

第五章 函 数

目录

- 函数基础
- 参数传入
- 特殊函数
- 迭代器与生成器
- 变量作用域

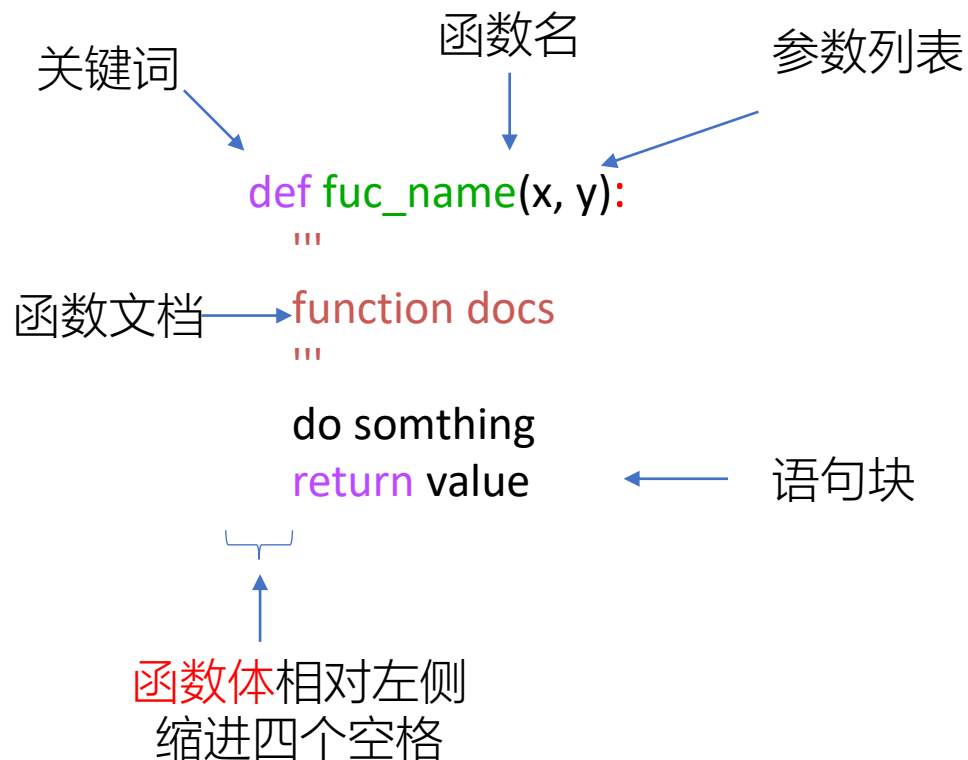
函数基础

定义函数

- 函数的基本样式

函数是能够完成特定功能的代码块，可以在程序中重复使用，提高程序的执行效率。

基本所有的高级语言都支持函数，python内部有很多内置函数，可以直接调用。



```
1 def my_abs(x):  
2     if x>0:  
3         return x  
4     else:  
5         return -x  
6 print(my_abs(-3))
```

3

函数返回值

- 返回值

- 关键词return，函数体内部,语句执行时，**执行到return时，函数体执行完毕**
- 如果没有return语句，默认是return None
- 可以返回多个值

```
1 def sum(arr):
2     x=0
3     for i in arr:
4         x+=i
5     return x
6 y=sum([1, 2, 3, 4, 5])
7 print(y)
```

15

```
1 def sum(arr):
2     x=0
3     for i in arr:
4         x+=i
5     #return x
6 y=sum([1, 2, 3, 4, 5])
7 print(y)
```

None

```
1 def sum(arr):
2     x=0
3     for i in arr:
4         x+=i
5         return x
6 y=sum([1, 2, 3, 4, 5])
7 print(y)
```

1

Return的位置影响返回值!

调用函数

- 函数举例:

```
def add(x, y):  
    """  
    This is my first function.  
    """  
    r = x + y  
    return r  
a = add(3, 4)  
print(a)
```

程序执行结果为:

7

```
#自定义 len() 函数  
def my_len(str):  
    length = 0  
    for c in str:  
        length = length + 1  
    return length  
#调用自定义的 my_len() 函数  
length = my_len("http://c.biancheng.net/python/")  
print(length)
```

```
#再次调用 my_len() 函数  
length = my_len("http://c.biancheng.net/shell/")  
print(length)
```

程序执行结果为:

30

29

函数是对象

- 理解函数是对象——Python中万物皆对象
 - 对象和对象的调用（运行）
 - 对象的属性

属性

• 函数对象

- 函数作为对象可以赋值给一个变量、可以作为元素添加到集合对象中、可作为参数值传递给其它函数，还可以当做函数的返回值，这些特性就是**第一类对象**所特有的
- 函数身为一个对象，拥有**对象模型的三个通用属性**：**id**、**类型**和**值**

```
def foo():  
    print('from foo')  
foo()
```

```
print(id(foo))  
print(type(foo))  
print(foo)
```

```
from foo  
85791200  
<class 'function'>  
<function foo at 0x00000000051D11E0>
```

