```
#encoding=utf-8

########## 函数和函数式编程  ##################


#********* Part 1: 函数  ******************
#函数是对程序逻辑进行结构化或过程化的一种编程方法。能将整块代码巧妙地隔离成易于管理
#的小块，把重复代码放到函数中而不是进行大量的拷贝--这样既能节省空间，也有助于保持一致性，
#因为你只需改变单个的拷贝而无须去寻找再修改大量复制代码的拷贝。

## 1.1 函数 vs 过程
#两者都是可以被调用的实体，但是传统意义上的函数或者“黑盒”，
#可能不带任何输入参数，经过一定的处理，最后向调用者传回返回值。其中一些函数则是布尔类型的，
#返回一个“是“或者“否“的回答，更确切地说，一个非零或者零值。而过程是简单，特殊，
#没有返回值的函数。其实 python 的过程就是函数，因为解释器会隐式地返回默认值 None

## 1.2 返回值与函数类型
#函数会向调用者返回一个值，而实际编程中大偏函数更接近过程，不显示地返回任何东西。把
#过程看待成函数的语言通常对于“什么都不返回”的函数设定了特殊的类型或者值的名字。这些函
#数在 c 中默认为 void 的返回类型，意思是没有值返回。 在 python 中， 对应的返回对象类型是 none。

#def hello():
#       print('hello world')
#
#res = hello()
#print(res)
#type(res)
#-->
#hello world
#None
```

#python 里的函数可以返回一个值或者对象。只是在返回一个容器对象的时候有点不同

```
def foo():
    return ['xyz', 1000000, -98.6]
def bar():
    return 'abc', [42, 'python'], "Guido"
```

#foo()函数返回一个列表，bar()函数返回一个元组。由于元组语法上不需要一定带上圆括号，所以让人真的以为可以返回多个对象。
#bar()的定义看起来会是这样：
#def bar():
return ('abc', [4-2j, 'python'], "Guido")
#从返回值的角度来考虑， 可以通过很多方式来存储元组。接下来的 3 种保存返回值的方式是等价的

```
#aTuple = bar()
#x, y, z = bar()
#(a, b, c) = bar()
#
#print(aTuple)
#print(x, y, z)
#print(a, b, c)
#-->
#('abc', [42, 'python'], 'Guido')
#abc [42, 'python'] Guido
#abc [42, 'python'] Guido
```

#在对 x,y,z 和 a,b,c 的赋值中，根据值返回的顺序，每个变量会接收到与之对应的返回值。而 aTuple 直接获得函数隐式返回的整个元组

#简而言之，当没有显式地返回元素或者如果返回 None 时， python 会返回一个 None.那么调用
#者接收的就是 python 返回的那个对象,且对象的类型仍然相同。如果函数返回多个对象，python 把
#他们聚集起来并以一个元组返回

```
##**********  Part 2: 调用函数 ***************
## 2.1 函数操作符
#用一对圆括号调用函数,任何输入的参数都必须放置在括号中。
#作为函数声明的一部分，括号也会用来定义那些参数。在 python 中，函数的操作符同样用于类的实例化。

## 2.2 关键字参数
```

