



Python语言

第八周 Python函数应用

陈世峰 岭南师范学院信息工程学院
chenshifeng@lingnan.edu.cn/13234075348



教学目的

- 1 掌握函数的定义和调用方法
- 2 理解函数中参数的作用
- 3 理解变量的作用范围
- 4 函数的嵌套与递归

6.1 函数定义

- 函数是一段具有特定功能的、可重用的语句组，用函数名来表示并通过函数名进行完成功能调用。
- 函数也可以看作是一段具有名字的子程序，可以在需要的地方调用执行，不需要在每个执行地方重复编写这些语句。每次使用函数可以提供不同的参数作为输入，以实现对不同数据的处理；函数执行后，还可以反馈相应的处理结果。

Python语言的函数分类：

- 系统内置函数
- Python标准库（模块中定义的）函数。
- 用户自定义函数

系统内置函数是用户可直接使用的函数。Python标准库中的函数，要导入相应的标准库，才能使用其中的函数。

用户自定义函数是用户自己定义的函数，只有定义了这个函数，用户才能调用。这是本章要讨论的问题。

先了解C中函数的定义

Python定义一个函数使用def保留字，语法形式如下：

```
def <函数名>(<参数列表>):
```

```
    """注释"""
```

```
    <函数体>
```

```
    [return <返回值列表>]
```

❖ 注意事项

- ✓ 函数形参不需要声明其类型，也不需要指定函数返回值类型
- ✓ 即使该函数不需要接收任何参数，也必须保留一对空的圆括号
- ✓ 括号后面的冒号必不可少
- ✓ 函数体相对于def关键字必须保持一定的空格缩进
- ✓ Python允许嵌套定义函数

关键字 函数命名 输入参数

```
def square(x):
```

函数主体 {

```
    s = x*x
```

```
    return s
```

返回变量

```
a = 3
```

```
#调用函数，并打印出来  
print square(a)
```

```
#调用函数，保存至变量b  
b = square(a)
```

```
9
```

6.2 形参与实参

- 声明函数时所声明的参数，即为形式参数，简称形参
- 调用函数时，提供函数所需要的参数的值，即为实际参数，简称实参
- 在定义函数时，对参数个数并没有限制，如果有多个形参，需要使用逗号进行分割。

编写函数，接受两个整数，并输出其中最大数。

```
def printMax(a, b):  
    if a>b:  
        print(a, 'is the max')  
    else:  
        print(b, 'is the max')
```


- 对于绝大多数情况下，在函数内部直接修改形参的值不会影响实参。例如：

```
def addOne(a):  
    print(a)  
    a += 1  
    print(a)
```

```
>>>a = 3  
>>addOne(a)  
3  
4
```

```
>>>a  
3
```

- 在有些情况下，可以通过特殊的方式在函数内部修改实参的值，例如下面的代码。

```
def modify(v):  
    v[0] = v[0]+1
```

```
>>> a = [2]  
>>> modify(a)  
>>> a  
[3]
```

```
def append(v, item):  
    v.append(item)
```

```
>>> a = [2]  
>>> append(a,3)  
>>> a  
[2, 3]
```

Python语言只有一种参数传递方式，**值拷贝**。这种传值方式是让形参直接拷贝实参的值。从理论上讲，**如果实参是一个变量，形参变量的变化不会影响实参变量。**

如果传递的对象是可变对象，在函数中又修改了可变对象，这些修改将反映到原始对象中。这可以理解为形参拷贝了实参在内存中的引用。

6.3 参数类型

- 在Python中，函数参数有很多种：可以为普通参数、默认值参数、关键参数、可变长度参数等等。
- Python在定义函数时不需要指定形参的类型，完全由调用者传递的实参类型以及Python解释器的理解和推断来决定，类似于重载和泛型。
- Python函数定义时也不需要指定函数的类型，这将由函数中的return语句来决定，如果没有return语句或者return没有得到执行，则认为返回空值None。