(1) Identity

Python中每个对象有一个唯一标识identity，一个对象的标识在对象被创建后不再改变。可以认为**对象的identity是对象在内存中的地址**，其值可以由内置函数id()求得。is操作符可以比较两个对象的identity是否相同，即两个对象是否是同一个。

(2) Type

type 是对象的类型，决定了对象保存值的类型、可以执行的操作，以及所遵循的规则。可以使用内置函数type()查看一个对象的类型。因为Python中一切皆是对象，type() 函数返回的也是对象，而不是简单字符串。

(3) value

对象表示的数据。值是可变的，值可变的对象称为mutable对象，值一经创建不可再变的对象称为immutable对象。一个对象的可变性由其类型决定，例如 数字, 字符串 和元组是不可变的, 而字典和列表则是可变的。

嵌套函数

- 向函数中“传函数对象”
- 嵌套函数
 - 返回函数对象
 - 调用返回值

```
def doc():  
    name='new student'  
    def doc1():  
        name='new student1'  
        def doc2():  
            name='new student2'  
            print("name:", name)  
        doc2()  
    doc1()  
doc()  
print(doc())
```

```
name: new student2  
name: new student2  
None
```

嵌套函数

- 嵌套函数

- 在函数内定义的函数只能在函数内调用，外面无法调用

```
def get(x):  
    def clean(t): #嵌套在get函数内的clean函数  
        a = t*2  
        return a  
    new_x = clean(x) #在get函数内调用clean函数  
    return new_x
```

```
print(get(5))
```

10

在函数内部又嵌套定义一个函数—闭包

```
1 def get(x):  
2     def clean(t):  
3         a = t*2  
4         return a  
5     new_x = clean(x)  
6     return new_x  
7  
8 #print(get(5))  
9 print(clean(5))
```

```
-----  
-  
NameError  
<ipython-input-6-81eeb5d637e0> in <module>  
7  
8 #print(get(5))  
----> 9 print(clean(5))  
  
NameError: name 'clean' is not defined
```

装饰器

- 装饰器

- 用于拓展原来函数功能的一种函数
- 函数特殊之处
 - 返回值也是一个函数调用
 - 给函数增加新的功能，而不用更改原函数的代码

装饰器

- 一个简单的装饰器:

- w1函数是装饰器，它的参数是一个函数，然后返回值也是一个函数
 - 作为参数的函数f1()或f2()就在返回函数inner()的内部执行
 - 在函数f1()和f2()前面加上@w1，f1()函数和f2()就相当于被注入了验证权限功能，现在只要调用f1()或f2()，它就已经变身为“新的功能更多”的函数
- 装饰器拓展了原来函数的功能
- 既不需要侵入函数内更改代码，也不需要重复执行原函数

原来的函数

```
def f1():  
    print('f1 called')  
def f2():  
    print('f2 called')
```

附加新功能的函数

```
def w1(func):  
    def inner():  
        print('...验证权限...') #赋予的新的功能  
        func() #原来的功能  
    return inner
```

修饰的对象

```
@w1  
def f1(): #原函数  
    print('f1 called')
```

修饰的对象

```
@w1  
def f2(): #原函数  
    print('f2 called')
```

f1()
f2()



输出结果为

f1 called
...验证权限...
f2 called
...验证权限...

装饰器

“@函数”修饰的函数不再是原来的函数，而是被替换成一个新的东西（取决于装饰器的返回值），即如果装饰器函数的返回值为普通变量，那么被修饰的函数名就变成了变量名；同样，如果装饰器返回的是一个函数的名称，那么被修饰的函数名依然表示一个函数。

#funA 作为装饰器函数

```
def funA(fn):  
    print("C语言中文网")  
    fn() # 执行传入的fn参数  
    print("http://c.biancheng.net")  
    return "装饰器函数的返回值"
```

@funA

```
def funB():  
    print("学习 Python")
```

```
def funA(fn):  
    # 定义一个嵌套函数  
    def say(arc):  
        print("Python教程:",arc)  
        fn(arc)  
    return say
```

@funA

```
def funB(arc):  
    print("funB():", arc)
```

```
funB("http://c.biancheng.net/python")
```

```
In [65]: #funA 作为装饰器函数  
def funA(fn):  
    print("C语言中文网")  
    fn() # 执行传入的fn参数  
    print("http://c.biancheng.net")  
    return "装饰器函数的返回值"  
  
@funA  
def funB():  
    print("学习 Python")  
  
type(funB)
```

C语言中文网
学习 Python
<http://c.biancheng.net>

Out[65]: str

```
In [66]: def funA(fn):# 定义一个嵌套函数  
def say(arc):  
    print("Python教程:",arc)  
    fn(arc)  
    return say  
  
@funA  
def funB(arc):  
    print("funB():", arc)  
funB("http://c.biancheng.net/python")  
  
type(funB)
```