#关键字参数的概念仅仅针对函数的调用。这种理念是让调用者通过函数调用中的参数名字来区
#分参数。这样规范允许参数缺失或者不按顺序,因为解释器能通过给出的关键字来匹配参数的值。

```
#def fun(x):
#       print('the input arguements is:',x)
#
#fun(5)
#fun(x=5)
#-->
#the input arguements is: 5
#the input arguements is: 5
#这里在调用时，指定和不指定 x，他们是一样的

#如果有多个参数时，不指定参数，就按默认顺序，如果不按默认顺序就需要特别指定参数

#def fun(name,sex):
#       print('Name:',name+',sex:',sex)
#
#fun('Dave','man')
#fun(sex='man',name='Dave')
#-->
#Name: Dave,sex: man
#Name: Dave,sex: man

## 2.3 默认参数
#默认参数就是声明了默认值的参数。因为给参数赋予了默认值，所以， 在函数调用时，不向该参数传入值也是允许的。

#def fun(name='dave',sex='man'):
#       print('Name:',name+',sex:',sex)
#
#fun()
#fun('DMM','boy')
#-->
#Name: dave,sex: man
#Name: DMM,sex: boy

## 2.4 参数组
#Python 允许程序员执行一个没有显式定义参数的函数，相应的方法是通过一个把元组（非关键字参数）或
#字典（关键字参数）作为参数组传递给函数。基本上，你可以将所有参数放进一个元组或者字典中，
```

```
#仅仅用这些装有参数的容器来调用一个函数，而不必显式地将它们放在函数调用中：
#       func(*tuple_grp_nonkw_args, **dict_grp_kw_args)

#其中的 tuple_grp_nonkw_args 是以元组形式体现的非关键字参数组, dict_grp_kw_args 是装有关键字参数的字典。
#
#实际上，也可以给出形参. 这些参数包括标准的位置参数和关键字参数，所以在 python 中允许的函数调用的完整语法为：
#       func(positional_args, keyword_args, *tuple_grp_nonkw_args, **dict_grp_kw_args)
#该语法中的所有的参数都是可选的---从参数传递到函数的过程来看，在单独的函数调用时，每个参数都是独立的。
#这可以有效地取代 apply()内建函数。(在 Python1.6 版本之前，这样的参数对象只能通过 apply()函数来调用)。

#def fun(*nums):
#       print(nums)
#       print(len(nums))
#       for item in nums:
#               print(item)
#
#a=['dave','is','dba']
#fun(a)
#-->
#(['dave', 'is', 'dba'],) -->注意这里我们的列表变成了元组
#1                                       -->该元组的长度为 1
#['dave', 'is', 'dba']       -->第一个值是我们的列表 a


#*************  Part 3：  创建函数  ********************
## 3.1 def 语句

#函数是用 def 语句来创建的，语法如下：
#def function_name(arguments):
#       "function_documentation_string"
#       function_body_suite
#
#标题行由 def 关键字，函数的名字，以及参数的集合（如果有的话）组成。def 子句的剩余部分包括了一个虽然可选但是强烈推荐的文档字串，和必需的函数体。

#def hello(who):
#       return "Hello " + str(who)
#
#print(hello('Dave'))
#-->Hello Dave
```

3.2 声明与定义比较
#在某些编程语言里，函数声明和函数定义区分开的。一个函数声明包括提供对函数名，参数的
#名字（传统上还有参数的类型），但不必给出函数的任何代码，具体的代码通常属于函数定义的范畴。
#在声明和定义有区别的语言中，往往是因为函数的定义可能和其声明放在不同的文件中。
python
#将这两者视为一体，函数的子句由声明的标题行以及随后的定义体组成的。

3.3 前向引用
#Python 不允许在函数未声明之前,对其进行引用或者调用.
#
#示例:
#def foo():
#　　print('in foo()')
#　　bar()

#如果我们调用函数 foo(),肯定会失败,因为函数 bar()还没有声明。

3.4 函数属性

#可以获得每个 pyhon 模块，类，和函数中任意的名字空间。你可以在模块 foo 和 bar 里都有
#名为 x 的一个变量,，但是在将这两个模块导入你的程序后，仍然可以使用这两个变量。所以，即使
#在两个模块中使用了相同的变量名字，这也是安全的，因为句点属性标识对于两个模块意味了不同
#的命名空间，比如说，在这段代码中没有名字冲突：
#import foo, bar
#print(foo.x + bar.x)

#函数属性是 python 另外一个使用了句点属性标识并拥有名字空间的领域

#def bar():
#　　pass
#
#bar.__doc__ = 'Oops, forgot the doc str above'
#bar.version = 0.1
#
#print(bar.version)

3.5 内部/内嵌函数
#在函数体内创建另外一个函数（对象）是完全合法的。这种函数叫做内部/内嵌函数.
#def fun1(str):
#　　print('hello'+str)
#
#def fun2(str2):
#　　fun1(str2)
#
#fun2('Dave')
#-->helloDave

3.6 函数（与方法）装饰器
#装饰器实际就是函数。它接受函数对象。一般说来，当你包装一个函数的时候，你最终会调用它。
#最棒的是我们能在包装的环境下在合适的时机调用它。我们在执行函数之前，可以运行些预备代码，如 post-morrem 分析,
#也可以在执行代码之后做些清理工作。所以当你看见一个装饰器函数的时候，很可能在里面找到这样一些代
#码，它定义了某个函数并在定义内的某处嵌入了对目标函数的调用或者至少一些引用。从本质上看，
#这些特征引入了 java 开发者称呼之为 AOP（Aspect Oriented Programming，面向方面编程）的概念。

#你可以考虑在装饰器中置入通用功能的代码来降低程序复杂度。例如，可以用装饰器来：
#　　引入日志
#　　增加计时逻辑来检测性能
#　　给函数加入事务的能力
#
#对于用 python 创建企业级应用，支持装饰器的特性是非常重要的.

