Python 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。你已经知道Python提供了许多内建函数，比如print()。但你也可以自己创建函数，这被叫做用户自定义函数。

定义一个函数

你可以定义一个由自己想要功能的函数，以下是简单的规则：

- 函数代码块以 def 关键词开头，后接函数标识符名称和圆括号()。
- 任何传入参数和自变量必须放在圆括号中间。圆括号之间可以用于定义参数。
- 函数的第一行语句可以选择性地使用文档字符串—用于存放函数说明。
- 函数内容以冒号起始，并且缩进。
- return [表达式] 结束函数，选择性地返回一个值给调用方。不带表达式的return相当于返回 None。

语法

```
def functionname( parameters ):
"函数_文档字符串"
function_suite
return [expression]
```

默认情况下，参数值和参数名称是按函数声明中定义的顺序匹配起来的。

实例

以下为一个简单的Python函数，它将一个字符串作为传入参数，再打印到标准显示设备上。

实例(Python 2.0+)

```
def printme( str ):
"打印传入的字符串到标准显示设备上"
print str
return
```

函数调用

定义一个函数只给了函数一个名称，指定了函数里包含的参数，和代码块结构。

这个函数的基本结构完成以后，你可以通过另一个函数调用执行，也可以直接从Python提示符执行。

如下实例调用了printme（ ）函数：

实例(Python 2.0+)

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
# 定义函数
def printme( str ):
"打印任何传入的字符串"
print str
return
# 调用函数
printme("我要调用用户自定义函数!")
printme("再次调用同一函数")
```

以上实例输出结果：

```
我要调用用户自定义函数!
再次调用同一函数
```

参数传递

在 python 中，类型属于对象，变量是没有类型的：

```
a=[1,2,3]
a="Runoob"
```

以上代码中，**[1,2,3]** 是 List 类型，**"Runoob"** 是 String 类型，而变量 a 是没有类型，她仅仅是一个对象的引用（一个指针），可以是 List 类型对象，也可以指向 String 类型对象。

可更改(mutable)与不可更改(immutable)对象

在 python 中，strings, tuples, 和 numbers 是不可更改的对象，而 list,dict 等则是可以修改的对象。

- **不可变类型：**变量赋值 **a=5** 后再赋值 **a=10**，这里实际是新生成一个 int 值对象 10，再让 a 指向它，而 5 被丢弃，不是改变a的值，相当于新生成了a。
- **可变类型：**变量赋值 **la=[1,2,3,4]** 后再赋值 **la[2]=5** 则是将 list la 的第三个元素值更改，本身la没有动，只是其内部的一部分值被修改了。

python 函数的参数传递：

- **不可变类型：**类似 c++ 的值传递，如 整数、字符串、元组。如fun（a），传递的只是a的值，没有影响a对象本身。比如在 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：**类似 c++ 的引用传递，如 列表，字典。如 fun（la），则是将 la 真正的传过去，修改后fun外部的la也会受影响

python 中一切都是对象，严格意义我们不能说值传递还是引用传递，我们应该说传不可变对象和传可变对象。

python 传不可变对象实例

实例(Python 2.0+)

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
def ChangeInt( a ):
a = 10
b = 2
ChangeInt(b)
print b # 结果是 2
```

实例中有 int 对象 2，指向它的变量是 b，在传递给 ChangeInt 函数时，按传值的方式复制了变量 b，a 和 b 都指向了同一个 Int 对象，在 a=10 时，则新生成一个 int 值对象 10，并让 a 指向它。

传可变对象实例

实例(Python 2.0+)

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
# 可写函数说明
def changeme( mylist ):
"修改传入的列表"
mylist.append([1,2,3,4])
print "函数内取值: ", mylist
return
# 调用changeme函数
mylist = [10,20,30]
changeme( mylist )
print "函数外取值: ", mylist
```

实例中传入函数的和在末尾添加新内容的对象用的是同一个引用，故输出结果如下：

```
函数内取值:  [10, 20, 30, [1, 2, 3, 4]]
函数外取值:  [10, 20, 30, [1, 2, 3, 4]]
```

参数

以下是调用函数时可使用的正式参数类型：

- 必备参数
- 关键字参数
- 默认参数
- 不定长参数

必备参数

必备参数须以正确的顺序传入函数。调用时的数量必须和声明时的一样。

调用printme()函数，你必须传入一个参数，不然会出现语法错误：

实例(Python 2.0+)

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
#可写函数说明
def printme( str ):
"打印任何传入的字符串"
print str
return
#调用printme函数
printme()
```

以上实例输出结果：

```
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    printme()
TypeError: printme() takes exactly 1 argument (0 given)
```