Python教程: <http://c.biancheng.net/python>
funB(): <http://c.biancheng.net/python>

Out[66]: function

装饰器

```
import time
def timing(func):# 装饰器函数
    def inner():
        start = time.time()
        time.sleep(0.12)
        func() # 被装饰函数
        end = time.time()
        print('现在代码运行时长为: %s' % (end - start))
    return inner
@ timing
def func():#原来的函数
    start = time.time()
    time.sleep(0.12)
    print('看看代码运行了多长时间! ')
    end = time.time()
    print('原来代码运行时长为: %s'%(end - start))
func()
```

运行结果

看看代码运行了多长时间!
原来代码运行时长为: 0.12504243850708008
现在代码运行时长为: 0.25005030632019043

装饰器

- 函数装饰器嵌套

上面示例中，都是使用一个装饰器的情况，但实际上，Python 也支持多个装饰器，比如：

```
1.@funA
2.@funB
3.@funC
4.def fun():
5.#...
```

上面程序的执行顺序是里到外，所以它等效于下面这行代码：

```
fun = funA( funB ( funC (fun) ) )
```

编个程序你试一试？！

参数传入

必要参数

- 函数调用时必须传的参数

- 位置传入：按照输入参数列表的位置确认

```
#多项式  $s = 1 + 2x + y^2 + zy$   
def polynomial(x,y,z):  
     $s = 1 + 2x + y^2 + zy$   
    return s  
#按位置输入  
print(polynomial(1,2,3))
```

13

- 关键词：直接在参数列表里设定关键词确定

```
#多项式  $s = 1 + 2x + y^2 + zy$   
def polynomial(x,y,z):  
     $s = 1 + 2x + y^2 + zy$   
    return s  
#关键词  
print(polynomial(x=1,y=2,z=3))
```

13

必要参数

- 可以将位置和关键词的方法混合使用

#位置和关键词方法混合使用

```
print(polynomial(1,y=2,z=3))
```

13

- 如果传入的第一个参数是用关键词传入的，那么后面每个参数都需要是关键词传入，否则会出现语法错误

```
print(polynomial(x=1, 2,z=3))
```

```
File "<ipython-input-7-27e4b7f5d196>", line 1
```

```
    print(polynomial(x=1, 2,z=3))
```

```
        ^
```

```
SyntaxError: invalid character in identifier
```

默认参数

- 输入的参数可以是事先设定好赋值，也就是默认值。在调用函数的时候，可以不输入默认参数，函数内部会直接调用默认参数值。例如默认 $z=3$

```
def polynomial(x,y,z=3):  
    s = 1 + 2*x + y*y + z*y  
    return s  
#调用函数  
print(polynomial(x=1,y=2))
```

13

```
1 def polynomial(x, y, z=3):  
2     s = 1 + 2*x + y*y + z*y  
3     return s  
4 #调用函数  
5 print(polynomial(x=1, y=2, z=5))
```

默认参数也可以进行修改

17

默认参数

- 默认参数的默认值是可以修改的，将上面的 z 值传入设置为 4
- 默认参数的位置：需要注意的是，默认参数必须放到参数列表的末位

#调用函数，并输入修改的 z 参数
`print(polynomial(1,2,z=4))`

15

```
1 def polynomial(x, y=3, z):
2     s = 1 + 2*x + y*y + z*y
3     return s
4 #调用函数
5 print(polynomial(x=1, y=2, z=5))
```

```
File "<ipython-input-5-e8a62ab300f2>", line 1
def polynomial(x, y=3, z):
```

SyntaxError: non-default argument follows default argument

不定长参数

- 为了解决不确定需要传入参数个数的情况
- 有两种不定长参数收集方法
 - `*args`
 - `**kwargs`

不定长参数

- *args 用来将参数打包成元组tuple给函数体调用

```
def print_keywords(x,*args):
```

```
    for i in args:
```

```
        print(x+i)
```

```
    return
```

```
print_keywords(1,1,2,3)#x=1,arg=(1,2,3)
```

```
2
```

```
3
```

```
4
```

- 如果输入是一个列表 list ， 那么可以用 *list 的方式传入

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

```
print_keywords(1,*a)
```

```
2
```

```
3
```

```
4
```

不定长参数

- *args 传入的时候，如果调用函数使用关键词传入参数时会出错(不能使用)

`print_keywords(1,y=1, z=2, m=3)#用关键词参数传入`

```
-----  
TypeError                                Traceback (most recent call  
last)  
<ipython-input-15-afe9cee154c5> in <module>()  
----> 1 print_keywords(1,y=1, z=2, m=3)#用关键词参数传  
入  
  
TypeError: print_keywords() got an unexpected keyword  
argument 'y'
```