#装饰器的语法以@开头，接着是装饰器函数的名字和可选的参数。紧跟着装饰器声明的是被修饰
#的函数，和装饰函数的可选参数。装饰器看起来会是这样：
#@decorator(dec_opt_args)
#def func2Bdecorated(func_opt_args):pass
#
#
#装饰器可以如函数调用一样“堆叠“起来，如：
#@deco2
#@deco1
#def func(arg1, arg2, ...): pass

##有参数和无参数的装饰器

```
#没有参数的情况，一个装饰器如：
#@deco
#def foo(): pass
#
#等于
#foo = deco(foo)

#带参数的装饰器 decomaker()
#@decomaker(deco_args)
#def foo(): pass

#需要自己返回以函数作为参数的装饰器。换句话说，decomaker()用 deco_args 做了些事并返回
#函数对象，而该函数对象正是以 foo 作为其参数的装饰器。简单的说来：
#foo = decomaker(deco_args)(foo)
#
#这里有一个含有多个装饰器的例子，其中的一个装饰器带有一个参数
#@deco1(deco_arg)
#@deco2
#def func(): pass
#这等价于：
#func = deco1(deco_arg)(deco2(func))


#*************  Part 4：  传递函数 ********************
#当学习一门如 C 的语言时，函数指针的概念是一个高级话题，但是对于函数就像其他对象的
#python 来说就不是那么回事了.函数是可以被引用的（访问或者以其他变量作为其别名），也作为参
#数传入函数，以及作为列表和字典等等容器对象的元素
#函数有一个独一无二的特征使它同其他对象区分开来，那就是函数是可调用的。

#因为所有的对象都是通过引用来传递的，函数也不例外。当对一个变量赋值时，实际是将相同
#对象的引用赋值给这个变量。如果对象是函数的话，这个对象所有的别名都是可调用的。

#def fun():
#       print('Hello Dave!')
#
#fun1=fun()
#
#def fun2(f):
#       f()
#
```

```
#fun2(fun)
#-->
#Hello Dave!
#Hello Dave!

#示例 2：
#def convert(func, seq):
# 'conv. sequence of numbers to same type'
# return [func(eachNum) for eachNum in seq]
#
#myseq = (123, 45.67, -6.2e8)
#print(convert(int, myseq))
#print(convert(float, myseq))
#-->
#[123, 45, -620000000]
#[123.0, 45.67, -620000000.0]

#*************  Part 5：  形式参数 ******************
#python  函数的形参集合由在调用时要传入函数的所有参数组成，这参数与函数声明中的参数列
#表精确的配对。这些参数包括了所有必要参数(以正确的定位顺序来传入函数的)，关键字参数（以
#顺序或者不按顺序传入，但是带有参数列表中曾定义过的关键字），以及所有含有默认值，函数调用
#时不必要指定的参数。(声明函数时创建的)局部命名空间为各个参数值，创建了一个名字。一旦函
#数开始执行，即能访问这个名字。

## 5.1  位置参数
#位置参数必须以在被调用函数中定义的准确顺序来传递。另
#外，没有任何默认参数（见下一个部分）的话，传入函数（调用）的参数的精确的数目必须和声明的数字一致。

#def fun(str):
#       print(str)

#fun 函数有一个位置参数。那意味着任何对 fun()的调用必须有唯一的一个参数。否则你会频频看到 TypeError。
#作为一个普遍的规则，无论何时调用函数，都必须提供函数的所有位置参数。

#fun()
#-->TypeError: fun() takes exactly 1 argument (0 given)
#fun('dave')
#-->dave
```

```
#fun('david','dai')
#-->TypeError: fun() takes exactly 1 positional argument (2 given)

## 5.2 默认参数
#对于默认参数如果在函数调用时没有为参数提供值则使用预先定义的的默认值。这些定义在函数声明的标题行中给出.

#默认参数让程序的健壮性上升到极高的级别，因为它们补充了标准位置参数没有提供的一些灵
#活性。这种简洁极大的帮助了程序员。当少几个需要操心的参数时候，生活不再那么复杂。这在一
#个程序员刚接触到一个 API 接口时，没有足够的知识来给参数提供更对口的值时显得尤为有帮助。

#示例：
#def fun(cost,rate=0.8):
#       print('the cost is:',cost*rate)
#
#fun(100)
#fun(100,0.5)
#-->
#the cost is: 80.0
#the cost is: 50.0

#所有必需的参数都要在默认参数之前。简单说来就是因为它们是强制性的，但默认参数不是。
#从句法构成上看，对于解释器来说，如果允许混合模式，确定什么值来匹配什么参数是不
#可能的。如果没有按正确的顺序给出参数，就会产生一个语法错误。

#**********  Part 6： 可变长度的参数  ********************
#可能会有需要用函数处理可变数量参数的情况。这时可使用可变长度的参数列表。变长的参数
#在函数声明中不是显式命名的，因为参数的数目在运行时之前是未知的（甚至在运行的期间，每次
#函数调用的参数的数目也可能是不同的），这和常规参数（位置和默认）明显不同，常规参数都是在
#函数声明中命名的。由于函数调用提供了关键字以及非关键字两种参数类型，python 用两种方法来
#支持变长参数，
#在函数调用中使用*和**符号来指定元组和字典的元素作为非关键字以及关键字参数。

## 6.1
#       当函数被调用的时候，所有的形参（必须的和默认的）都将值赋给了在函数声明中相对应的局
```

```
#部变量。剩下的非关键字参数按顺序插入到一个元组中便于访问。
#
#      可变长的参数元组必须在位置和默认参数之后，带元组（或者非关键字可变长参数）的函数普
#遍的语法如下：
#def function_name([formal_args,] *vargs_tuple):
#      "function_documentation_string"
#      function_body_suite
#
#      星号操作符之后的形参将作为元组传递给函数,元组保存了所有传递给函数的"额外"的参数(匹
#配了所有位置和具名参数后剩余的)。如果没有给出额外的参数，元组为空。

#示例：
#def fun(arg1, arg2='default', *theRest):
#      'display regular args and non-keyword variable args'
#      print('formal arg 1:', arg1)
#      print('formal arg 2:', arg2)
#      for eachXtrArg in theRest:
#            print('another arg:', eachXtrArg)

#fun(1)
#-->
#formal arg 1: 1
#formal arg 2: default

#fun(1,2)
#-->
#formal arg 1: 1
#formal arg 2: 2

#fun(1,2,'dave','dai')
#-->
#formal arg 1: 1
#formal arg 2: 2
#another arg: dave
#another arg: dai

#l1=['dave','dai','dba']
#fun(1,*l1)
#-->
#formal arg 1: 1
#formal arg 2: dave
#another arg: dai
```