默认值参数(可选参数)

- 在声明函数时，如果希望函数的一些参数是可选的，可以在声明函数时为这些参数指定默认值
- 调用该函数时，如果没有传入对应的实参值，则函数使用声明时指定的默认参数值

```
def sum(x, y=0, z=1) :  
    s=x+y+z  
    return s  
  
ad = add(100)
```

上面，“y=0, z=1 ”给了默认值，调用时，实参可不给值，直接使用默认值。
如果只对y给默认值，而不给z默认值，将引发异常。

- 默认值参数如果使用不当，会导致很难发现的逻辑错误，例如：

```
def demo(newitem,old_list=[]):  
    old_list.append(newitem)  
    return old_list
```

```
>>>demo('5', [1,2,3,4])
```

```
[1, 2, 3, 4, '5']
```

```
>>>demo('aaa', ['a','b'])
```

```
['a', 'b', 'aaa']
```

```
>>>demo('a')
```

```
['a']
```

```
>>>demo('b')
```

```
['a', 'b']
```

原因在于默认值参数的赋值只会在函数定义时被解释一次。
当使用可变序列作为参数默认值时，一定要谨慎操作。

正确写法：

```
def demo(newitem,old_list=None):  
    if old_list is None:  
        old_list=[]  
    old_list.append(newitem)  
    return old_list
```

✓默认值参数只在函数定义时被解释一次

✓可以使用“函数名.__defaults__”查看所有默认参数的当前值

关键字参数

- 函数调用时，实参默认按位置顺序传递形参。按位置传递的参数称之为位置参数
- 函数调用时，也可以通过名称（关键字）指定传入的参数，按名称指定传入的参数称为命名参数，也称之为关键字参数。使用关键字参数具有三个优点：
 - 参数按名称意义明确；
 - 传递的参数与顺序无关；
 - 如果有多个可选参数，则可以选择指定某个参数值

```
def sum(x, y=0, z=1):  
    s = x + y + z  
    return s
```

```
>>> sum(1, 2, 3)  
6
```

#位置参数

```
>>> sum(x=1, y=2, z=3)  
6
```

#关键字参数

```
>>> sum(z=3, x=2)  
5
```

强制关键字参数（Keyword-only）

- 在带星号的参数后面申明参数会导致强制关键字参数（Keyword-only）。调用时必须显式使用命名参数传递值，因为按位置传递的参数默认收集为一个元组，传递给前面带星号的可变参数
- 如果不需要带星的可变参数，只想使用强制命名参数，可以简单地使用一个星号。

```
def sum(x, *, y=0, z=1) :  
    s = x + y + z  
    return s
```

```
>> sum(4)  
5
```

```
>>sum(4, 5)
```

Traceback (most recent call last):

File "<pyshell#48>", line 1, in <module>
 add(4,5)

TypeError: add() takes 1 positional argument but 2 were given

```
>> sum(4, z=3, y=2)  
11
```

可变参数（VarArgs）

- 在声明函数时，通过带星的参数，如*param1，允许向函数传递可变数量的实参。调用函数时，从那一点后所有的参数被收集为一个元组
- 在声明函数时，也可以通过带双星的参数，如**param2，允许向函数传递可变数量的实参。调用函数时，从那一点后所有的参数被收集为一个字典
- 带星或双星的参数必须位于形参列表的最后位置

❖ *parameter的用法

```
def sum(a, b, *c):  
    total = a + b  
    for n in c:  
        total+=n  
    return total
```

```
>>> sum(1,2)
```

```
3
```

```
>>> sum(1,2,3,4,5)
```

```
15
```

```
>>> sum(1,2,3,4,5,6,7)
```

```
28
```

❖ **parameter的用法

```
def sum(a, b, **c):  
    total = a + b  
    for key in c:  
        total+=c[key]  
    return total
```

```
>>> sum(1, 2)  
3
```

```
>>> sum(1, 2, 3, 4, 5)  
出错
```

```
>>> sum(1, 2, x=3, y=4, z=5)  
15
```