不定长参数

- `**kwargs` 传入的参数是 字典dict 类型

```
def test(**kwargs):  
    print(kwargs)  
    keys = kwargs.keys()  
    value = kwargs.values()  
    print(keys)  
    print(value)
```

```
test(a=1, b=2, c=3, d=4)
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4}  
dict_keys(['a', 'b', 'c', 'd'])  
dict_values([1, 2, 3, 4])
```

```
def print_keywords(x, **kwargs):  
    for i in kwargs:  
        print(x + kwargs[i])  
    print(kwargs)  
    return  
print_keywords(x=1, y=1, z=2, m=3)
```

```
2  
3  
4  
{'y': 1, 'z': 2, 'm': 3}
```


不定长参数

不定长参数传入可以混合使用 *args,**kwargs

```
def func(a,*b,**c):  
    return a,b,c  
result=func(1,9,2,7,3,x=5,y=3)  
x=6  
print(result)
```

结果是

(1, (9, 2, 7, 3), {'x': 5, 'y': 3})

```
def func(a,*b,**c):  
    return a,c  
result=func(1,9,2,7,3,x=5,y=3)  
x=6  
print(result)
```

结果是

(1, {'x': 5, 'y': 3})

特殊函数

lambda函数

- lambda()函数-匿名函数

- lambda 函数可以接收任意多个参数 (包括可选参数) 并且返回单个表达式的值。

polynomial = lambda x,y,z:1 + 2*x + y*y + z*y
print(polynomial(1,2,3))

13

输入参数

函数主体+返回变量

name = lambda [list] : 表达式 ↔
def name(list):
 return 表达式
name(list)

```
1 def polynomial(x, y, z):  
2     X=1+2*x+y*y+z*y  
3     return X  
4 print(polynomial(1, 2, 3))
```

13

- 对于单行函数, 使用 lambda 表达式可以省去定义函数的过程, 让代码更加简洁;
- 对于不需要多次复用的函数, 使用 lambda 表达式可以在用完之后立即释放, 提高程序执行的性能。

lambda函数

和列表联合使用

```
L = [lambda x:x**2, lambda x:x**3, lambda x:x**4]
for f in L:
    print( f(2))
```

4
8
16

也可以如下面这样调用

```
L = [lambda x:x**2, lambda x:x**3, lambda x:x**4]
for f in L:
    print( f(2))
print( L[0](3))
```

4
8
16
9

<https://www.cnblogs.com/xisheng/p/7301245.html>

map函数

• map()函数—映射

- Python 内置的高阶函数，它接收一个函数和一个列表，根据提供的函数对指定序列做映射通过函数依次作用在列表的每个元素上，得到一个新的列表并返回
- 不改变原有的列表，而是返回一个新的列表

语法 map(function, iterable, ...)

- function -- 函数
- iterable -- 一个或多个序列

```
def f(x):#将序列中各元素平方
    return x*x
print (list(map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])))
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
#将首字母大写，其余小写
```

```
def format_name(s):
    s1=s[0:1].upper()+s[1:].lower();
    return s1;
print(list(map(format_name, ['adam', 'LISA', 'barT'])))
```

```
['Adam', 'Lisa', 'Bart']
```

高阶函数：一个函数可以作为参数传给另外一个函数，或者一个函数的返回值为另外一个函数（若返回值为该函数本身，则为递归），满足其一则为高阶函数。

filter函数

- **filter()函数—过滤**

- 接收一个函数和一个序列, **过滤掉**不符合条件的元素,返回由符合条件元素组成的新列表。
- 把传入的函数依次作用于每个元素, 然后**根据返回值是True还是False**决定保留还是丢弃该元素

语法 filter(function, iterable)

- function -- 判断函数。
- iterable -- 可迭代对象。

#删掉偶数, 只保留奇数

```
def is_odd(n):  
    return n % 2 == 1
```

```
a = filter(is_odd, [1, 2, 4, 5, 6, 9, 10, 15])  
print(list(a))
```

```
[1, 5, 9, 15]
```

reduce函数

- **reduce()函数—累积**

- 接收一个有两个参数的函数和一个可迭代的序列(或有初始值)，不断对序列中的每个元素进行迭代运算，每次运算的结果都作为第二次运算的参数，和最后一个元素的运算结果作为reduce函数的返回值。

语法 `reduce(function, iterable[, initializer])`

`function` -- 函数，有两个参数

`iterable` -- 可迭代对象

`initializer` -- 可选，初始参数

```
from functools import reduce
```

```
def mm(x, y):      # 两数相乘
```

```
    return x * y
```

```
sum1 = reduce(mm, [1,2,3,4,5]) # 计算列表积: 1*2*3*4*5
```

```
sum2 = reduce(lambda x, y: x*y, [1,2,3,4,5]) # 使用 lambda 匿名函数
```

```
print(sum1)
```

```
print(sum2)
```

```
from functools import reduce
```

结果

120

120

注意: Python3.x `reduce()` 已经被移到 `functools` 模块里，如果我们要使用，需要引入 `functools` 模块来调用 `reduce()` 函数：

eval函数

- eval()函数

- eval() 函数用来执行一个字符串表达式，并返回表达式的值。

语法 `eval(expression[, globals[, locals]])`
 `expression` -- 表达式。
 `globals` -- 变量作用域，全局命名空间，如果被提供，则必须是一个字典对象。
 `locals` -- 变量作用域，局部命名空间，如果被提供，可以是任何映射对象。

