

Python函数

函数

数学定义

- $y=f(x)$ ， y 是 x 的函数， x 是自变量。 $y=f(x_0, x_1, \dots, x_n)$

Python函数

- 由若干语句组成的语句块、函数名称、参数列表构成，它是组织代码的最小单元
- 完成一定的功能

函数的作用

- 结构化编程对代码的最基本的**封装**，一般按照功能组织一段代码
- 封装的目的为了**复用**，减少冗余代码
- 代码更加简洁美观、可读易懂

函数的分类

- 内建函数，如`max()`、`reversed()`等
- 库函数，如`math.ceil()`等
- 自定义函数，使用`def`关键字定义

函数定义

```
def 函数名(参数列表):  
    函数体（代码块）  
    [return 返回值]
```

- 函数名就是标识符，命名要求一样
- 语句块必须缩进，约定4个空格
- Python的函数若没有`return`语句，会隐式返回一个`None`值
- 定义中的参数列表称为**形式参数**，只是一种符号表达（标识符），简称**形参**

函数调用

- 函数定义，只是声明了一个函数，它不能被执行，需要调用执行
- 调用的方式，就是**函数名后加上小括号**，如有必要在括号内填写上参数
- 调用时写的参数是**实际参数**，是实实在在传入的值，简称**实参**

```
def add(x, y): # 函数定义
    result = x + y # 函数体
    return result # 返回值

out = add(4,5) # 函数调用, 可能有返回值, 使用变量接收这个返回值
print(out) # print函数加上括号也是调用
```

上面代码解释:

- 定义一个函数add, 及函数名是add, 能接受2个参数
- 该函数计算的结果, 通过返回值返回, 需要return语句
- 调用时, 通过函数名add后加2个参数, 返回值可使用变量接收
- **函数名也是标识符**
- **返回值也是值**
- 定义需要在调用前, 也就是说调用时, 已经被定义过了, 否则抛NameError异常
- 函数是**可调用的对象**, callable(add)返回True

看看这个函数是不是通用的? 体会一下Python函数的好处

函数参数

函数在定义是要定义好形式参数, 调用时也提供足够的实际参数, 一般来说, 形参和实参个数要一致 (可变参数除外)。

实参传参方式

1、位置传参

定义时def f(x, y, z), 调用使用 f(1, 3, 5), 按照参数定义顺序传入实参

2、关键字传参

定义时def f(x, y, z), 调用使用 f(x=1, y=3, z=5), 使用形参的名字来传入实参的方式, 如果使用了形参名字, 那么**传参顺序就可和定义顺序不同**

要求位置参数必须在关键字参数之前传入, 位置参数是按位置对应的

```
def add(x, y):
    print(x)
    print(y)
    print('-' * 30)

add(4, 5)
add(5, 4) # 按顺序对应, 反过来x和y值就不同

add(x=[4], y=(5,))
add(y=5.1, x=4.2) # 关键字传参, 按名字对应, 无所谓顺序

add(4, y=5) # 正确
add(y=5, 4) # 错误传参
```

切记: 传参指的是**调用时**传入实参, 就2种方式。

下面讲的都是形参定义。

形参缺省值

缺省值也称为默认值，可以在函数定义时，为形参增加一个缺省值。其作用：

- 参数的默认值可以在未传入足够的实参的时候，对没有给定的参数赋值为默认值
- 参数非常多的时候，并不需要用户每次都输入所有的参数，简化函数调用

```
def add(x=4, y=5):  
    return x+y
```

测试调用 `add()`、`add(x=5)`、`add(y=7)`、`add(6, 10)`、`add(6, y=7)`、`add(x=5, y=6)`、`add(y=5, x=6)`、`add(x=5, 6)`、`add(y=8, 4)`、`add(11, x=20)`

能否这样定义 `def add(x, y=5)` 或 `def add(x=4,y)` ？

```
# 定义一个函数login, 参数名称为host、port、username、password  
def login(host='localhost', port=3306, username='root', password='root'):  
    print('mysql://{2}:{3}@{0}:{1}/'.format(host, port, username, password))  
  
login()  
login('127.0.0.1')  
login('127.0.0.1', 3361, 'wayne', 'wayne')  
login('127.0.0.1', username='wayne')  
login(username='wayne', password='wayne', host='www.magedu.com')
```

可变参数

需求：写一个函数，可以对多个数累加求和

```
def sum(iterable):  
    s = 0  
    for x in iterable:  
        s += x  
    return s  
  
print(sum([1,3,5]))  
print(sum(range(4)))
```

上例，传入可迭代对象，并累加每一个元素。

也可以使用可变参数完成上面的函数。

```
def sum(*nums):  
    sum = 0  
    for x in nums:  
        sum += x  
    return sum  
  
print(sum(1, 3, 5))  
print(sum(1, 2, 3))
```