#another arg: dba
#
#注意这里我们并没有指定第二个参数，但是这个参数是必须的。 在这个示例中，我们列表里的第一个参数被占用了。

6.2 关键字变量参数（Dictionary）
#　　在我们有不定数目的或者额外集合的关键字的情况中， 参数被放入一个字典中，字典中键为参
#数名，值为相应的参数值。为什么一定要是字典呢?因为为每个参数-参数的名字和参数值--都是成
#对给出---用字典来保存这些参数自然就最适合不过了。
#
#这给出使用了变量参数字典来应对额外关键字参数的函数定义的语法：
#def function_name([formal_args,][*vargst,] **vargsd):
#　　function_documentation_string function_body_suite
#为了区分关键字参数和非关键字非正式参数，使用了双星号（**）。 **是被重载了的以便不与
#幂运算发生混淆。关键字变量参数应该为函数定义的最后一个参数，带**。

#示例：
#def dictVarArgs(arg1, arg2='defaultB', **theRest):
#　　'display 2 regular args and keyword variable args'
#　　print('formal arg1:', arg1)
#　　print('formal arg2:', arg2)
#　　for eachXtrArg in theRest.keys():
#　　　　print('Xtra arg %s: %s' % (eachXtrArg, str(theRest[eachXtrArg])))

#dictVarArgs(1, 2, c='dave')
#-->
#formal arg1: 1
#formal arg2: 2
#Xtra arg c: dave

#dictVarArgs('one', d=10, e='zoo', men=('freud', 'gaudi'))
#-->
#formal arg1: one
#formal arg2: defaultB
#Xtra arg men: ('freud', 'gaudi')
#Xtra arg e: zoo
#Xtra arg d: 10