在后两个参数省略的情况下，eval在当前的作用域执行

```
>>>x = 7
>>> eval('3 * x')
21
>>> eval('pow(2,2)')
4
>>> eval('2 + 2')
4
>>> n=81
>>> eval("n + 4")
85
```

```
a = "[[1,2], [3,4], [5,6], [7,8], [9,0]]"
b = eval(a)
b
Out[3]: [[1, 2], [3, 4], [5, 6], [7, 8], [9, 0]]
type(b)
Out[4]: list
a = "{1: 'a', 2: 'b'}"
b = eval(a)
b
Out[7]: {1: 'a', 2: 'b'}
type(b)
Out[8]: dict
a = "([1,2], [3,4], [5,6], [7,8], (9,0))"
b = eval(a)
b
Out[11]: ([1, 2], [3, 4], [5, 6], [7, 8], (9, 0))
```


eval函数

在globals指定的情况下：

```
a=10;  
g={'a':4}  
print(eval("a+1",g))  
执行结果为： 5
```

eval中提供了globals参数，这时候eval的作用域就是g指定的这个字典了，也就是外面的a=10被屏蔽掉了，eval是看不见的，所以使用了a为4的值。

在 locals 指定的情况下：

```
a=10  
b=20  
c=30  
g={'a':6,'b':8}  
L={'b':100,'c':10}  
print(eval('a+b+c',g,L))  
执行的结果为： 116
```

当locals和globals起冲突时，locals是起决定作用的，这在很多编程语言里都是一样的，是作用域的覆盖问题，当前指定的小的作用域会覆盖以前大的作用域

eval函数—应用注意问题

```
s="abck"
```

```
print(eval(s))
```

执行的结果为：NameError: name 'abck' is not defined

当它解析到这个表达式是不可以计算后，它就会查找它是不是一个变量的名字，如果是一个变量的名字，那么它会输出这个变量的内容，否则就会产生这种报错。

```
s="abck"
```

```
print(eval('s'))
```

执行的结果为：abck

一个是有引号括起来的，一个是没有的，引号括起来代表字符串，虽然不可以求值，但是是有意义的，可以进行输出，而没引号的便无法判断“身份”了，只能当做变量名进行解析，而abck并不是一个变量名，所以就报错了。

```
s='["a","b","c"]'
```

```
print(eval(s))
```

执行的结果为：['a', 'b', 'c']

```
a=10
```

```
b=20
```

```
c=30
```

```
s='[a,b,c]'
```

```
print(eval(s))
```

执行的结果为：[10, 20, 30]

eval检查到列表的 '[' 符号时，是会对里面的元素进行解析的，这里a、b、c显然不是具体的数据，便去查找它们是否是变量名，然后确认是变量名后，用它们的内容替换掉它。

变量作用域

局部变量

- 局部变量是指那些有固定的变量作用域，只有在此作用域内才能调用的变量
 - 比如，函数内的局部变量的作用域仅限于函数内，新建函数，命名为mean()，用于求平均

#求1到100累加的平均数

```
def mean(x):  
    global length #初始化全局变量length  
    length = 100 # 全局变量length赋值  
    sum_x = 0  
    for i in x:  
        sum_x += i  
    return sum_x/length  
print(mean(range(101)))
```

50.5

```
#求1到100累加的平均数  
def mean(x):  
    global length#初始化全局变量length  
    length=100#全局变量length赋值  
    sum_x = 0  
    for i in x:  
        sum_x += i  
    return sum_x/length  
print(mean(range(101)))
```

50.5

局部变量

- 在关键词“def”定义函数的范围内，**新定义或者新赋值的变量都是局部变量**
- 在该函数之外引用函数内命名的变量的时候会报错

```
def echo(p):  
    L = "hello"+p # 定义局部变量L  
    print(L)  
echo("world")  
print(L) #尝试输出局部变量L
```

```
helloworld
```

```
-----  
NameError                                Traceback (most recent  
call last)
```

```
<ipython-input-127-50cd5d7d5faf> in <module>()  
      3     print(L)
```

```
      4 echo("world")
```

```
----> 5 print(L)
```

```
NameError: name 'L' is not defined
```

局部变量

- 在局部区域引用局部区域以外的变量，也会引起报错。因为内部函数有引用外部函数的同名变量或者全局变量，并且对这个变量有修改的时候，此时 Python 会认为它是一个局部变量，而函数中并没有 x 的定义和赋值，所以报错。

```
A=0
def A_add_one():
    A=A+1
    return A
A_add_one()
```

```
1 A=0
2 def A_add_one():
3     global A
4     A=A+1
5     return A
6 A_add_one()
7
```

UnboundLocalError Traceback (most recent call last)

