

# 函数作用域\*\*\*

## 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

```
1 def foo():
2     x = 100
3
4 print(x) # 可以访问到吗
```

上例中x不可以访问到，会抛出异常（**NameError: name 'x' is not defined**），原因在于函数是一个封装，它会开辟一个**作用域**，x变量被限制在这个作用域中，所以在函数外部x变量**不可见**。

注意：每一个函数都会开辟一个作用域

## 作用域分类

- 全局作用域
  - 在整个程序运行环境中都可见
  - 全局作用域中的变量称为**全局变量**global
- 局部作用域
  - 在函数、类等内部可见
  - 局部作用域中的变量称为**局部变量**，其使用范围不能超过其所在局部作用域
  - 也称为**本地作用域**local

```
1 # 局部变量
2 def fn1():
3     x = 1 # 局部作用域，x为局部变量，使用范围在fn1内
4
5 def fn2():
6     print(x) # x能打印吗？可见吗？为什么？
7
8 print(x) # x能打印吗？可见吗？为什么？
```

```
1 # 全局变量
2 x = 5 # 全局变量，也在函数外定义
3 def foo():
4     print(x) # 可见吗？为什么？
5
6 foo()
```

- 一般来讲外部作用域变量**可以在函数内部可见**，可以使用
- 反过来，函数内部的局部变量，不能在函数外部看到

## 函数嵌套

在一个函数中定义了另外一个函数

```
1 def outer():
2     def inner():
3         print("inner")
4     inner()
5     print("outer")
6
7 outer() # 可以吗?
8 inner() # 可以吗?
```

内部函数inner不能在外部直接使用，会抛NameError异常，因为它在函数外部不可见。

其实，inner不过就是一个标识符，就是一个函数outer内部定义的变量而已。

## 嵌套结构的作用域

对比下面嵌套结构，代码执行的效果

```
1 def outer1():
2     o = 65
3     def inner():
4         print('inner', o, chr(o))
5     inner()
6     print('outer', o, chr(o))
7
8 outer1() # 执行后，打印什么
9
10
11 def outer2():
12     o = 65
13     def inner():
14         o = 97
15         print('inner', o, chr(o))
16     inner()
17     print('outer', o, chr(o))
18
19 outer2() # 执行后，打印什么
```

从执行的结果来看：

- 外层变量在内部作用域可见
- 内层作用域inner中，如果定义了 `o = 97`，相当于在当前函数inner作用域中重新定义了一个新的变量o，但是，这个o并不能覆盖掉外部作用域outer2中的变量o。只不过对于inner函数来说，其只能可见自己作用域中定义的变量o了

| 内建函数 | 函数签名   | 说明                |
|------|--------|-------------------|
| chr  | chr(i) | 通过unicode编码返回对应字符 |
| ord  | ord(c) | 获得字符对应的unicode    |

```

1 print(ord('中'), hex(ord('中')), '中'.encode(), '中'.encode('gbk'))
2
3 chr(20013) # '中'
4 chr(97)

```

## 一个赋值语句的问题

再看下面左右2个函数

|   |  |
|---|--|
| <pre> x = 5 def foo():     print(x)  foo() </pre> | <pre> x = 5 def foo():     y = x + 1 # 报错吗     #x += 1   # 打开这一句报错吗？为什么？换成x=1行吗     print(x)  foo() </pre> |
|---|--|

| 左边函数                | 右边函数  |
|---------------------|---|
| 正常执行，函数外部的变量在函数内部可见 | 执行错误吗，为什么？难道函数内部又不可见了？ $y = x + 1$ 可以正确执行，可是为什么 $x += 1$ 却不能正确执行？ |

仔细观察函数2返回的错误指向 $x += 1$ ，原因是什么呢？

```

1 x = 5
2 def foo():
3     x += 1
4 foo() # 报错如下

