



马哥教育

IT 人的高薪职业学院

Python函数返回值、作用域

讲师：Wayne

从业十余载，漫漫求知路

函数的返回值

□ 举例

```
def showplus(x):  
    print(x)  
    return x + 1
```

```
showplus(5)
```

□ 举例

```
def showplus(x):  
    print(x)  
    return x + 1  
print(x+1) # 会执行吗？
```

```
showplus(5)
```



函数的返回值

□ 多条return语句

```
def guess(x):  
    if x > 3:  
        return "> 3"  
    else:  
        return "<= 3"
```

```
print(guess(10))
```

□ return可以执行多次吗？

□ 多条return语句

```
def showplus(x):  
    print(x)  
    return x + 1  
    return x + 2
```

```
showplus(5)
```



函数返回值

□ 举例

```
def fn(x):  
    for i in range(x):  
        if i > 3:  
            return i  
    else:  
        print("{} is not greater than 3".format(x))
```

print(fn(5)) # 打印什么？

print(fn(3)) # 打印什么？



函数的返回值

□ 总结

- Python函数使用return语句返回“返回值”
- 所有函数都有返回值，如果没有return语句，隐式调用return None
- return 语句并不一定是函数的语句块的最后一条语句
- 一个函数可以存在多个return语句，但是只有一条可以被执行。如果没有一条return语句被执行到，隐式调用return None
- 如果有必要，可以显示调用return None，可以简写为return
- 如果函数执行了return语句，函数就会返回，当前被执行的return语句之后的其它语句就不会被执行了
- 作用：结束函数调用、返回值

函数的返回值

□ 返回多个值

```
def showlist():
```

```
    return [1, 3, 5]
```

showlist函数是返回了多个值吗？

```
def showlist():
```

```
    return 1, 3, 5
```

这次showlist函数是否返回了多个值呢？



函数的返回值

□ 返回多个值

- 函数不能同时返回多个值

- `return [1, 3, 5]` 是指明返回一个列表，是一个列表对象

- `return 1, 3, 5` 看似返回多个值，隐式的被python封装成了一个元组

```
def showlist():
```

```
    return 1, 3, 5
```

```
x, y, z = showlist() # 使用解构提取更为方便
```

函数嵌套

□ 函数嵌套

- 在一个函数中定义了另外一个函数

```
def outer():  
    def inner():  
        print("inner")  
    print("outer")  
    inner()
```

outer()

inner()

- 函数有可见范围，这就是**作用域**的概念
- 内部函数不能在外部直接使用，会抛NameError异常，因为它不可见



作用域***

□ 作用域

□ 一个标识符的可见范围，这就是标识符的作用域。一般常说的是变量的作用域

□ 举例，对比左右2个函数

```
x = 5
```

```
def foo():  
    print(x)
```

```
foo()
```

x到底可见还是不可见？

```
x = 5
```

```
def foo():  
    x += 1  
    print(x)
```

```
foo()
```

作用域***

□ 全局作用域

- 在整个程序运行环境中都可见

□ 局部作用域

- 在函数、类等内部可见
- 局部变量使用范围不能超过其所在的局部作用域

```
def fn1():
```

```
    x = 1 # 局部作用域，在fn1内
```

```
def fn2():
```

```
    print(x) # x可见吗
```

```
print(x) # x可见吗
```

作用域***

□ 嵌套结构

```
def outer1(): #  
    o = 65  
    def inner():  
        print("inner {}".format(o))  
        print(chr(o))  
    print("outer {}".format(o))  
    inner()
```

outer1()

□ 左边和右边代码中变量o的差别

□ 嵌套结构

```
def outer2(): #  
    o = 65  
    def inner():  
        o = 97  
        print("inner {}".format(o))  
        print(chr(o))  
    print("outer {}".format(o))  
    inner()
```

outer2()