<ipython-input-11-e7737b429c28> in <module>

```
3     A=A+1
4     return A
--> 5 A_add_one()
```

<ipython-input-11-e7737b429c28> in A_add_one()

```
1 A=0
2 def A_add_one():
--> 3     A=A+1
4     return A
5 A_add_one()
```

UnboundLocalError: local variable 'A' referenced before assignment

全局变量

- 全局变量是相对局部变量而言，作用范围在全局，即在初始定义赋值后，无论是函数、类、lambda函数内都可以引用全局变量
 - 在关键词"def"、"class"、"lambda"之外定义的变量，都作为全局变量
 - 在mean()函数内定义的length变量移至关键词"def"之外即变为全局变量

```
length = 100# 全局变量
def mean(x):
    sum_x = 0
    for i in x:
        sum_x += i
    return sum_x/length
print(mean(range(101)))
```

50.5

局部变量转化为全局变量

- 将函数内部定义的变量变为函数外可以引用的全局变量
- 比如前面的mean()函数中最开始“length”定义时是局部变量，现只需要在定义变量时使用关键词“global”即可将其定义为全局变量

```
length = 100# 全局变量
def mean(x):
    sum_x = 0
    global length
    for i in x:
        sum_x += i
    return sum_x/length
print(mean(range(101)))
```

50.5

局部变量转化为全局变量

```
def f():  
    global x  
    x = 10  
    x += 1  
    print(x)  
#print(x)      # 这句会报错, 因为在函数外部  
  
f()
```

11

```
x = 5  
def f():  
    global x  
    x += 1  
    print(x)      # 6  
print(x)          # 5  注意与上面代码的区别  
  
f()
```

5

6

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。尽量不用global, 学习它就是为了深入理解变量作用域。

同名变量引用

- 当某局部变量和全局变量都有相同变量名时，函数内引用该变量会直接调用函数内定义的局部变量

```
# 求1到100累加的平均数
length = 1
def mean(x):
    length = 100# 没有声明为global，为局部变量
    sum_x = 0
    for i in x:
        sum_x += i
    return sum_x/length
print(mean(range(101)))
```

50.5

LEGB原理介绍

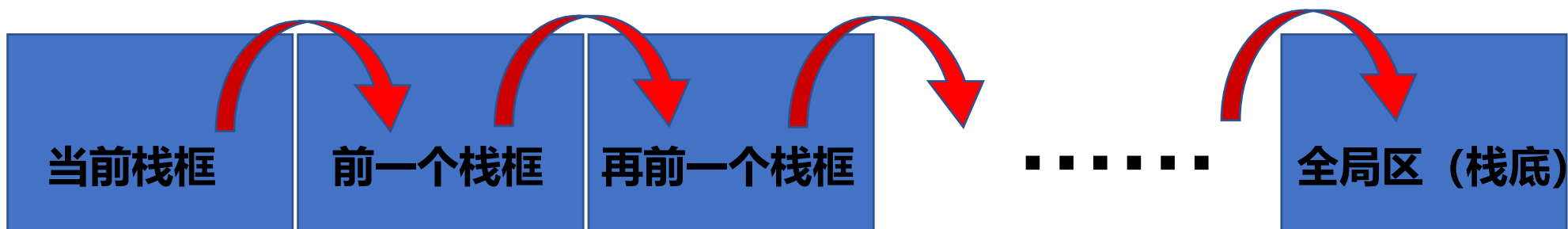
- LEGB含义解释

字母	英语	释义	简称	作用空间
L	Local(function)	当前函数内的作用域	局部作用域	局部
E	Enclosing Functions Locals	外部嵌套函数的作用域	嵌套作用域	局部
G	Global(module)	函数外部所在的命名空间	全局作用域	全局
B	Built In(python)	Python内置模块的命名空间	内建作用域	内置

寻找变量的调用顺序，就是采用的LEGB原则（就近原则）
寻找顺序：从下往上、从里往外

LEGB原理介绍

- 当一个函数体内需要引用一个变量的时候，会按照如下顺序查找：
 - 首先查找局部变量（Locals）
 - 如果找不到该名称的局部变量，则去函数体的外层去寻找局部变量（Enclosing function locals）。同样适用于嵌套函数的情况
 - 若函数体外部的局部变量中也要不到该名称的局部变量，则从全局变量（Global）中寻找
 - 如果再找不到，就去找内置库（Built-in）



LEGB原理介绍

```
#!/usr/bin/env python
# encoding: utf-8

x = 1

def foo():
    x = 2
    def innerfoo():
        x = 3
        print('locals ', x)
    innerfoo()
    print('enclosing function locals ', x)

foo()
print('global ', x)
```

```
locals 3
enclosing function locals 2
global 1
```

```
#!/usr/bin/env python
# encoding: utf-8

x = 1

def foo():
    x = 2
    def innerfoo():
        #x = 3
        print('locals ', x)
    innerfoo()
    print('enclosing function locals ', x)

foo()
print('global ', x)
```

```
locals 2
enclosing function locals 2
global 1
```