- 几种不同类型的参数可以混合使用，但是不建议这样做

```
def test(a, b, c=4, *x, **y):  
    print(a, b, c)  
    print(x)  
    print(y)
```

```
>>> test(1,2,3,4,5,6,7,8,9,xx='1',yy='2',zz=3)
```

```
1, 2, 3
```

```
(4, 5, 6, 7, 8, 9)
```

```
{'yy': '2', 'xx': '1', 'zz': 3}
```

```
>>> test(a=11,b=22,c=33,d=44,e=55)
```

```
11 22 33
```

```
()
```

```
{'d': 44, 'e': 55}
```

参数传递的序列解包

- 传递参数时，可以通过在实参序列前加星号将其解包，然后传递给多个单变量形参。

```
def demo(a, b, c):  
    print(a+b+c)
```

```
>>> seq = [1, 2, 3]  
>>> demo(*seq)  
6
```

```
>>> tup = (1, 2, 3)  
>>> demo(*tup)  
6
```

```
>>> dic = {1:'a', 2:'b', 3:'c'}  
>>> demo(*dic)  
6  
>>> Set = {1, 2, 3}  
>>> demo(*Set)  
6  
>>> demo(*dic.values())  
abc
```

▪注意：调用函数时如果对实参使用一个星号*进行序列解包，那么这些解包后的实参将会被当做普通位置参数对待，并且会在**关键参数和使用两个星号**进行序列解包**的参数之前进行处理。

```
def demo(a, b, c):
```

#定义函数

```
    print(a, b, c)
```

```
>>> demo(*(1, 2, 3))
```

#调用，序列解包

```
1 2 3
```

```
>>> demo(1, *(2, 3))
```

#位置参数和序列解包同时使用

```
1 2 3
```

```
>>> demo(1, *(2,), 3)
```

```
1 2 3
```

```
>>> demo(a=1, *(2, 3)) #序列解包相当于位置参数，优先处理
```

```
Traceback (most recent call last):
```

```
File "<pyshell#26>", line 1, in <module>
```

```
demo(a=1, *(2, 3))
```

```
TypeError: demo() got multiple values for argument 'a'
```

```
>>> demo(b=1, *(2, 3))
```

```
Traceback (most recent call last):
```

```
File "<pyshell#27>", line 1, in <module>
```

```
demo(b=1, *(2, 3))
```

```
TypeError: demo() got multiple values for argument 'b'
```

```
>>> demo(c=1, *(2, 3))
```

```
2 3 1
```

```
>>> demo(**{'a':1, 'b':2}, *(3,))
```

#序列解包不能在关键参数解包之后

SyntaxError: iterable argument unpacking follows keyword argument unpacking

```
>>> demo(*(3,), **{'a':1, 'b':2})
```

Traceback (most recent call last):

File "<pyshell#30>", line 1, in <module>

demo(*(3,), **{'a':1, 'b':2})

TypeError: demo() got multiple values for argument 'a'

```
>>> demo(*(3,), **{'c':1, 'b':2})
```

```
3 2 1
```

函数定义与调用时，不定长参数的传入

函数定义时

*可以将按位置传递进来的参数“打包”成**元组**(tuple)类型

可以将按关键字传递进来的参数“打包”成字典**(dict)类型

函数调用时

*可以“解压”待传递到函数中的 元组、列表、集合、字符串等类型，并按位置传递到函数入口参数中

**可以“解压”待传递到函数中的字典，并按关键字传递到函数入口参数中

参数类型检查

- 定义函数时，不用限定其参数和返回值的类型
 - 这种灵活性可以实现多态性，即允许函数适用于不同类型的对象，例如，`sum(a, b, c)`函数，即可以返回三个`int`对象的和，也可以返回三个`float`对象的和，也可以连接三个`str`对象
- 当使用不支持的类型参数调用函数时，则会产生错误。例如，`sum(3, 4, 'a')`时，Python在运行时将抛出错误
TypeError
 - 用户调用函数时必须理解并保证传入正确类型的参数值