1、可变位置参数

- 在形参前使用 * 表示该形参是可变位置参数，可以接受多个实参
- 它将收集来的实参组织到一个tuple中

2、可变关键字参数

- 在形参前使用 ** 表示该形参是可变关键字参数，可以接受多个关键字参数
- 它将收集来的实参的名称和值，组织到一个dict中

```
def showconfig(**kwargs):
    for k,v in kwargs.items():
        print('{}={}'.format(k,v), end=', ')

showconfig(host='127.0.0.1', port=8080, username='wayne', password='magedu')
```

混合使用

可以定义为下列方式吗？

```
def showconfig(username, password, **kwargs)
def showconfig(username, *args, **kwargs)
def showconfig(username, password, **kwargs, *args) # ?
```

总结：

- 有可变位置参数和可变关键字参数
- 可变位置参数在形参前使用一个星号*
- 可变关键字参数在形参前使用两个星号**
- 可变位置参数和可变关键字参数都可以收集若干个实参，可变位置参数收集形成一个tuple，可变关键字参数收集形成一个dict
- 混合使用参数的时候，普通参数需要放到参数列表前面，可变参数要放到参数列表的后面，可变位置参数需要在可变关键字参数之前

使用举例

```
def fn(x, y, *args, **kwargs):
    print(x, y, args, kwargs, sep='\n', end='\n\n')

fn(3, 5, 7, 9, 10, a=1, b='abc')
fn(3, 5)
fn(3, 5, 7)
fn(3, 5, a=1, b='abc')
fn(x=1, y=2, z=3)
fn(x=3, y=8, 7, 9, a=1, b='abc') # ?
fn(7, 9, y=5, x=3, a=1, b='abc') # ?
```

fn(x=3, y=8, 7, 9, a=1, b='abc'), 错在位置传参必须在关键字传参之前

fn(7, 9, y=5, x=3, a=1, b='abc'), 错在7和9已经按照位置传参了，x=3、y=5有重复传参了

keyword-only参数

先看一段代码

```
def fn(*args, x, y, **kwargs):
    print(x, y, args, kwargs, sep='\n', end='\n\n')

fn(3, 5) #
fn(3, 5, 7) #
fn(3, 5, a=1, b='abc') #
fn(3, 5, y=6, x=7, a=1, b='abc')
```

在Python3之后，新增了keyword-only参数。

keyword-only参数：在形参定义时，在一个*星号之后，或一个可变位置参数之后，出现的普通参数，就已经不是普通的参数了，称为keyword-only参数。

```
def fn(*args, x):
    print(x, args, sep='\n', end='\n\n')

fn(3, 5) #
fn(3, 5, 7) #
fn(3, 5, x=7)
```

keyword-only参数，言下之意就是这个参数必须采用关键字传参。

可以认为，上例中，args可变位置参数已经截获了所有位置参数，其后的变量x不可能通过位置传参传入了。

思考：def fn(**kwargs, x)可以吗？

```
def fn(**kwargs, x):
    print(x, kwargs, sep='\n', end='\n\n')
```

直接语法错误了。

可以认为，kwargs会截获所有关键字传参，就算写了x=5，x也没有机会得到这个值，所以这种语法不存在。

keyword-only参数另一种形式

* 星号后所有的普通参数都成了keyword-only参数。

```
def fn(*, x, y):
    print(x, y)
fn(x=6, y=7)
fn(y=8, x=9)
```

Positional-only参数

Python 3.8 开始，增加了最后一种形参类型的定义：Positional-only参数。（2019年10月发布3.8.0）

```
def fn(a, /):  
    print(a, sep='\n')  
  
fn(3)  
fn(a=4) # 错误，仅位置参数，不可以使用关键字传参
```

参数的混合使用

```
# 可变位置参数、keyword-only参数、缺省值  
def fn(*args, x=5):  
    print(x)  
    print(args)  
fn() # 等价于fn(x=5)  
fn(5)  
fn(x=6)  
fn(1,2,3,x=10)
```

```
# 普通参数、可变位置参数、keyword-only参数、缺省值  
def fn(y, *args, x=5):  
    print('x={}, y={}'.format(x, y))  
    print(args)  
fn() #  
fn(5)  
fn(5, 6)  
fn(x=6) #  
fn(1, 2, 3, x=10)  
fn(y=17, 2, 3, x=10) #  
fn(1, 2, y=3, x=10) #  
fn(y=20, x=30)
```

```
# 普通参数、缺省值、可变关键字参数  
def fn(x=5, **kwargs):  
    print('x={}'.format(x))  
    print(kwargs)  
fn()  
fn(5)  
fn(x=6)  
fn(y=3, x=10)  
fn(3, y=10)  
fn(y=3, z=20)
```