当注释掉x = 3以后，函数innerfoo内部查找到的x是x = 2

在上述两个例子中，从内到外，依次形成四个命名空间：

- def innerfoo(): Local，即函数内部命名空间；
- def foo(): Enclosing function locals；外部嵌套函数的名字空间
- module(文件本身)：Global(module)；函数定义所在模块（文件）的名字空间
- Python内置模块的名字空间：Builtin

迭代器与生成器

创建迭代器

- 迭代器是访问集合元素的一种方式
- 迭代器的特点：
 - 迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束
 - 迭代器只能往前不会后退
 - 迭代器不要求事先准备好整个迭代过程中所有的元素，仅仅在迭代到某个元素时才计算该元素，而在这之前或之后，元素可以不存在或者被销毁，这个特点使得它特别适合用于遍历一些巨大的或是无限的集合
 - 不能随机访问集合中的某个值，只能从头到尾依次访问
 - 便于循环比较大的数据集合，节省内存

迭代器的函数

- 字符串，列表或元组对象都可用于创建迭代器
- 迭代器有两个基本的方法：iter() 和 next()
 - iter() 函数用来生成迭代器
 - next() 返回迭代器的下一个项目
- 迭代器对象可以使用循环语句进行遍历

```
list1=['b','o','y','a']
it1=iter(list1)
print(next(it1))
print(next(it1))
print('=====')
for x in it1:
    print(x, end='\n') #结尾换行
```

```
b
o
=====
y
a
```

```
# 首先获得Iterator对象:
it = iter([1, 2, 3, 4, 5])
# 循环:
while True:
    try:
        # 获得下一个值:
        x = next(it)
        print(x)
    except StopIteration:
        # 遇到StopIteration就退出循环
        break
```

```
1
2
3
4
5
```


生成器

- 生成器 (generator)

- 使用了 yield 的函数
- 跟普通函数不同的是，生成器是一个返回迭代器的函数，只能用于迭代操作
- yield 作用是返回一个生成器，它会保存当前函数状态，记录下一次函数被调用next的时候运行状态

- 使用方法

- 在调用生成器运行的过程中，每次遇到 yield 时函数会暂停并保存当前所有的运行信息，返回 yield 的值，并在下一次执行 next() 方法时从当前位置继续运行
- yield作用就是返回一个生成器，它会保存当前函数状态，记录下一次函数被调用next的时候运行状态

创建生成器

- yield可以使函数中断，并保存中断状态，中断后，代码可以继续往下执行，过一段时间还可以再重新调用这个函数，从上次yield的下一句开始执行

```
def create_counter(n):  
    print ("create counter")  
    while True:  
        yield n#循环在此中断，返回n，再次运行函数时从此处开始  
        print('increment n')  
        n += 1
```

```
cnt = create_counter(2)  
print (cnt)  
print (next(cnt))#在yield后中断，返回n，但不打印'increment n'  
print (next(cnt))#在yield后运行，首先打印'increment n'
```

```
<generator object create_counter at 0x0000000005585FC0>  
create counter  
2  
increment n  
3
```

生成器的使用

- 使用生成器

- 利用生成器，产生斐波那契数列

```
def fibonacci(n): # 生成器函数 - 斐波那契数列 (这个数列从第3项开始,
每一项都等于前两项之和)
```

```
    a, b, counter = 0, 1, 0
```

```
    while True:
```

```
        if (counter > n):
```

```
            return
```

```
        yield a
```

```
        a, b = b, a + b
```

```
        counter += 1
```

```
f = fibonacci(10) # f 是一个迭代器，由生成器返回生成
```

```
while True:
```

```
    try: #检测异常
```

```
        print (next(f), end=" ")
```

```
    except StopIteration: #next完成后触发异常
```

```
    # 遇到StopIteration就退出循环
```

```
        break
```

0 1 1 2 3 5 8 13 21 34 55

第五章 结束

1、编写一个函数，使得列表中每个单词最后一个字母变成大写

`['Xml', 'history', 'bart']`

2、编写一个函数，同时返回最大值和最小值

难题解析

```
my_list = [ lambda : n for n in range(5) ]
```

```
for x in my_list:
```

```
    print( x())
```

```
# output
```

```
4
```

```
4
```

```
4
```

```
4
```

```
4
```

```
def func(x):
```

```
    return x
```

```
#等价于
```

```
func = lambda x:x
```

```
my_list = [ lambda : n for n in range(5) ]
```

```
    for x in my_list:
```

```
        print(x())
```

```
# 等价于
```

```
my_list = []
```

```
for n in range(5):
```

```
    my_list.append(lambda : n)
```

```
    for x in my_list:
```

```
        print (x())
```

my_list.append(lambda : n) 中定义的 lambda, 其中的 n 是引用 for n in range(5) 这一句中的, 这只是 lambda 的定义阶段, lambda 并没有执行, 等这两句执行完之后, n 已经等于 4 了, 也就是说, 定义的这5个lambda 全部变成了 lambda : 4, 等到执行的时候自然输出就成了4, 4, 4, 4, 4

x()后面的括号表示调用lambda表达式!!!

```
my_list = []
for n in range(5):
    my_list.append(lambda : n)
for x in my_list:
    print (x)
```

```
<function <lambda> at 0x000001F9031F0040>
<function <lambda> at 0x000001F9031F00D0>
<function <lambda> at 0x000001F9031F0160>
<function <lambda> at 0x000001F9031F01F0>
<function <lambda> at 0x000001F9031F0280>
```

