ 3.3 函数调用
- ☐ 3.4 函数参数传递
- ☐ 3.5 函数参数的类型
- ☐ 3.6 函数模块化
- ☐ 3.7 lambda表达式
- ☐ 3.8 变量的作用域
- ☐ 3.9 函数的递归调用
- ☐ 3.10 常用内置函数

3.1 为什么要用函数

- 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。
- 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、不精炼。
- 对于上面的两个问题，我们可以使用函数来解决，使代码得以重用。
- 函数是用来完成一定的功能。函数可看作是实现特定功能的小方法或是小程序。函数可简单地理解成：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。使用的时候只要调用这个名字，就可以实现这些语句的功能了。

第3章 函数

- 3.1 为什么要用函数
- ✓ 3.2 怎样定义函数
- 3.3 函数调用
- 3.4 函数参数传递
- 3.5 函数参数的类型
- 3.6 函数模块化
- 3.7 lambda表达式
- 3.8 变量的作用域
- 3.9 函数的递归调用
- 3.10 常用内置函数

3.2 怎样定义函数

在python中定义函数的语法如下所示：

```
def 函数名([参数列表]):
    函数体
```

（1）函数代码块以def 关键词开头，之后是函数名，def和函数名中间要敲一个空格。

（2）圆括号内用于定义函数参数，称为形式参数或简称为形参，参数是可选的，函数可以没有参数。如果有多个参数，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。

（3）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。

3.2 怎样定义函数

在python中定义函数的语法如下所示：

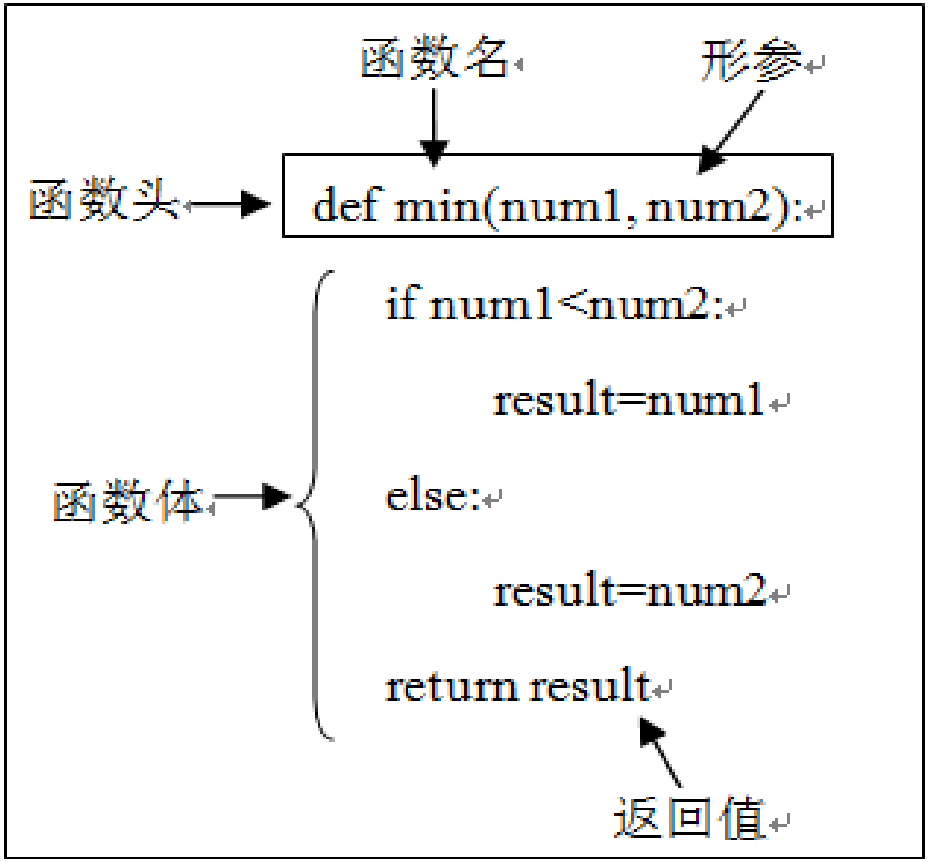
```
def 函数名([参数列表]):
    函数体
```

（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。

（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。

3.2 怎样定义函数

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。



第3章 函数

- 3.1 为什么要用函数
- 3.2 怎样定义函数
- ✓ 3.3 函数调用
- 3.4 函数参数传递
- 3.5 函数参数的类型
- 3.6 函数模块化
- 3.7 lambda表达式
- 3.8 变量的作用域
- 3.9 函数的递归调用
- 3.10 常用内置函数