6.4 return语句

- return语句用来从一个函数中返回一个值，同时结束函数。
- 如果函数没有return语句，或者有return语句但是没有执行到，或者只有return而没有返回值，Python将认为该函数以return None结束。

```
def maximum( x, y ):
    if x>y:
        return x
    else:
        return y
```

多条return语句

return语句可以放置在函数中任何位置，当执行到第一个return语句时，程序返回到调用程序

例 判断素数。先编制一个判断一个数是否为素数的函数，然后编写测试代码，判断并输出1~99中的素数

```
def isPrime(n):  
    if n<2: return False  
    i = 2  
    while i*i <= n  
        #一旦n能被2~ $\sqrt{n}$  中的任意整数整除, n就不是素数, 返回False  
        if n%i ==0: return False  
        i+=1  
    return True
```

```
for i in range(100):  
    if isPrime(m): print(i, end='')
```

程序运行结果如下: ↵

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 ↵

返回多个值

- 在函数体中使用**return**语句，可实现从函数返回一个值，并跳出函数。如果需要返回多个值，则可以返回一个元组

```
>>>def func(a, b):  
    return b,a  
>>>s = func('knock~', 2)  
>>>print(s, type(s))  
(2, 'knock~') <class 'tuple'>
```

6.5 函数的嵌套

- 嵌套函数的意思就是函数里边套函数，即在一个函数里边，再定义一个函数。像这样定义在其他函数内的函数叫做内部函数，内部函数所在的函数叫做外部函数。

```
def A(a):  
    print('This is A')  
    def B(b):  
        print('This is B')  
        print('a + b =',a + b)  
    B(3)  
    print('OVER!')
```

上述代码定义了一个函数A(a)，在该函数内部，又定义了一个函数B(b)。若调用函数A(a)，将参数设为5，即为A(5)，则运行结果如下：

```
This is A  
This is B  
a + b = 8  
OVER!
```

6.6 函数的递归调用

- “递归”过程是指函数直接或间接调用自身完成某任务的过程。递归分为两类：直接递归和间接递归。
- 直接递归就是在函数中直接调用函数自身；
- 间接递归就是间接的调用一个函数，如第一个函数调用另一个函数，而该函数又调用了第一个函数。

求 $fac(n)=n!$ 的值。

根据求 $n!$ 的定义可知 $n! = n*(n-1)!$ ，可写成如下形式：

$$fac(n) = \begin{cases} 1 & n = 1 \\ n * fac(n-1) & n > 1 \end{cases}$$

```
def fac(n):  
    if n<=1:  
        return 1  
    else:  
        return n*fac(n-1)  
print(fac(4))
```

让我们来跟踪这个程序的计算过程，令 $n=4$ 调用这个函数：

(1) $\text{fac}(4)=4*\text{fac}(3)$ $n=4$ 调用函数过程 $\text{fac}(3)$

(2) $\text{fac}(3)=3*\text{fac}(2)$ $n=3$ 调用函数过程 $\text{fac}(2)$

(3) $\text{fac}(2)=2*\text{fac}(1)$ $n=2$ 调用函数过程 $\text{fac}(1)$

(4) $\text{fac}(1)=1$ $n=1$ 求得 $\text{fac}(1)$ 的值

(5) $\text{fac}(2)=2*1=2$ 回归， $n=2$ ，求得 $\text{fac}(2)$ 的值

(6) $\text{fac}(3)=3*2=6$ 回归， $n=3$ ，求得 $\text{fac}(3)$ 的值

(7) $\text{fac}(4)=4*6=24$ 回归， $n=4$ ，求得 $\text{fac}(4)$ 的值

上面第(1)步到第(4)步，求出 $\text{fac}(1)=1$ 的步骤称为递推，从第(4)步到第(7)步求出 $\text{fac}(4)=4*6$ 的步骤称为回归。

