**目录：**

**0、代码复用技术/模块化思维：函数的诞生之由**

**1、初识函数式编程**

**2、函数嵌套**

**3、可调用对象**

**4、修饰器**

**5、函数参量**

**6、变量作用域**

**7、lambda表达式（微缩型函数）**

**8、生成器函数**

**0、代码复用技术：函数的诞生之由**

1>在软件开发的过程中，常有许多操作是重复出现的，其仅仅是处理的数据不同，代码总是保持高度的相似性。

2>重复编写高相似度的代码，会增加代码量，也消减了代码可的维护性与可读性。

1. **初识函数式编程：**
2. **代码复用技术之一**

（可以保证代码的一致性，只需要修改该函数代码，则所有调用位置都能得到体现）

#设计类是另一种代码复用技术

1. **模块化编程理念**

（函数式编程是也一种封装技术，它暗含有面向对象式编程及模块化编程的理念）

|**面向对象式编程(树状结构思维)**是基于工程化编程的应用场景应运而生的，其强调模块 （类）及程序之间的关系。**模块**指的是：完成某个特定功能时所需的程序集合，它们在 功能上具有类似性并且程序集合间的内聚性强。|

|**面向过程的编程**：思考的是完成一个任务的具体步骤，然后通过函数来实现每一步。 而面向对象的编程思维是把一个任务按层次来细分下来（类似树状结构），且其第二层 就是类，而类又是函数集合的封装。可见面向过程是面向对象编程思维的前驱者。|

|**封装技术**是指：编写一个特定功能的函数程序后，在后面调用时，给其他的程序提供 参数接口及特定的输出端口。|

1. **函数式编程时的注意事项：**

（追求高内聚性及低隐式耦合的编程思维）

（少用函数的嵌套，因为在使用时内部函数会多次再定义，增加了时空复杂度）

**4>定义函数的一般格式（函数头+函数体）**

函数头：def f([参数列表：形参]):

函数体： ’’’注释’’’

功能语句

print（...）

return 表达式

#可以使用函数：help(f)来查看该函数内部写的注释。

#**print与return的区别：**

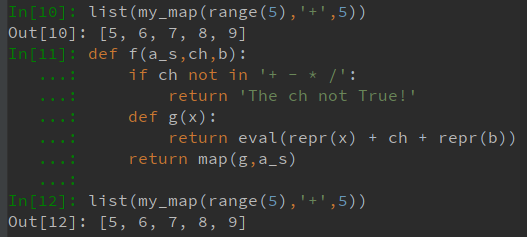
print：其作用是输出函数的值到控制台中（打印到你能看得到的地方）。具有脱壳（去掉组织该数据的数据结构）外显的功能。

return：1>退出函数（具有短路的性质）2>选择性的向调用方返回一个表达式。

**5>调用函数的一般格式**

函数名（标识符）（实参）--> f（[实参]）

**2、函数嵌套**



1. 注意函数间和语句间的缩进关系
2. 注意嵌套函数间的参量传递
3. eval()：计算str并返回一个值
4. repr()：返回对象为能被解释器读取的str形式
5. g(x)不外显

**3、可调用对象**

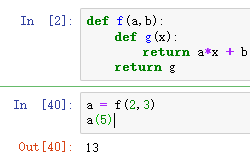
**1>除了用户定义的函数，调用运算符（即 ()）还可以应用到其他对象上。**

#如果想判断对象，能否调用，可以使用内置的 callable() 函数。

**2>常见的可调用对象**：

**· 用户定义的函数**

【用户自定义函数、lambda表达式（伪函数）】



a = f(2,3):使用f(a,b)后，向a调用g(x)，形成了一个将g 储存在a 内的闭包。当a(5)时，会向该闭包中传达实参5给g(x)。

**· 内置函数**

**· 内置方法**

|**內建方法和内建函数不同之处**：

內建方法的\_\_self\_\_属性指向一个python对象

内建函数的\_\_self\_\_指向None

**· 自定义方法**

【用户自定义方法是在类的定义体中定义的函数或者说是被定义为类的一部分的函数】

**· 类**  
【调用类时会运行类的 \_\_new\_\_ 方法创建一个实 例，然后运行 \_\_init\_\_ 方法，初始化实例，最后把实例返回给调用方。因为 Python 没有 new 运算符，所以调用类相当于调用函数。（通常，调用类会创建那个类的实例，不过覆盖 \_\_new\_\_ 方法的话，也可能出现其他行为。）】