#关键字和非关键字可变长参数都有可能用在同一个函数中，只要关键字字典是最后一个参数并

#且非关键字元组先于它之前出现

#********** Part 7： 函数式编程 **********
#Python 不是也不大可能会成为一种函数式编程语言，但是它支持许多有价值的函数式编程语言
#构建。也有些表现得像函数式编程机制但是从传统上也不能被认为是函数式编程语言的构建。Python
#提供的以 4 种内建函数和 lambda 表达式的形式出现

7.1 匿名函数与 lambda
#python 允许用 lambda 关键字创造匿名函数。匿名是因为不需要以标准的方式来声明，比如说，
#使用 def 语句。（除非赋值给一个局部变量，这样的对象也不会在任何的名字空间内创建名字.)然而，
#作为函数，它们也能有参数。一个完整的 lambda “语句”代表了一个表达式，这个表达式的定义体
#必须和声明放在同一行。我们现在来演示下匿名函数的语法:
lambda [arg1[, arg2, ... argN]]: expression
#参数是可选的，如果使用的参数话，参数通常也是表达式的一部分

#核心笔记：lambda 表达式返回可调用的函数对象。
#用合适的表达式调用一个 lambda 生成一个可以像其他函数一样使用的函数对象。它们可被传入
#给其他函数，用额外的引用别名化，作为容器对象以及作为可调用的对象被调用（如果需要的话，
#可以带参数）。当被调用的时候，如过给定相同的参数的话，这些对象会生成一个和相同表达式等价
#的结果。它们和那些返回等价表达式计算值相同的函数是不能区分的。

#def true(): return True
#
#等价的 lamdba 表达式:
#lambda :True
#
#可以把 lambda 表达式赋值给一个如列表和元组的数据结构，其中，基于一些输入标准，我们可以选择哪些函数可以执行，以及参数应该是什么
#虽然看起来 lambdda 是一个函数的单行版本，但是它不等同于 c++
#的内联语句，这种语句的目的是由于性能的原因，在调用时绕过函数的栈分配。lambda 表达式运作
#起来就像一个函数，当被调用时，创建一个框架对象。

#def add(x, y=2): return x + y
##等价与：

```
#a = lambda x, y=2: x + y
#print(a(1))
#print(add(1))
#-->
#3
#3
```

7.2 内建函数 apply()、filter()、map()、reduce()

#内建函数 描述
#apply(func[, nkw][, kw]) 用可选的参数来调用 func，nkw 为非关键字参数，kw 关键字参数；返回值是函数调用的返回值。
#filter(func, seq) 调用一个布尔函数 func 来迭代遍历每个 seq 中的元素；返回一个使 func 返回值为 ture 的元素的序列。
#map(func, seq1[,seq2...]) 将函数 func 作用于给定序列（s)的每个元素，并用一个列表来提供返回值；如果 func 为 None，
func 表现为一个身份函数，返回一个含有每个序列中元素集合的 n 个元组的列表。
#reduce(func, seq[, init]) 将二元函数作用于 seq 序列的元素，每次携带一对（先前的结果以及下一个序列元素），连续的将现有的结果
和下雨给值作用在获得的随后的结果上，最后减少我们的序列为一个单一的返回值；
如果初始值 init 给定，第一个比较会是 init 和第一个序列元素而不是序列的头两个元素。

7.3 偏函数应用
#currying 的概念将函数式编程的概念和默认参数以及可变参数结合在一起。一个带 n 个参数，
#curried 的函数固化第一个参数为固定参数，并返回另一个带 n-1 个参数函数对象，分别类似于 LISP
#的原始函数 car 和 cdr 的行为。Currying 能泛化成为偏函数应用（PFA）， 这种函数将任意数量（顺
#序）的参数的函数转化成另一个带剩余参数的函数对象。

```
#from operator import add, mul
#from functools import partial
#add1 = partial(add, 1) # add1(x) == add(1, x)
#mul100 = partial(mul, 100) # mul100(x) == mul(100, x)
#
#print(add1(10))
#print(add1(1))
#print(mul100(10))
```

#-->
#11
#2
#1000

#在这些程序中需要经常将二进制（作为字符串）转换成为整数。
#from functools import partial
#baseTwo = partial(int, base=2)
#baseTwo.__doc__ = 'Convert base 2 string to an int.'
#print(baseTwo('10010'))
#-->18