参数规则

参数列表参数一般顺序是：positional-only参数、普通参数、缺省参数、可变位置参数、keyword-only参数（可带缺省值）、可变关键字参数。

注意：

- 代码应该易读易懂，而不是为难别人
- 请按照书写习惯定义函数参数

```
def fn(a, b, /, x, y, z=3, *args, m=4, n, **kwargs):
    print(a, b)
    print(x, y, z)
    print(m, n)
    print(args)
    print(kwargs)
    print('-' * 30)

def connect(host='localhost', user='admin', password='admin', port='3306',
            **kwargs):
    print(host, port)
    print(user, password)
    print(kwargs)

connect(db='cddb') # 参数的缺省值把最常用的缺省值都写好了
connect(host='192.168.1.123', db='cddb')
connect(host='192.168.1.123', db='cddb', password='mysql')
```

- 定义最常用参数为普通参数，可不提供缺省值，必须由用户提供。注意这些参数的顺序，最常用的先定义
- 将必须使用名称的才能使用的参数，定义为keyword-only参数，要求必须使用关键字传参
- 如果函数有很多参数，无法逐一定义，可使用可变参数。如果需要知道这些参数的意义，则使用可变关键字参数收集

参数解构

```
def add(x, y):
    print(x, y)
    return x + y

add(4, 5)
add((4, 5)) # 可以吗?
t = 4, 5
add(t[0], t[1])
add(*t)
add(*(4, 5))
add(*[4, 5])
add(*{4, 5}) # 注意有顺序吗?
add(*range(4, 6))

add(*{'a':10, 'b':11}) # 可以吗?
add(**{'a':10, 'b':11}) # 可以吗?
add(**{'x':100, 'y':110}) # 可以吗?
```

参数解构：

- 在给函数提供实参的时候，可以在可迭代对象前使用 * 或者 ** 来进行结构的解构，提取出其中所有元素作为函数的实参
- 使用 * 解构成位置传参
- 使用 ** 解构成关键字传参
- 提取出来的元素数目要和参数的要求匹配

```
def add(*iterable):
    result = 0
    for x in iterable:
        result += x
    return result
```

```
add(1, 2, 3)
add(*[1, 3, 5])
add(*range(5))
```

3.8以后, 下面就不可以使用字典解构后的关键字传参了

```
def add(x, y, /): # 仅位置形参
    print(x, y)
    return x + y
```

```
add(**{'x':10, 'y':11})
```

练习

- 编写一个函数, 能够接受至少2个参数, 返回最小值和最大值

下面几种实现, 哪一种好些?

```
# 使用内建函数完成, 谁好
def get_values(a, b, *args): # 最大值、最小值
    src = (a, b, *args)
    mx, mn = max(src), min(src)

def get_values(x, y, *args): # 排序
    max, _, min = sorted((x, y, *args))
    return max, min
```

```
# 自己实现
def get_values(x, y, *args):
    # 假设类型都一样
    # print(x, y, args)
    max, min = (x, y) if x > y else (y, x)
    for i in args:
        if i > max:
            max = i
        elif i < min:
            min = i
    return min, max

print(get_values(*range(10)))
```

函数返回值

先看几个例子

```
# return语句之后可以执行吗？
def showplus(x):
    print(x)
    return x + 1
    print('~~end~~') # return之后会执行吗？

showplus(5)

# 多条return语句都会执行吗
def showplus(x):
    print(x)
    return x + 1
    return x + 2

showplus(5)

# 下例多个return可以执行吗？
def guess(x):
    if x > 3:
        return "> 3"
    else:
        return "<= 3"

print(guess(10))

# 下面函数执行的结果是什么
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？
```

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
- 返回值的作用：结束函数调用、返回“返回值”

能够一次返回多个值吗？

```
def showvalues():  
    return 1, 3, 5
```

showvalues() # 返回了多个值吗?