**· 类的实例**  
【如果类定义了 \_\_call\_\_方法，那么它的实例可以作为函数调用】

**· 生成器函数**  
【使用 yield 关键字的函数或方法，调用生成器函数返回的是生成器对象】|

**4、修饰器**

【装饰器是一个很著名的设计模式，经常被用于有切面需求的场景，较为经典的有插入日志、性能测试、事务处理等应用。装饰器是解决这类问题的绝佳设计，有了修饰器，我们就可以抽离出大量函数中与函数功能本身无关的雷同代码并继续重用。概括的讲，**修饰器的作用就是为已经存在的对象添加额外的功能。】**

**【** 1>修饰器本质上也是函数

1. 这个函数接受其他函数为参数
2. 并且基于调用的修饰器返回值来修饰函数

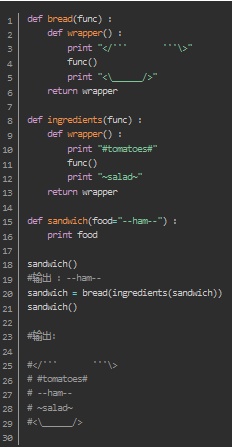
4>最终得到一个新函数**】**

实例A：



* 这里的@makebold和@makeitalic似乎给Hello加上了一层包装，这就是修饰器的体现。
* @f具有寄存修饰器函数闭包与引用的作用
* Line1和line6定义了形如[f(g)]的两个修饰器

实例B：



【顺序在修饰器中很重要！】

1>Line20相当于:

@break

@ingredients

@sandwich

**5、函数参量**

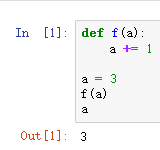
**【初识函数参量】**

1. **定义函数时不需要声明参数类型**

#参数列表中的参数由逗号分隔，存在形如（1，2，3）这样的三目参数。

1. **在函数内部直接修改形参的值不会影响实参**

#函数本质上是种封装技术，内部的局部变量的值的变化，在无特殊声明的情况下，优先级低于外部的全局变量，因而无从干涉到外部。



#但是当是可变序列型的数据结构作为函数参数的时候，如果函数内部通过可变序列特有的方式(下标、增删改的方法、增删改的函数)修改参数中的元素时，同样的作用会出现在实参上。**（原理：基于值的自动管理模式）**

**【位置参数】**

在**调用函数**时，实参和形参的**顺序**必须保持**一致**，并且他们的**数量**必须**相同**。

**【默认值参数】**

1. **定义函数时可以为形参定义默认值**

#任何一个默认值参数右边都不能再出现没有默认值的普通位置参数

#可以通过 函数名.\_ \_ defaults\_ \_ /函数名.func\_defaults 这两种方式，来查看函数的所有的默认值参数。

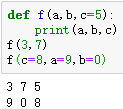
1. **多次调用函数并且不为默认值函数传递值时，默认值参数只在定义时进行一次解释和初始化。**

#对于列表、字典这样可变的数据结构的默认值参数，可能会导致严重的逻辑错误，不建议这样使用。

1. **尽管我们要避免用列表这样的可变数据结构的默认值形参，但是由于他们具有记忆性。我们利用这样的特点可以，连续多次调用该函数而不给该参数传值时，再次调用将保留上一次调用的结果。**
2. **函数的默认值参数是在函数定义时确定值的，只进行一次初始化。**