关键字参数

关键字参数和函数调用关系紧密，函数调用使用关键字参数来确定传入的参数值。

使用关键字参数允许函数调用时参数的顺序与声明时不一致，因为 Python 解释器能够用参数名匹配参数值。

以下实例在函数 printme() 调用时使用参数名：

实例(Python 2.0+)

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
#可写函数说明
def printme( str ):
"打印任何传入的字符串"
print str
return
#调用printme函数
printme( str = "My string")
```

以上实例输出结果：

```
My string
```

下例能将关键字参数顺序不重要展示得更清楚：

实例(Python 2.0+)

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
#可写函数说明
def printinfo( name, age ):
"打印任何传入的字符串"
print "Name: ", name
print "Age ", age
return
#调用printinfo函数
printinfo( age=50, name="miki" )
```

以上实例输出结果：

```
Name:  miki
Age  50
```

默认参数

调用函数时，默认参数的值如果没有传入，则被认为是默认值。下例会打印默认的age，如果age没有被传入：

实例(Python 2.0+)

```
#!/usr/bin/python
```

```
# -*- coding: UTF-8 -*-
#可写函数说明
def printinfo( name, age = 35 ):
"打印任何传入的字符串"
print "Name: ", name
print "Age ", age
return
#调用printinfo函数
printinfo( age=50, name="miki" )
printinfo( name="miki" )
```

以上实例输出结果：

```
Name:  miki
Age  50
Name:  miki
Age  35
```

不定长参数

你可能需要一个函数能处理比当初声明时更多的参数。这些参数叫做不定长参数，和上述2种参数不同，声明时不会命名。基本语法如下：

```
def functionname([formal_args,] *var_args_tuple ):
"函数_文档字符串"
function_suite
return [expression]
```

加了星号（*）的变量名会存放所有未命名的变量参数。不定长参数实例如下：

实例(Python 2.0+)

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
# 可写函数说明
def printinfo( arg1, *vartuple ):
"打印任何传入的参数"
print "输出: "
print arg1
for var in vartuple:
print var
return
```

```
# 调用printinfo 函数
printinfo( 10 )
printinfo( 70, 60, 50 )
```

以上实例输出结果：

```
输出:
10
输出:
70
60
50
```

匿名函数

python 使用 lambda 来创建匿名函数。

- lambda只是一个表达式，函数体比def简单很多。
- lambda的主体是一个表达式，而不是一个代码块。仅仅能在lambda表达式中封装有限的逻辑进去。
- lambda函数拥有自己的命名空间，且不能访问自有参数列表之外或全局命名空间里的参数。
- 虽然lambda函数看起来只能写一行，却不等同于C或C++的内联函数，后者的目的是调用小函数时不占用栈内存从而增加运行效率。

语法

lambda函数的语法只包含一个语句，如下：

```
lambda [arg1 [,arg2,.....argn]]:expression
```

如下实例：

实例(Python 2.0+)

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
# 可写函数说明
sum = lambda arg1, arg2: arg1 + arg2
# 调用sum函数
print "相加后的值为 : ", sum( 10, 20 )
print "相加后的值为 : ", sum( 20, 20 )
```

以上实例输出结果：

```
相加后的值为 :  30
相加后的值为 :  40
```

---

return 语句

return语句[表达式]退出函数，选择性地向调用方返回一个表达式。不带参数值的return语句返回None。之前的例子都没有示范如何返回数值，下例便告诉你怎么做：

实例(Python 2.0+)

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
# 可写函数说明
def sum( arg1, arg2 ):
# 返回2个参数的和."
total = arg1 + arg2
print "函数内 : ", total
return total
# 调用sum函数
total = sum( 10, 20 )
```

以上实例输出结果：

```
函数内 :  30
```

变量作用域

一个程序的所有的变量并不是在哪个位置都可以访问的。访问权限决定于这个变量是在哪里赋值的。变量的作用域决定了在哪一部分程序你可以访问哪个特定的变量名称。两种最基本的变量作用域如下：

- 全局变量
- 局部变量

全局变量和局部变量

定义在函数内部的变量拥有一个局部作用域，定义在函数外的拥有全局作用域。

局部变量只能在其被声明的函数内部访问，而全局变量可以在整个程序范围内访问。调用函数时，所有在函数内声明的变量名称都将被加入到作用域中。如下实例：

实例(Python 2.0+)

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
total = 0 # 这是一个全局变量
# 可写函数说明
def sum( arg1, arg2 ):
#返回2个参数的和."
total = arg1 + arg2 # total在这里是局部变量.
print "函数内是局部变量 : ", total
return total
#调用sum函数
sum( 10, 20 )
print "函数外是全局变量 : ", total
```

以上实例输出结果：

```
1  函数内是局部变量 :  30
2  函数外是全局变量 :  0
```