← Python3 循环语句 Python3 数据结构 →

Python3 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

Python 定义函数使用 def 关键字，一般格式如下：

```
def 函数名（参数列表）：
    函数体
```

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

让我们使用函数来输出"Hello World！"：

```
>>>def hello() :
print("Hello World!")
>>> hello()
Hello World!
>>>
```

更复杂点的应用，函数中带上参数变量:

实例(Python 3.0+)

```
#!/usr/bin/python3
# 计算面积函数
def area(width, height):
return width * height
def print_welcome(name):
print("Welcome", name)
```

```
print_welcome("Runoob")
w = 4
h = 5
print("width =", w, " height =", h, " area =", area(w, h))
```

以上实例输出结果：

```
Welcome Runoob
width = 4  height = 5  area = 20
```

函数调用

定义一个函数：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 命令提示符执行。

如下实例调用了 **printme()** 函数：

实例(Python 3.0+)

```
#!/usr/bin/python3
# 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），可以是指向 List 类型对象，也可以是指向 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 3.0+)

```
#!/usr/bin/python3
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

可变对象在函数里修改了参数，那么在调用这个函数的函数里，原始的参数也被改变了。例如：

实例(Python 3.0+)

```
#!/usr/bin/python3
# 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)
```

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：

- 必需参数
- 关键字参数
- 默认参数
- 不定长参数

必需参数

必需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 3.0+)

```
#!/usr/bin/python3
#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 10, in <module>
    printme()
TypeError: printme() missing 1 required positional argument: 'str'
```

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 3.0+)

```
#!/usr/bin/python3
#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 = "菜鸟教程")
```

以上实例输出结果：

```
菜鸟教程
```

以下实例中演示了函数参数的使用不需要使用指定顺序：

实例(Python 3.0+)

```
#!/usr/bin/python3
#可写函数说明
def printinfo( name, age ):
"打印任何传入的字符串"
print ("名字: ", name)
print ("年龄: ", age)
return
#调用printinfo函数
printinfo( age=50, name="runoob" )
```

以上实例输出结果：

```
名字:  runoob
年龄:  50
```

默认参数

调用函数时，如果没有传递参数，则会使用默认参数。以下实例中如果没有传入 age 参数，则使用默认值：

实例(Python 3.0+)

```
#!/usr/bin/python3
#可写函数说明
def printinfo( name, age = 35 ):
"打印任何传入的字符串"
print ("名字: ", name)
print ("年龄: ", age)
return
#调用printinfo函数
printinfo( age=50, name="runoob" )
print ("------------------------")
printinfo( name="runoob" )
```

以上实例输出结果：

```
名字:  runoob
年龄:  50
------------------------
```

```
名字:  runoob
年龄:  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 `*` 的参数会以元组(tuple)的形式导入，存放所有未命名的变量参数。

实例(Python 3.0+)

```
#!/usr/bin/python3
# 可写函数说明
def printinfo( arg1, *vartuple ):
"打印任何传入的参数"
print ("输出: ")
print (arg1)
print (vartuple)
# 调用printinfo 函数
printinfo( 70, 60, 50 )
```

以上实例输出结果：

```
输出:
70
(60, 50)
```

如果在函数调用时没有指定参数，它就是一个空元组。我们也可以不向函数传递未命名的变量。如下实例：

实例(Python 3.0+)

```
#!/usr/bin/python3
# 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
```

还有一种就是参数带两个星号 `**` 基本语法如下：

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

加了两个星号 `**` 的参数会以字典的形式导入。

实例(Python 3.0+)

```
#!/usr/bin/python3
# 可写函数说明
def printinfo( arg1, **vardict ):
"打印任何传入的参数"
print ("输出: ")
print (arg1)
print (vardict)
# 调用printinfo 函数
printinfo(1, a=2,b=3)
```

以上实例输出结果：

```
输出:
1
{'a': 2, 'b': 3}
```

声明函数时，参数中星号 `*` 可以单独出现，例如:

```
def f(a,b,*,c):
    return a+b+c
```

如果单独出现星号 `*` 后的参数必须用关键字传入。

```
>>> def f(a,b,*,c):
...     return a+b+c
...
>>> f(1,2,3)   # 报错
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: f() takes 2 positional arguments but 3 were given
>>> f(1,2,c=3) # 正常
6
>>>
```

匿名函数

python 使用 lambda 来创建匿名函数。

所谓匿名，意即不再使用 def 语句这样标准的形式定义一个函数。

- 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，且不能访问自己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 3.0+)