- 函数不能同时返回多个值
- return 1, 3, 5 看似返回多个值，隐式的被python封装成了一个元组
- x, y, z = showlist() 使用解构提取返回值更为方便

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

```
def foo():  
    x = 100  
  
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

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

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

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

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

outer() # 可以吗？
inner() # 可以吗？
```

内部函数inner不能在外边直接使用，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

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

    inner()
    print("outer {}".format(o))

outer1() # 执行后，打印什么


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

    inner()
    print("outer {}".format(o))

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

```
print(ord('中'), hex(ord('中')), '中'.encode(), '中'.encode('gbk'))

chr(20013) # '中'
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？

```
x = 5
def foo():
    x += 1
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，就被右边拿来做法加1操作了

如何解决这个常见问题？

global语句

```
x = 5
def foo():
    global x # 全局变量
    x += 1
    print(x)
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

```
# 有错吗？
def foo():
    global x
    x += 1
    print(x)
foo()
```

```
# 有错吗?
def foo():
    global x
    x = 10
    x += 1
    print(x)
foo()
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`，也不会再在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

```
def counter():
    c = [0]
    def inc():
        c[0] += 1 # 报错吗? 为什么 # line 4
        return c[0]
    return inc

foo = counter() # line 8
print(foo(), foo()) # line 9
c = 100
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？

```
def counter():
    count = 0
    def inc():
        count += 1
        return count
    return inc

foo = counter()
print(foo(), foo())
```

上例一定出错，使用global可以解决

```
def counter():
    global count
    count = 0
    def inc():
        global count
        count += 1
        return count
    return inc

foo = counter()
print(foo(), foo())
count = 100
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。

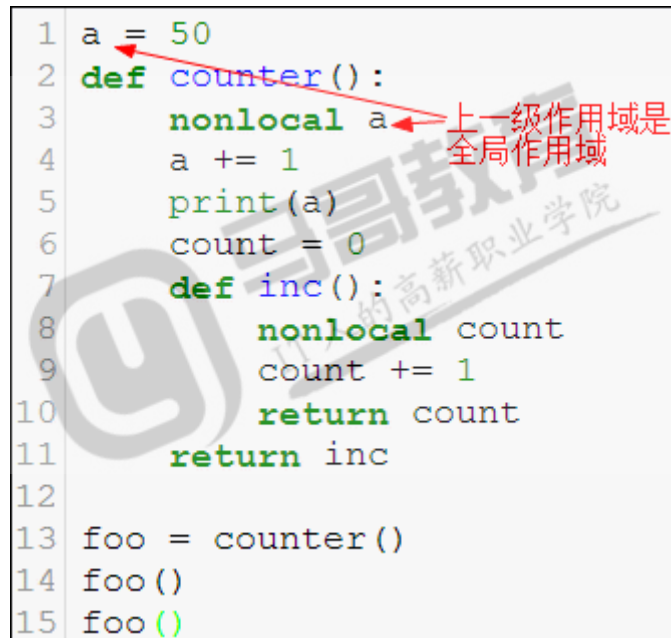
```
def counter():
    count = 0
    def inc():
        nonlocal count # 声明变量count不是本地变量
        count += 1
        return count
    return inc

foo = counter()
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。

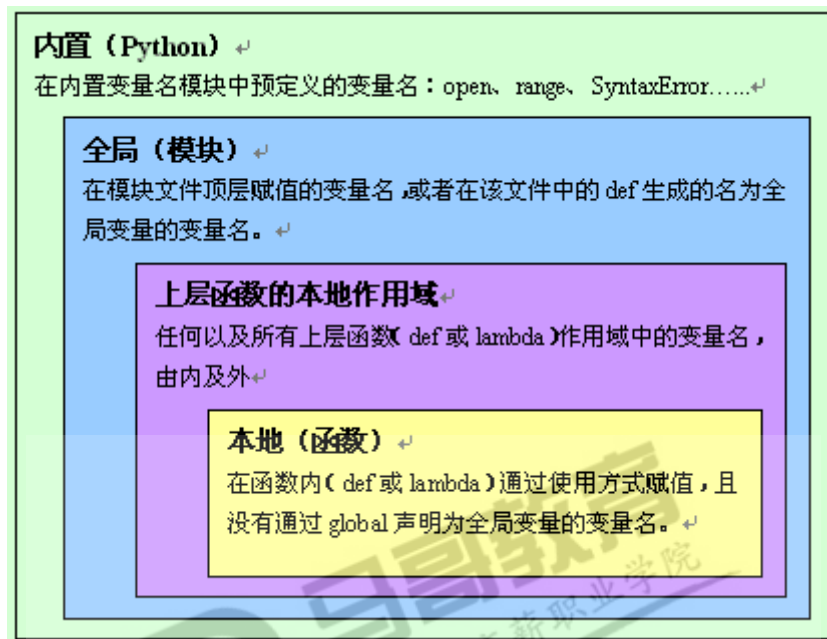
Python程序结束时，所有对象销毁。

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 如果已经取完, 继续取抛StopIteration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 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)

```
lst = [1, 3, 5, 7, 9]

it = iter(lst) # 返回一个迭代器对象
print(next(it))
print(next(it))

for i, x in enumerate(it, 2):
    print(i, x)
#print(next(it)) # StopIteration
print()

for x in reversed(lst):
    print(x)
```