从这个例子可以看出，递归求解有2个条件：

(1) 给出递归终止的条件和相应的状态。本例中递归终止的条件是 $n=1$ ，状态是 $\text{fac}(1)=1$ 。

(2) 给出递归的表述形式，并且要向着终止条件变化，在有限步骤内达到终止条件。在本例中，当 $n>1$ 时，给出递归的表述形式为 $\text{fac}(n)=\text{fac}(n-1)*n$ 。函数值 $\text{fac}(n)$ 用函数 $\text{fac}(n-1)$ 来表示。参数的值向减少的方向变化，在第 n 步出现终止条件 $n=1$ 。

6.7 变量的作用域

- 变量起作用的代码范围称为变量的作用域，不同作用域内变量名可以相同，互不影响。
- 一个变量在函数外部定义和在函数内部定义，其作用域是不同的。
- 全局变量指在函数之外定义的变量，一般没有缩进，在程序执行全过程有效。
- 局部变量指在函数内部使用的变量，仅在函数内部有效，当函数退出时变量将不存在。

- 如果局部变量和全局变量同名，则在定义局部变量的函数中，只有局部变量是有效的。
- 局部变量的引用比全局变量速度快，应优先考虑使用。

```
a = 100                                #全局变量
def setNumber():
    a = 10                             #局部变量
    print(a)                           #打印局部变量a
setNumber()
print(a)                               #打印全局变量a
```

运行结果如下：

10
100

a = 100	#全局变量
def setNumber():	
global a	#声明全局变量
a = 10	#修改全局变量的值
print(a)	#打印全局变量a
setNumber()	
print(a)	#打印全局变量a

运行结果如下：

10

10

- 除了局部变量和全局变量，Python还支持使用nonlocal关键字定义一种介于二者之间的变量。关键字nonlocal声明的变量会引用距离最近的非全局作用域的变量，**要求声明的变量已经存在**，关键字nonlocal不会创建新变量。
- nonlocal关键字修饰变量后标识该变量是上一级函数中的局部变量

```

def scope_test():
    def do_local():
        spam = "111"

    def do_nonlocal():
        nonlocal spam    #这时要求spam必须是已存在的变量
        spam = "222"

    def do_global():
        global spam       #如果全局作用域内没有spam，就自动新建一个
        spam = "333"

    spam = "000"
    do_local()
    print("局部变量赋值后：", spam)
    do_nonlocal()
    print("nonlocal变量赋值后：", spam)
    do_global()
    print("全局变量赋值后：", spam)

scope_test()
print("全局变量：", spam)

```

局部变量赋值后： 000
nonlocal变量赋值后： 222
全局变量赋值后： 222
全局变量： 333

LEGB原理简要介绍

当一个函数体内需要引用一个变量的时候，会按照如下顺序查找：

- 首先查找局部变量（**L**ocals）；
- 如果找不到叫做该名称的局部变量，则去函数体的外层去寻找局部变量（**E**nclosing function locals）。（适用于嵌套函数的情况）
- 如果函数体外部的局部变量中也找不到叫做该名称的局部变量，则从全局变量（**G**lobal）中寻找。再找不到，只好去找内置库（**B**uilt-in）
- 像C语言就没有这种机制，局部区找不到就直接跳到静态变量（static）区了

lambda函数

- Python的lambda表达式的函数体只能有唯一的一条语句，也就是返回值表达式语句。其语法如下：
- 返回函数名 = **lambda** 参数列表:函数返回值表达式语句

例，下面的lambda表达式可以计算x、y和z等3个参数的和：

```
sum = lambda x,y,z : x + y + z
```

可以使用sum(x,y,z)调用上面的lambda表达式。

lambda表达式相当于下面的函数。

```
def sum(x,y,z):  
    return x + y + z
```