作用域***

□ 从嵌套结构例子看出

- 外层变量作用域在内层作用域可见

- 内层作用域inner中，如果定义了o=97，相当于当前作用域中重新定义了一个新的变量o，但是这个o并没有覆盖外层作用域outer中的o

- 再看下面代码

```
x = 5
```

```
def foo():
```

```
    y = x + 1 # 报错吗
```

```
    x += 1 # 报错，报什么错？为什么？换成x=1还有错吗？
```

```
    print(x) # 为什么它不报错
```

```
foo()
```



作用域***

□ 代码

x = 5

def foo():

 x += 1

```
def foo():  
    x += 1  
foo()
```

```
UnboundLocalError                                Traceback (most recent call last)  
<ipython-input-108-f79b3c374b59> in <module>()  
      1 def foo():  
      2     x += 1  
----> 3 foo()  
  
<ipython-input-108-f79b3c374b59> in foo()  
      1 def foo():  
----> 2     x += 1  
      3 foo()  
  
UnboundLocalError: local variable 'x' referenced before assignment
```

- x += 1 其实是 x = x + 1
- 相当于在foo内部定义一个局部变量x，那么foo内部所有x都是这个局部变量x了
- 但是这个x还没有完成赋值，就被右边拿来加1操作了
- 如何解决这个问题？

作用域***

□ 全局变量global

```
x = 5
```

```
def foo():
```

```
    global x
```

```
    x += 1
```

- 使用global关键字的变量，将foo内的x声明为使用外部的全局作用域中定义的x
- 全局作用域中必须有x的定义
- 如果全局作用域中没有x定义会怎样？

作用域***

□ 全局变量global

```
#x = 5
```

```
def foo():
```

```
    global x
```

```
    x = 10
```

```
    x += 1 # 报错吗？
```

```
    print(x) # 打印什么？
```

```
print(x) #打印什么？
```

做这些实验建议不要使用ipython、jupyter，因为它会上下文中有x定义，可能测试不出效果

□ 使用global关键字的变量，将foo内的x声明为使用外部的全局作用域中定义的x

□ 但是，x = 10 赋值即定义，在内部作用域为一个外部作用域的变量x赋值，**不是在内部作用域定义**

个新变量，所以x+=1不会报错。注意，**这里x的作用域还是全局的**



作用域***

□ global总结

- $x += 1$ 这种是特殊形式产生的错误的原因？先引用后赋值，而python动态语言是赋值才算定义，才能被引用。解决办法，在这条语句前增加 $x = 0$ 之类的赋值语句，或者使用global 告诉内部作用域，去全局作用域查找变量定义
- 内部作用域使用 $x = 5$ 之类的赋值语句会重新定义局部作用域使用的变量 x ，但是，一旦这个作用域中使用global声明 x 为全局的，那么 $x = 5$ 相当于在为全局作用域的变量 x 赋值

□ global使用原则

- 外部作用域变量会内部作用域可见，但也不要在这个内部的局部作用域中直接使用，因为函数的目的就是为了封装，尽量与外界隔离
- 如果函数需要使用外部全局变量，请使用函数的形参传参解决
- 一句话：**不用global**。学习它就是为了深入理解变量作用域

闭包*

- 自由变量：未在本地作用域中定义的变量。例如定义在内存函数外的外层函数的作用域中的变量
- 闭包：就是一个概念，出现在嵌套函数中，指的是内层函数引用到了外层函数的自由变量，就形成了闭包。很多语言都有这个概念，最熟悉就是JavaScript

- 先看右边一段代码

- 第4行会报错吗？为什么
 - 第8行打印什么结果？
 - 第10行打印什么结果？

```
1 def counter():
2     c = [0]
3     def inc():
4         c[0] += 1 #报错吗？为什么
5         return c[0]
6     return inc
7 foo = counter()
8 print(foo(), foo())
9 c = 100
10 print(foo())
```


闭包*

□ 代码解析

□ 第4行会报错吗？为什么

- 不会报错，c已经在counter函数中定义过了。而且inc中的使用方式是修改c的元素值，而不是重新定义变量

□ 第8行打印什么结果？

- 打印 1 2

□ 第10行打印什么结果？

- 打印 3

- 第9行的c和counter中的c不一样，而inc引用的是自由变量正式counter的变量c

- 这是Python2中实现闭包的方式，Python3还可以使用nonlocal关键字

```
1 def counter():
2     c = [0]
3     def inc():
4         c[0] += 1 #报错吗？为什么
5         return c[0]
6     return inc
7 foo = counter()
8 print(foo(), foo())
9 c = 100
10 print(foo())
```