**【关键参数】**

**调用函数时，可以按形参名来对号给值。**



优点：避免用户需要牢记参数位置和顺序的麻烦，使得函数的调用和参数传递更加灵活方便。

**【可变长度参数】**

1. **概念理解**

他首先是在定义函数时，在形参列表中使用的参数样式方法。我们可以形象的理解到，他更像是给参数一个可变的外挂空间，它可以用来接收任意长度的溢出实参值。

1. **样式与区别**

在定义函数时，存在参量的样式：**1：\*parameter 2：\*\*parameter**

>>>def f(\*x):

print(p)

>>>f(1,2,3)

>>>(1,2,3)

#1：用来接收任意多个实参，并将其放在一个**元组**中。

#2：用来接收任意多个实参，并将其放在一个**字典**中。

**【传递参数时的序列解包】**

**1>概念理解**

它是在调用函数时，在实参列表中使用的参数样式方法。我们可以形象的理解到，把我们赋给形参的含有多维性质的数据集解包，并按照一定次序分发给我们的形参列表中的形参。

**2>样式与区别**

在调用函数时，存在参量的样式：**1：\*x 2：\*\*x**

>>>P = {‘a’:1, ‘b’: 2, ‘c’:3}

>>>def f(a,b,c=1):

print(a,b,c)

>>>f(\*\*P)

1 2 3

#1：可以解列表、元组、集合、对字典的键、对字典的值，进行解包

#2：可以对字典解包

（对字典的值解包：\*dict.values(x) == \*\*dict(x))