```
#!/usr/bin/python3
# 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 [表达式] 语句用于退出函数，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，以下实例演示了 return 语句的用法：

实例(Python 3.0+)

```
#!/usr/bin/python3
# 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 )
print ("函数外 : ", total)
```

以上实例输出结果：

```
函数内 :  30
函数外 :  30
```

变量作用域

Python 中，程序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可以访问哪个特定的变量名称。Python的作用域一共有4种，分别是：

- L （ Local ）局部作用域
- E （ Enclosing ）闭包函数外的函数中
- G （ Global ）全局作用域
- B （ Built-in ）内置作用域（ 内置函数所在模块的范围 ）

以 L –> E –> G –>B 的规则查找，即：在局部找不到，便会去局部外的局部找（ 例如闭包 ），再找不到就会去全局找，再者去内置中找。

```
g_count = 0  # 全局作用域
def outer():
    o_count = 1  # 闭包函数外的函数中
    def inner():
        i_count = 2  # 局部作用域
```

内置作用域是通过一个名为 builtin 的标准模块来实现的，但是这个变量名自身并没有放入内置作用域内，所以必须导入这个文件才能够使用它。在Python3.0中，可以使用以下的代码来查看到底预定义了哪些变量:

```
>>> import builtins
>>> dir(builtins)
```

Python 中只有模块（ module ），类（ class ）以及函数（ def、lambda ）才会引入新的作用域，其它的代码块（ 如 if/elif/else/、try/except、for/while等 ）是不会引入新的作用域的，也就是说这些语句内定义的变量，外部也可以访问，如下代码：

```
>>> if True:
...  msg = 'I am from Runoob'
...
>>> msg
'I am from Runoob'
>>>
```

实例中 msg 变量定义在 if 语句块中，但外部还是可以访问的。

如果将 msg 定义在函数中，则它就是局部变量，外部不能访问：

```
>>> def test():
...     msg_inner = 'I am from Runoob'
...
>>> msg_inner
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'msg_inner' is not defined
>>>
```

从报错的信息上看，说明了 msg_inner 未定义，无法使用，因为它是局部变量，只有在函数内可以使用。

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 3.0+)

```
#!/usr/bin/python3
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：

```
函数内是局部变量 :  30
函数外是全局变量 :  0
```

global 和 nonlocal关键字

当内部作用域想修改外部作用域的变量时，就要用到global和nonlocal关键字了。

以下实例修改全局变量 num：

实例(Python 3.0+)

```
#!/usr/bin/python3
num = 1
def fun1():
global num # 需要使用 global 关键字声明
print(num)
num = 123
print(num)
fun1()
print(num)
```

以上实例输出结果：

```
1
123
123
```

如果要修改嵌套作用域（enclosing 作用域，外层非全局作用域）中的变量则需要 nonlocal 关键字了，如下实例：

实例(Python 3.0+)

```
#!/usr/bin/python3
def outer():
num = 10
def inner():
nonlocal num # nonlocal关键字声明
num = 100
print(num)
inner()
print(num)
outer()
```

以上实例输出结果：

```
100
100
```

另外有一种特殊情况，假设下面这段代码被运行：

实例(Python 3.0+)

```
#!/usr/bin/python3
a = 10
def test():
a = a + 1
print(a)
test()
```

以上程序执行，报错信息如下：

```
Traceback (most recent call last):
  File "test.py", line 7, in <module>
    test()
  File "test.py", line 5, in test
    a = a + 1
UnboundLocalError: local variable 'a' referenced before assignment
```

错误信息为局部作用域引用错误，因为 test 函数中的 a 使用的是局部，未定义，无法修改。

修改 a 为全局变量，通过函数参数传递，可以正常执行输出结果为：

实例(Python 3.0+)

```
#!/usr/bin/python3
a = 10
def test(a):
a = a + 1
print(a)
test(a)
```

执行输出结果为：

```
11
```

课后练习

下一题 完成 重新测验

Python3 循环语句 Python3 数据结构

14 篇笔记 写笔记