闭包*

- ❑ 下面这段代码会报错吗？为什么？
- ❑ 使用global能否解决？

```
1 def counter():  
2     count = 0  
3     def inc():  
4         count += 1  
5         return count  
6     return inc  
7  
8 foo = counter()  
9 foo()  
10 foo()
```

- ❑ 使用global可以解决，但是这使用的是全局变量，而不是闭包
- ❑ 如果要对普通变量的闭包，Python3中可以使用nonlocal



nonlocal关键字

- 使用了nonlocal关键字，将变量标记为不在本地作用域定义，而在上级的某一级局部作用域中定义，但不能是全局作用域中定义

```
1 def counter():
2     count = 0
3     def inc():
4         nonlocal count
5         count += 1
6         return count
7     return inc
8
9 foo = counter()
10 foo()
11 foo()
```



```
1 a = 50
2 def counter():
3     nonlocal a
4     a += 1
5     print(a)
6     count = 0
7     def inc():
8         nonlocal count
9         count += 1
10        return count
11    return inc
12
13 foo = counter()
14 foo()
15 foo()
```

上一级作用域是全局作用域

- count 是外层函数的局部变量，被内部函数引用
- 内部函数使用nonlocal关键字声明count变量在上级作用域而非本地作用域中定义
- 左边代码可以正常使用，且形成闭包
- 右边代码不能正常运行，变量 a 不能在全局作用域中

默认值的作用域

□ 默认值举例

```
def foo(xyz=1):  
    print(xyz)  
foo() # 打印什么？  
foo() # 打印什么？  
print(xyz) # 打印什么？
```

□ 默认值举例

```
def foo(xyz=[]):  
    xyz.append(1)  
    print(xyz)  
foo() # 打印什么？  
foo() # 打印什么？  
print(xyz) # 打印什么？
```



默认值的作用域

□ 默认值举例

```
def foo(xyz=[]):  
    xyz.append(1)  
    print(xyz)  
foo() # [1]  
foo() # [1,1]  
print(xyz) # NameError , 当前作用域没有xyz变量
```

□ 为什么第二次调用foo函数打印的是[1,1]？

- 因为**函数也是对象**，python把函数的默认值放在了属性中，这个属性就伴随着这个函数对象的整个生命周期
- 查看foo.__defaults__属性



默认值的作用域

□ 运行这个例子

```
def foo(xyz=[], u='abc', z=123):  
    xyz.append(1)  
    return xyz  
print(foo(), id(foo))  
print(foo.__defaults__)  
print(foo(), id(foo))  
print(foo.__defaults__)
```

□ 函数地址并没有变，就是说函数这个对象的没有变，调用它，它的属性__defaults__中使用元组保存默认值

□ xyz默认值是引用类型，引用类型的元素变动，并不是元组的变化



默认值的作用域

□ 非引用类型例子

```
def foo(w, u='abc', z=123):  
    u = 'xyz'  
    z = 789  
    print(w, u, z)  
print(foo.__defaults__)  
foo('magedu')  
print(foo.__defaults__)
```

□ 属性__defaults__中使用元组保存所有位置参数默认值，它不会因为在函数体内使用了它而发生改变



默认值的作用域

□ 举例

```
def foo(w, u='abc', *, z=123, zz=[456]):
```

```
    u = 'xyz'
```

```
    z = 789
```

```
    zz.append(1)
```

```
    print(w, u, z, zz)
```

```
print(foo.__defaults__)
```

```
foo('magedu')
```

```
print(foo.__kwdefaults__)
```