（对字典的全体解包要使用方法2|对字典的键解包要使用方法1/类似于集合的用法）

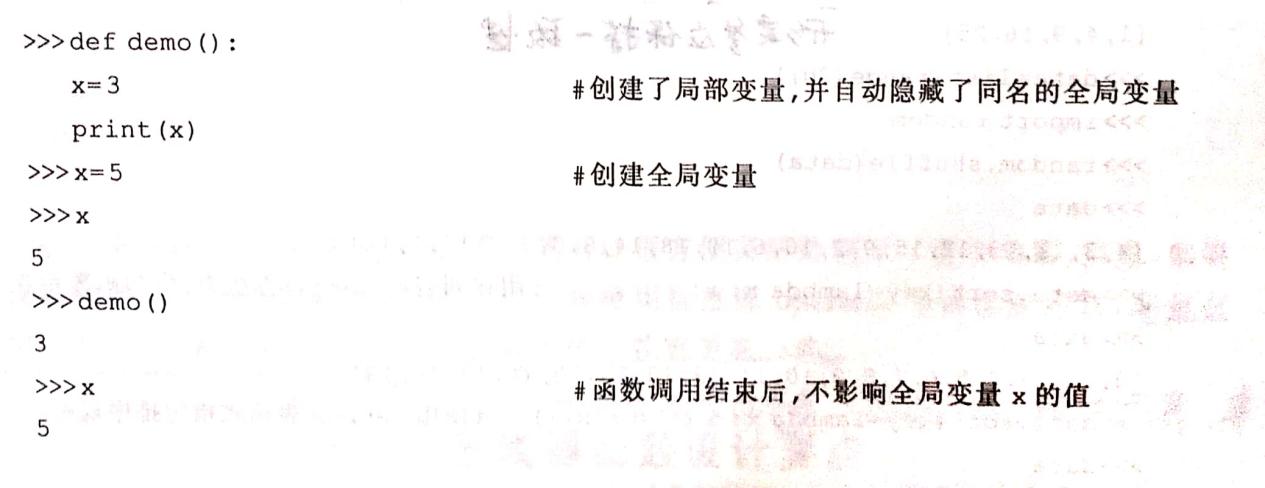
1. **变量作用域**

**【定义】**

变量起作用的代码范围称为变量的作用域。

**【使用习惯】**

1. **不同作用域内的同名变量间不受影响。**

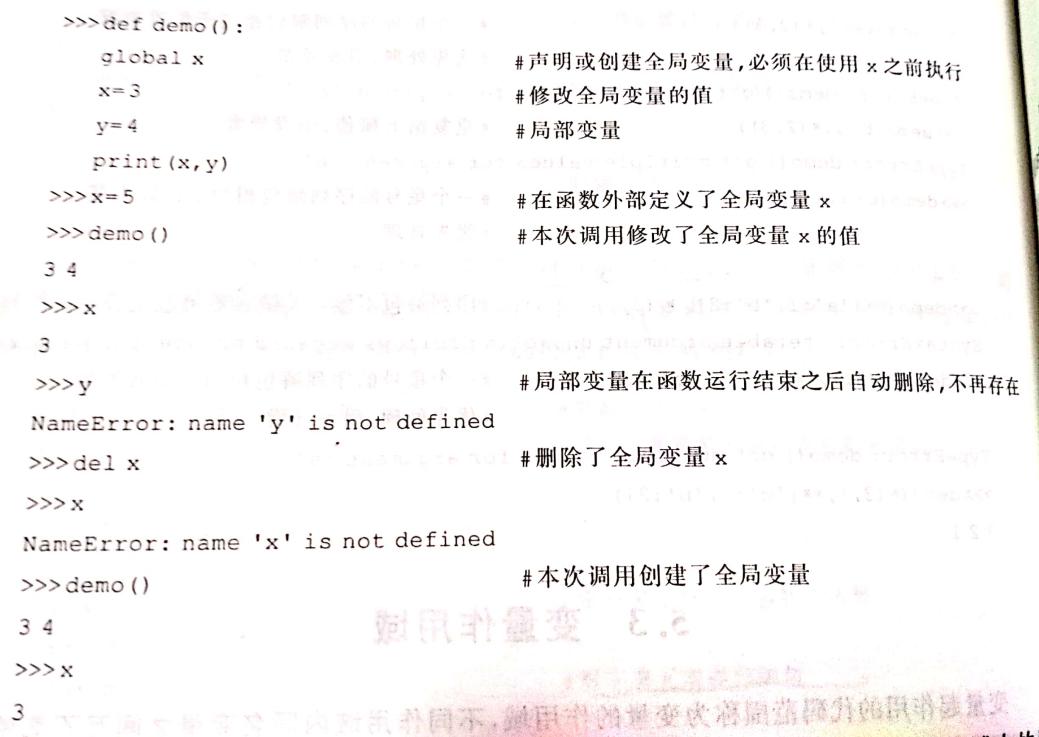


1. **在函数内定义的局部变量只在该函数内可见**。

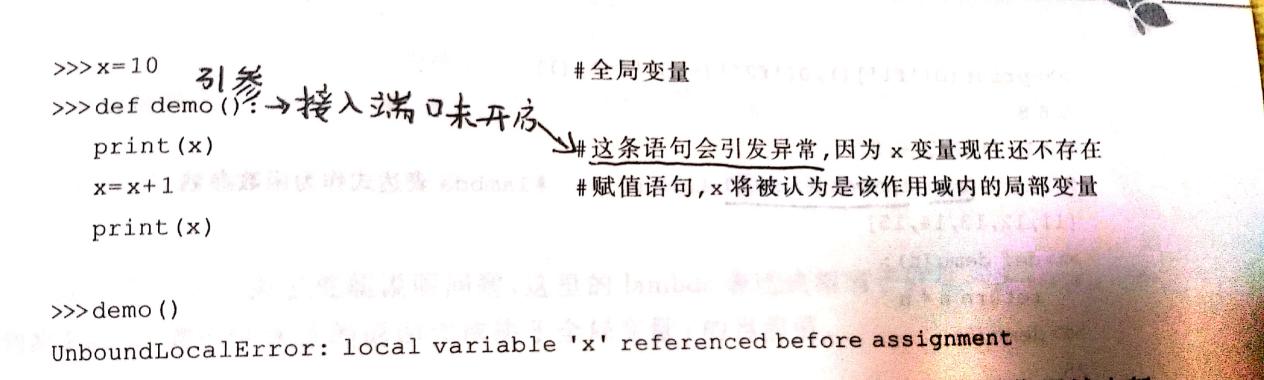
#他在其所在的领域内，优先级高于同名全局变量。

#在函数内用 global 定义的局部变量为全局变量，可干涉到函数外的运算。（这样可以在函数内修改变量的值并映射到全局）

#在函数内用 global 定义的局部变量为全局变量，如果函数外没有定义该全局变量，在调用这个函数的时候，会创建新的全局变量



#如果在某个作用域内有为变量赋值的操作，那么该变量将被认为是该作用域内的局部变量



1. **lambda表达式（微缩型函数）**

**【定义】**

其通常用来声明匿名函数，该函数是没有函数名字的临时使用的小函数，常用在需要类似于函数功能但又不想定义函数的场合。

#map(),,filter(),,list.sort(),,sorted()... ...

**#lambda表达式只可以包含一个表达式**

**【表达样式】**

lambda x : x+10

#“:”左边的参数为微缩函数的形参；右边的表达式相当于return运算式**。**

**【使用习惯】**

1. **可以给lambda表达式起一个名字**

#f = lambda x:x+2

1. **支持默认参数**

#f = lambda x,y=2,z=3:x+y+z

1. **调用时可使用关键参数**