X 不加括号显示对应对象

```
print(my_list)
```

```
[<function <lambda> at 0x000001F9031F0040>, <function <lambda> at 0x000001F9031F00D0>, <function <lambda> at 0x000001F9031F0160>, <function <lambda> at 0x000001F9031F01F0>, <function <lambda> at 0x000001F9031F08B0>]
```

X() 表示调用lambda表达式

```
my_list = []
for n in range(5):
    my_list.append(lambda : n)
for x in my_list:
    print (x())
```

```
4
4
4
4
4
```

修改为

```
my_list = []  
for i in range(5):  
    my_list.append(lambda n = i: n)  
for x in my_list:  
    print( x())
```

结果为



0
1
2
3
4

```
my_list = []  
for i in range(5):  
    my_list.append(lambda n = i: n)  
for x in my_list:  
    print( x())
```

0
1
2
3
4

参考: <https://www.cnblogs.com/liuq/p/6073855.html>


```
li = [lambda :x for x in range(10)]
```

```
li = [lambda :x for x in range(10)]
```

```
print(li)
```

 得到一个 lambda 的list, 看一看 li :

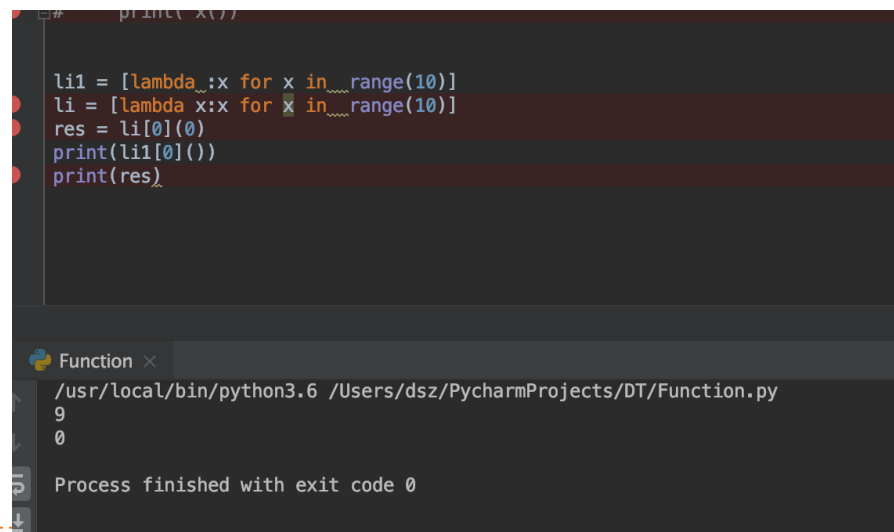
```
[<function <listcomp>.<lambda> at 0x000001F90313E4C0>, <function <listcomp>.<lambda> at 0x000001F90313E280>, <function <listcomp>.<lambda> at 0x000001F9031F93A0>, <function <listcomp>.<lambda> at 0x000001F9031F9310>, <function <listcomp>.<lambda> at 0x000001F9031F9430>, <function <listcomp>.<lambda> at 0x000001F9031F94C0>, <function <listcomp>.<lambda> at 0x000001F9031F9550>, <function <listcomp>.<lambda> at 0x000001F9031F95E0>, <function <listcomp>.<lambda> at 0x000001F9031F9670>, <function <listcomp>.<lambda> at 0x000001F9031F9700>]
```

打印 li[0]() 会得到什么, 注意这种调用方法, 因为是数组, 先拿第一个元素, li[0], 后面的括号表示调用lambda表达式。打印的结果为 0 吗? NO! 是 9. lambda表达式不会形成对函数体内变量的记忆, 只记录最后一个状态。

那么如果lambda的入参中带有x, 会得到我们想要的结果0 吗?

```
li = [lambda x :x for x in range(10)]
res = li[0](0)
print(res)
```

0

A screenshot of a Python IDE window. The top pane shows the following code:

```
li1 = [lambda :x for x in range(10)]
li = [lambda x:x for x in range(10)]
res = li[0](0)
print(li1[0]())
print(res)
```

The bottom pane shows the output of the code execution:

```
Function x
/usr/local/bin/python3.6 /Users/dsz/PycharmProjects/DT/Function.py
9
0
Process finished with exit code 0
```