□ 属性__defaults__中使用元组保存所有位置参数默认值

□ 属性__kwdefaults__中使用字典保存所有keyword-only参数的默认值



默认值的作用域

- 使用可变类型作为默认值，就可能修改这个默认值
- 有时候这个特性是好的，有的时候这种特性是不好的，有副作用
- 如何做到按需改变呢？看下面的2种方法



默认值的作用域

```
def foo(xyz=[], u='abc', z=123):
```

```
    xyz = xyz[:] # 影子拷贝
```

```
    xyz.append(1)
```

```
    print(xyz)
```

```
foo()
```

```
print(foo.__defaults__)
```

```
foo()
```

```
print(foo.__defaults__)
```

```
foo([10])
```

```
print(foo.__defaults__)
```

```
foo([10,5])
```

```
print(foo.__defaults__)
```

□ 1、函数体内，不改变默认值

□ xyz都是传入参数或者默认参数的副本，如果就想修改原参数，无能为力



默认值的作用域

```
def foo(xyz=None, u='abc', z=123):  
    if xyz is None:  
        xyz = []  
    xyz.append(1)  
    print(xyz)
```

```
foo()  
print(foo.__defaults__)  
foo()  
print(foo.__defaults__)  
foo([10])  
print(foo.__defaults__)  
foo([10,5])  
print(foo.__defaults__)
```

□ 2、使用不可变类型默认值

- 如果使用缺省值None就创建一个列表
- 如果传入一个列表，就修改这个列表



默认值的作用域

□ 第一种方法

- 使用影子拷贝创建一个新的对象，永远不能改变传入的参数

□ 第二种方法

- 通过值的判断就可以灵活的选择创建或者修改传入对象
- 这种方式灵活，应用广泛
- 很多函数的定义，都可以看到使用None这个不可变的值作为默认参数，可以说这是一种惯用法

变量名解析原则LEGB

- ❑ Local, 本地作用域、局部作用域的local命名空间。函数调用时创建, 调用结束消亡
 - ❑ Enclosing, Python2.2时引入了嵌套函数, 实现了闭包, 这个就是嵌套函数的外部函数的命名空间
 - ❑ Global, 全局作用域, 即一个模块的命名空间。模块被import时创建, 解释器退出时消亡
 - ❑ Build-in, 内置模块的命名空间, 生命周期从python解释器启动时创建到解释器退出时消亡。例如 print(open), print和open都是内置的变量
- ❑ 所以一个名词的查找顺序就是LEGB

内置 (Python)

在内置变量名模块中预定义的变量名: open、range、SyntaxError.....

全局 (模块)

在模块文件顶层赋值的变量名, 或者在该文件中的 def 生成的名为全局变量的变量名。

上层函数的本地作用域

任何以及所有上层函数 (def 或 lambda) 作用域中的变量名, 由内及外

本地 (函数)

在函数内 (def 或 lambda) 通过使用方式赋值, 且没有通过 global 声明为全局变量的变量名。

函数的销毁

□ 全局函数

```
def foo(xyz=[], u='abc', z=123):  
    xyz.append(1)  
    return xyz  
print(foo(), id(foo), foo.__defaults__)  
def foo(xyz=[], u='abc', z=123):  
    xyz.append(1)  
    return xyz  
print(foo(), id(foo), foo.__defaults__)  
del foo  
print(foo(), id(foo), foo.__defaults__)
```



函数的销毁

- 全局函数销毁
 - 重新定义同名函数
 - del 语句删除函数对象
 - 程序结束时



函数的销毁

□ 局部函数

```
def foo(xyz=[], u='abc', z=123):  
    xyz.append(1)  
    def inner(a=10):  
        pass  
    print(inner)  
    def inner(a=100):  
        print(xyz)  
    print(inner)  
    return inner  
bar = foo()  
print(id(foo),id(bar), foo.__defaults__, bar.__defaults__)  
del bar  
print(id(foo),id(bar), foo.__defaults__, bar.__defaults__)
```



函数的销毁

□ 局部函数销毁

- 重新在上级作用域定义同名函数
- del 语句删除函数名称，函数对象的引用计数减1
- 上级作用域销毁时



谢谢

咨询热线 400-080-6560