#print(f(2,z=5,y=5))

1. **运算及参量可嵌入lambda表达式中**

#D = {‘f1’:(lambda:2+3), ‘f2’:(lambda:2\*\*3)}

Print( D[‘f1’]( ), D[‘f1’]( ) ) >>>5 8

1. **在lambda表达式中的右目参数处可以调用函数（套娃）**

#必须保证lambda表达式中的左目参数与套娃的函数的形参是一致的。

1. **当lambda表达式中的左目参数为空时，其右目表达式中的参数随其全局变量**。

**【忠告】**

**尽量避免使用lambda表达式，operator中的函数执行效率更高一些。**

1. **生成器函数**

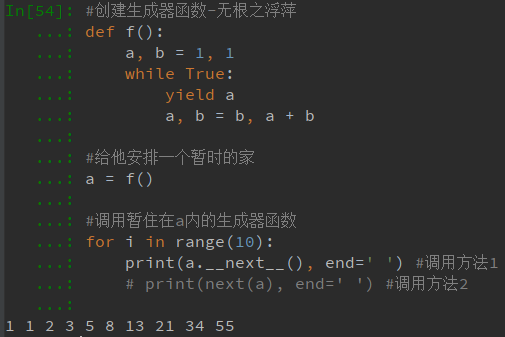
**【创建生成器函数】**

包含yield语句的函数可以用来创建生成器对象，这样的函数叫做生成器函数。

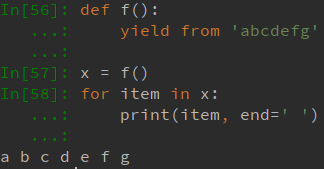
#yield语句与return语句的作用相似，都是用来从函数中返回值。但是，return语句具有执行即截止函数的功能；每次执行到yield语句并返回一个值之后会暂停或挂起后面代码的执行。

**【调用生成器函数】**

可通过生成器对象的.\_\_next\_\_() | next() | for循环遍历生成器对象元素等方式显示”索要“数据时恢复执行该函数。



#调用方法3：



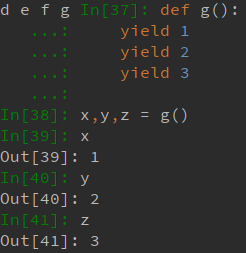
**【使用习惯】**

1. **生成器具有惰性求值及短路的特点，适合大数据处理。**

#在一次完整的调用中，前次运算会清楚掉遍历过的值，清除的值不会出现在后次运算中。

#不同次的调用间，互不受影响。

1. **生成器对象支持序列解包**



**【实战应用】**