```

```

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

```

原因分析：

- $x += 1$  其实是  $x = x + 1$
- 只要有“ $x =$ ”出现，这就是赋值语句。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
- $x = x + 1$  相当于使用了局部变量x，但是这个x还没有完成赋值，就被右边拿来加1操作了

```

1 x = 5
2
3 def foo(): # 函数被解释器解释，foo指向函数对象，同时解释器会理解x是什么作用域
4     print(x) # x 在函数解析时就被解释器判定为局部变量
5     x += 1 # x = x + 1
6
7 foo() # 调用时

```

如何解决这个常见问题？

## global语句

```

1 x = 5
2 def foo():
3     global x # 全局变量
4     x += 1
5     print(x)
6 foo()

```

- 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？

**注意**，下面试验如果在ipython、jupyter中做，上下文运行环境中有可能有x的定义，稍微不注意，就测试不出效果

```

1 # 有错吗？
2 def foo():
3     global x
4     x += 1
5     print(x)
6 foo()

```

```

1 # 有错吗？
2 def foo():
3     global x
4     x = 10
5     x += 1
6     print(x)
7 foo()
8 print(x) # 可以吗

```

使用global关键字定义的变量，虽然在foo函数中声明，但是这将告诉当前foo函数作用域，这个x变量将使用外部全局作用域中的x。

即使是在foo中又写了 `x = 10`，也不会再foo这个局部作用域中定义局部变量x了。

**使用了global，foo中的x不再是局部变量了，它是全局变量。**

## 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 = 10` 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

- 外部作用域变量会在内部作用域可见，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，请尽量使用函数的形参定义，并在调用传实参解决
- 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。例如定义在内层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

```
1  def counter():
2      c = [0]
3      def inc():
4          c[0] += 1 # 报错吗？ 为什么 # line 4
5          return c[0]
6      return inc
7
8  foo = counter() # line 8
9  print(foo(), foo()) # line 9
10 c = 100
11 print(foo()) # line 11
```

上面代码有几个问题：

- 第4行会报错吗？为什么
- 第9行打印什么结果？
- 第11行打印什么结果？

### 代码分析

- 第8行会执行`counter`函数并**返回inc对应的函数对象**，注意这个函数对象并不释放，因为有`foo`记着
- 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`中的使用方式是为`c`的元素修改值，而不是重新定义`c`变量
- 第9行打印什么结果？
  - 打印 1 2
- 第11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是counter中的变量c

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

再看下面这段代码，会报错吗？使用global能解决吗？

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

上例一定出错，使用gobal可以解决

```
1 def counter():
2     global count
3     count = 0
4     def inc():
5         global count
6         count += 1
7         return count
8     return inc
9
10 foo = counter()
11 print(foo(), foo())
12 count = 100
13 print(foo()) # 打印几？
```

上例使用global解决，这是全局变量的实现，而不是闭包了。

如果要对这个普通变量使用闭包，Python3中可以使用nonlocal关键字。

## nonlocal语句

**nonlocal**：将变量标记为不在本地作用域定义，而是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。

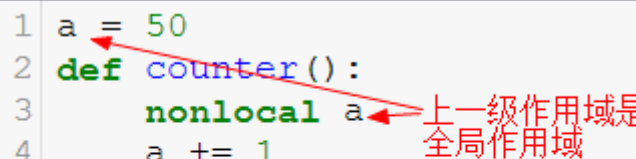
```
1 def counter():
2     count = 0
3     def inc():
4         nonlocal count # 声明变量count不是本地变量
5         count += 1
6         return count
7     return inc
8
9 foo = counter()
10 print(foo(), foo())
```