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，调用函数有两种方式。

3.3 函数调用

根据函数是否有返回值，调用函数有两种方式。

（1）带有返回值的函数调用

对这种函数的调用通常当作一个值处理，例如：

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

（2）没有返回值的函数调用

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。

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

3.3 函数调用

TestSum.py

```
def sum(num1, num2):
    result = 0
    for i in range(num1, num2 + 1):
        result += i
    return result
def main():
    print("Sum from 1 to 10 is", sum(1, 10))
    print("Sum from 11 to 20 is", sum(11, 20))
    print("Sum from 21 to 30 is", sum(21, 30))
main() # Call the main function
```

3.3 函数调用

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

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，将函数以及函数体（第1到5行）存储在内存中。然后，解释器将main函数的定义（第6到9行）读取到内存。最后，解释器读取到第10行时，调用main函数，即main函数被执行。

第3章 函数

- 3.1 为什么要用函数
- 3.2 怎样定义函数
- 3.3 函数调用
- ✓ 3.4 函数参数传递
- 3.5 函数参数的类型
- 3.6 函数模块化
- 3.7 lambda表达式
- 3.8 变量的作用域
- 3.9 函数的递归调用
- 3.10 常用内置函数

3.4 函数参数传递

- Python 中一切都是对象，调用函数时的参数传递就是传可变对象还是传不可变对象。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。

3.4 函数参数传递

```
1.  def self_add(a):
2.      a += a

3.  a_int = 1
4.  print(a_int)  # 1
5.  self_add(a_int)
6.  print(a_int)  # 1

7.  a_list = [1, 2]
8.  print(a_list)  # [1,2]
9.  self_add(a_list)
10. print(a_list)  # [1,2,1,2]
```

第3章 函数

- 3.1 为什么要用函数
- 3.2 怎样定义函数
- 3.3 函数调用
- 3.4 函数参数传递
- ✓ 3.5 函数参数的类型
- 3.6 函数模块化
- 3.7 lambda表达式
- 3.8 变量的作用域
- 3.9 函数的递归调用
- 3.10 常用内置函数

3.5 函数参数的类型

- 函数的作用就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。

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，要求实参和形参的个数必须一致，而且实参和形参必须一一对应，也就是说第一个实参对应第一个形参。例如：

```
>>> 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]))
>>> 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
```

3.5.2 关键字参数

关键字参数用于函数调用，通过“键-值”形式加以指定。关键字参数主要指调用函数时的参数传递方式。使用关键字参数调用函数时，关键字参数可以按参数名字传递值，实参顺序可以和形参顺序不一致，也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。

```
>>> 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

定义带有默认值参数的函数的语法格式：
functionName(…, 参数名=默认值)

在定义函数的时候，为形参设置了默认值。在调用设置了默认值参数的函数时，如果没有给设置了默认值的形式参数传递实参，那么这个形参就将使用默认值。

```
def add(x=3,y=5):
    return(x + y)
>>> print(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，任何一个默认值参数右边都不能再出现非默认值参数。

3.5.3 默认值参数

```
def sum(a = 10, b):
    return a + b
print(sum(3))


def sum(b, a = 10):
    return a + b
print(sum(3))
```

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“**”表示这个参数是个字典参数(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。**

3.5.4 可变长度参数

练习

```
def f1(a,b):
    print (a,b)
def f2(a,*b):
    print (a,b)
def f3(a,**b):
    print (a,b)
def f4(a,*b,**c):
    print (a,b,c)
def f5(a,b=2,c=3):
    print (a,b,c)
def f6(a,b=2,*c):
    print (a,b,c)
```

```
f1(1,2)
f1(b=2,a=1)
f2(1,2,3,4)
f3(1,x=2,y=3,z=4)
f4(1,x=2,y=3)
f5(1)
f5(1,4)
f6(1)
f6(1,3,4,5,4)
```

```
1 2
1 2
1 (2, 3, 4)
1 {'x': 2, 'y': 3, 'z': 4}
1 () {'x': 2, 'y': 3}
1 2 3
1 4 3
1 2 ()
1 3 (4, 5, 4)
```

3.5.4 可变长度参数

下面的代码演示了定义函数时几种不同形式的参数混合使用的方法：

```
>>> def varLength (arg1,*tupleArg,**dictArg):
        print("arg1=", arg1)
        print("tupleArg=", tupleArg)
        print("dictArg=", dictArg)
>>> varLength('hello world','Python','C',a=1,b=2)
arg1= hello world
tupleArg= ('Python', 'C')
dictArg= {'a': 1, 'b': 2}
```