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。

代码中内层函数引用外部局部作用域中的自由变量，形成闭包。

```
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

上例是错误的，nonlocal 声明变量 a 不在当前作用域，但是往外就是全局作用域了，所以错误。

## 函数的销毁

定义一个函数就是生成一个函数对象，函数名指向的就是函数对象。

可以使用del语句删除函数，使其引用计数减1。

可以使用同名标识符覆盖原有定义，本质上也是使其引用计数减1。

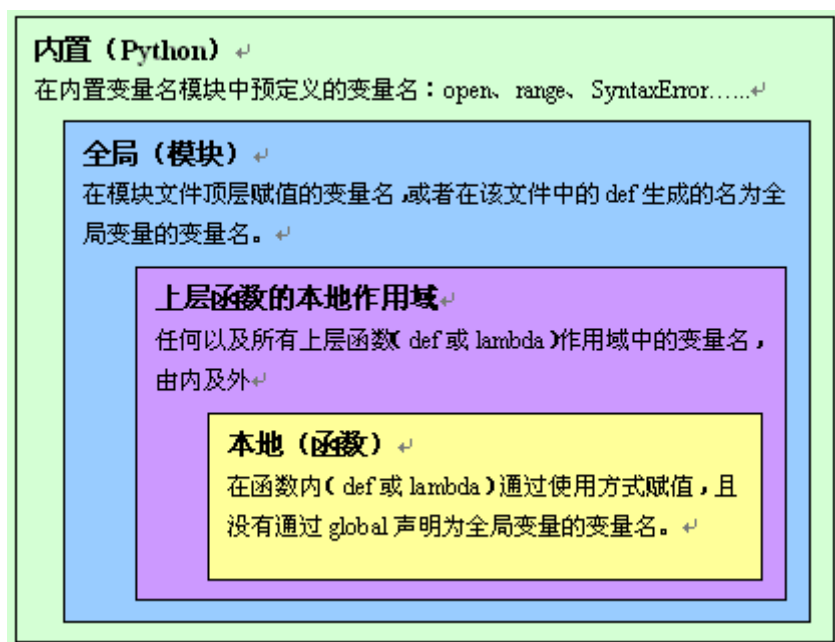
Python程序结束时，所有对象销毁。

函数也是对象，也不例外，是否销毁，还是看引用计数是否减为0。

## 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



| 内建函数      | 函数签名                      | 说明   |
|-----------|---------------------------|--|
| iter      | iter(iterable)            | 把一个可迭代对象包装成迭代器   |
| next      | next(iterable[, default]) | 取迭代器下一个元素<br>如果已经取完，继续取抛 <code>StopIteration</code> 异常 |
| reversed  | reversed(seq)             | 返回一个翻转元素的迭代器   |
| enumerate | enumerate(seq, start=0)   | 迭代一个可迭代对象，返回一个迭代器<br>每一个元素都是数字和元素构成的二元组                |

## 迭代器

- 特殊的对象，一定是可迭代对象，具备可迭代对象的特征
- 通过 `iter` 方法把一个可迭代对象封装成迭代器
- 通过 `next` 方法，获取 迭代器对象的一个元素
- 生成器对象，就是迭代器对象。但是迭代器对象未必是生成器对象

## 可迭代对象

- 能够通过迭代一次次返回不同的元素的对象
  - 所谓相同，不是指值是否相同，而是元素在容器中是否是同一个，例如列表中值可以重复的，`['a', 'a']`，虽然这个列表有2个元素，值一样，但是两个 `'a'` 是不同的元素
- 可以迭代，但是未必有序，未必可索引
- 可迭代对象有：`list`、`tuple`、`string`、`bytes`、`bytearray`、`range`、`set`、`dict`、生成器、迭代器等
- 可以使用成员操作符 `in`、`not in`
  - 对于线性数据结构，`in` 本质上是在遍历对象，时间复杂度为  $O(n)$

```
1 lst = [1, 3, 5, 7, 9]
2
3 it = iter(lst) # 返回一个迭代器对象
```



```
4 print(next(it))
5 print(next(it))
6
7 for i, x in enumerate(it, 2):
8     print(i, x)
9 #print(next(it)) # StopIteration
10 print()
11
12 for x in reversed(lst):
13     print(x)
14
15 # 比较下面的区别，说明原因？
16 it = iter(lst)
17 print(1 in it)
18 print(1 in it)
19 print(1 in lst)
20 print(1 in lst)
```