3.5.4 可变长度参数

下面的代码演示了定义函数时几种不同形式的参数混合使用的方法：

```
>>> def varLength (arg1,*tupleArg,**dictArg):
        print("arg1=", arg1)
        print("tupleArg=", tupleArg)
        print("dictArg=", dictArg)
>>> 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的参数传递方式，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解释器将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、字典以及其它可迭代对象，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。下面举例说明：

```
>>> 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
```

3.5.5 序列解包参数

```
>>> 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=(1, 0, 1)
>>> print1(tuple1[0], tuple1[1], tuple1[2])#常规调用函数
>>> print1(*tuple1)  #*将tuple1解开成3个元素并分别赋给x, y, z
1 0 1
```

```
dict1={‘x’:1, ‘y’:0, ‘z’:1}
print1(dict1[‘x’],dict1[‘y’],dict1[‘z’])
print1(**dict1)
```

第3章 函数

- 3.1 为什么要用函数
- 3.2 怎样定义函数
- 3.3 函数调用
- 3.4 函数参数传递
- 3.5 函数参数的类型
- ✓ 3.6 函数模块化
- 3.7 lambda表达式
- 3.8 变量的作用域
- 3.9 函数的递归调用
- 3.10 常用内置函数

3.6 函数模块化

在Python中，一个.py文件就构成一个模块。可以将多个函数的定义放在一个模块中，然后，将模块导入到其它程序中，这些程序就可以使用模块中定义的函数。

```
>>> 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
'Tue Mar  8 20:37:55 2022'
```

第3章 函数

- 3.1 为什么要用函数
- 3.2 怎样定义函数
- 3.3 函数调用
- 3.4 函数参数传递
- 3.5 函数参数的类型
- 3.6 函数模块化
- ✓ 3.7 lambda表达式
- 3.8 变量的作用域
- 3.9 函数的递归调用
- 3.10 常用内置函数

3.7 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它函数，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：**
- **lambda [参数1 ,参数2,.....参数n]:表达式**

3.7 lambda表达式

- **lambda表达式的语法：**

  lambda [参数1 ,参数2,.....参数n]:表达式

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
```

3.7 lambda表达式

lambda与def的区别：

1）def创建的方法是有名称的，而lambda没有。

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变量（函数名）。

3）lambda只是一个表达式，而def则是一个语句。

4）lambda表达式“:” 后面，只能有一个表达式，def则可以有多个

5）像if或for或print等语句不能用于lambda中，def可以。

6）lambda一般用来定义简单的函数，而def可以定义复杂的函数。lambda函数不能共享给别的程序调用，def可以。

第3章 函数

- ☐ 3.1 为什么要用函数
- ☐ 3.2 怎样定义函数
- ☐ 3.3 函数调用
- ☐ 3.4 函数参数传递
- ☐ 3.5 函数参数的类型
- ☐ 3.6 函数模块化
- ☐ 3.7 lambda表达式
- ✓ 3.8 变量的作用域
- ☐ 3.9 函数的递归调用
- ☐ 3.10 常用内置函数

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被称为局部变量，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。

有些情况需要在函数内部定义全局变量，这时可以使用global关键字来声明变量的作用域为全局。全局变量默认可读，如果需要改变全局变量的值，需要在函数内部使用global关键字来声明。

3.8.1 变量的局部作用域

```
num = 100
def func():
  global num   #声明num是全局变量
  num = 200   #修改name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
```

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。

3.8.2 变量的全局作用域

```
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
    print('函数内
    name1 = ['Ph
    print('函数内
    global name2
    name2 = '123
    print('函数内
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)
```

量

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
```

4.8.3 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。
- 搜索变量名的优先级：局部作用域 > 嵌套作用域 > 全局作用域。即变量名解析机制是：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。

4.8.3 变量的嵌套作用域

```
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()
```

```
In F3's F, a =1
```

4.8.3 变量的嵌套作用域

```
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()
```

```
In F4's F,a=2
In F4, a =1
```

4.8.3 变量的嵌套作用域

```
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()
```

```
In F5's F,a = 3
```

第3章 函数

- ☐ 3.1 为什么要用函数
- ☐ 3.2 怎样定义函数
- ☐ 3.3 函数调用
- ☐ 3.4 函数参数传递
- ☐ 3.5 函数参数的类型
- ☐ 3.6 函数模块化
- ☐ 3.7 lambda表达式
- ☐ 3.8 变量的作用域
- ✓ 3.9 函数的递归调用
- ☐ 3.10 常用内置函数

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，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。
- 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。

3.9 函数的递归调用

- 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：

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

- 计算n!的函数factorial(n)可简单地描述如下：
- def factorial (n):
- if n==0:
- return 1
- return n * factorial (n - 1)

第3章 函数

- 3.1 为什么要用函数
- 3.2 怎样定义函数
- 3.3 函数调用
- 3.4 函数参数传递
- 3.5 函数参数的类型
- 3.6 函数模块化
- 3.7 lambda表达式
- 3.8 变量的作用域
- 3.9 函数的递归调用
- ✓ 3.10 常用内置函数

3.10.1 map()函数

map(func, seq1[, seq2,…])：第一个参数接受一个函数名，后面的参数接受一个或多个可迭代的序列，将func依次作用在序列seq1[, seq2,…]的每个元素，得到一个新的序列。

（1）当序列seq只有一个时，将函数func作用于这个seq的每个元素上，并得到一个新的seq。

```
>>> L=[1,2,3,4,5]
>>> list(map((lambda x: x+5), L))   #将L中的每个元素加5
[6, 7, 8, 9, 10]
>>> list(map(str, L))      #将L中的每个元素转换为字符串
['1', '2', '3', '4', '5']
```

3.10.1 map()函数

（2）当序列seq多于一个时，每个seq的同一位置的元素同时传入多元的func函数（有几个列表，func就应该是几元函数），把得到的每一个返回值存放在一个新的序列中。

```
>>> def add(a, b):      #定义一个二元函数
        return a+b
>>> a=[1, 2, 3]
>>> b=[4, 5, 6]
>>> list(map(add, a, b))  #将a，b两个列表同一位置的元素相加求和
[5, 7, 9]
```

3.10.2 reduce()函数

reduce()函数在库functools里，reduce()函数的语法格式：

reduce(function, sequence[, initializer])

参数

function -- 函数，有两个参数；sequence -- 序列对象；

initializer -- 可选，初始参数

（1）reduce(function, sequence)：

先将sequence的第一个元素作为function函数的第一个参数和sequence的第二个元素作为function函数第二个参数进行function函数运算，然后将得到的返回结果作为下一次function函数的第一个参数和序列sequence的第三个元素作为function的第二个参数进行function函数运算，依次进行下去直到sequence中的所有元素都得到处理。

3.10.2 reduce()函数

（1）reduce(function, sequence)：

```
>>> from functools import reduce
>>> def add(x,y):
        return x+y
>>> reduce(add, [1, 2, 3, 4, 5])    #计算列表和：1+2+3+4+5
15
```

3.10.2 reduce()函数

（2）带初始参数initializer的reduce()函数：reduce(function, sequence, initializer)，先将初始参数initializer的值作为function函数的第一个参数和sequence的第一个元素作为function的第二个参数进行function函数运算，然后将得到的返回结果的作为下一次function函数的第一个参数和序列sequence的第二个元素作为function的第二个参数进行function函数运算，得到的结果再与第3个数据用 function 进行函数运算，依次进行下去直到sequence中的所有元素都得到处理。

```
>>> from functools import reduce
>>> reduce(lambda x, y: x + y, [2, 3, 4, 5, 6], 1)
21
```

3.10.2 reduce()函数

4.10.3 filter()函数

filter()函数：filter(func, iterable)用于过滤序列，即用函数func过滤掉iterable序列中不符合条件的元素，返回由符合条件元素组成的新序列。第一个参数为函数，第二个参数为序列，序列的每个元素作为参数传递给函数进行判断，最后将返回 True的元素放到新序列中。

例3-7. 过滤出列表中的所有奇数。

```
>>> def is_odd(n):
        return n%2 == 1
>>> newlist = filter(is_odd, range(20))
>>> list(newlist)
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

练习

1.利用reduce和lambda表达式编写n的阶乘函数

```
from functools import reduce

reduce(lambda x,y:x*y, range(1,n+1))
```

2.使用给定的整数n，编写一个程序生成一个包含(i, i*i)的字典，该字典包含1到n之间的整数(两者都包含)。然后程序应该打印字典。

```
print('请输入一个数字: ')
n=int(input())
d=dict()
for i in range(1,n+1):
    d[i]=i*i

print(d)
```

假设向程序提供以下输入:8

则输出为:

{1:1，2:4，3:9，4:16，5:25，6:36，,7:49，8:64}

THE END