#************ Part 8： 变量作用域 ******************
#标识符的作用域是定义为其声明在程序里的可应用范围，变量可以是局部域或者全局域。

8.1 全局变量与局部变量
#定义在函数内的变量有局部作用域，在一个模块中最高级别的变量有全局作用域。

#“声明适用的程序的范围被称为了声明的作用域。在一个过程中，如果名字在过程的声明之内，
#它的出现即为过程的局部变量；否则的话，出现即为非局部的“

#全局变量的一个特征是除非被删除掉，否则它们的存活到脚本运行结束，且对于所有的函数，
#他们的值都是可以被访问的，然而局部变量，就像它们存放的栈，暂时地存在，仅仅只依赖于定义
#它们的函数现阶段是否处于活动。当一个函数调用出现时，其局部变量就进入声明它们的作用域。
#在那一刻，一个新的局部变量名为那个对象创建了，一旦函数完成，框架被释放，变量将会离开作用域。

#核心笔记：搜索标识符（aka 变量，名字，等等）
#当搜索一个标识符的时候，python 先从局部作用域开始搜索。如果在局部作用域内没有找到那
#个名字，那么就一定会在全局域找到这个变量否则就会被抛出 NameError 异常。

#当使用全局变量同名的局部变量的时候要小心。如果在赋予局部变量值之前，你在函数
#中（为了访问这个全局变量）使用了这样的名字，你将会得到一个异常（NAMEERROR 或者 Unbound-
#LocalError），而这取决于你使用的 python 版本。

8.2 globa 语句
#如果将全局变量的名字声明在一个函数体内的时候，全局变量的名字能被局部变量给覆盖掉。
#为了明确地引用一个已命名的全局变量，必须使用 global 语句。global 的语法如下：
#global var1[, var2[, ... varN]]]

```
#is_this_global = 'xyz'
#def foo():
#       global is_this_global
#       this_is_local = 'abc'
#       is_this_global = 'def'
#       print(this_is_local + is_this_global)
#
#foo()
#print(is_this_global)
#-->
#abcdef
#def

#局部变量隐藏了全局变量，示例：
#j, k = 1, 2
#def proc1():
#       j, k = 3, 4
#       print("j == %d and k == %d" % (j, k))
#       k = 5
#
#def proc2():
#       j = 6
#       proc1()
#       print("j == %d and k == %d" % (j, k))
#
#k = 7
#proc1()
#print("j == %d and k == %d" % (j, k))
#-->
#j == 3 and k == 4
#j == 1 and k == 7
#
#j = 8
#proc2()
#print("j == %d and k == %d" % (j, k))
#-->
#j == 3 and k == 4
#j == 6 and k == 7
```

```
#j == 8 and k == 7

#*************  Part 9：  递归  ********************
#如果函数包含了对其自身的调用，该函数就是递归的。

#def factorial(n):
#    if n == 0 or n == 1:
#        print('%s=' %n,end='')
#        return 1
#    else:
#        print('%s*' %n,end='')
#        return (n * factorial(n-1))
#
#print(factorial(5))
#-->5*4*3*2*1=120

#*************  Part 10：  生成器  ********************
#生成器是一个带 yield 语句的函数。一个函数或者子程序只返回一次，但一个生成器能暂停执行并返回一个中间的结果----
#那就是 yield 语句的功能， 返回一个值给调用者并暂停执行。当生成器的 next()方法被调用的时候，它会准确地从离开地方继续
#（当它返回[一个值以及]控制给调用者时）

## 10.1 简单的生成器特性
#与迭代器相似，生成器以另外的方式来运作：当到达一个真正的返回或者函数结束没有更多的
#值返回（当调用 next()），一个 StopIteration 异常就会抛出.
#def simpleGen():
#    yield 1
#    yield '2 --> punch!'
#
#myG = simpleGen()
#print(myG.__next__())
#print(myG.__next__())
##-->
##1
##2 --> punch!
#print(myG.__next__())
#-->StopIteration

#由于 python 的 for 循环有 next()调用和对 StopIteration 的处理，使用一个 for 循环而不是手
#动迭代穿过一个生成器（或者那种事物的迭代器）总是要简洁漂亮得多。
```

```
#def simpleGen():
#       yield 1
#       yield '2 --> punch!'
#
#for eachItem in simpleGen():
#       print(eachItem)
#-->
#1
#2 --> punch!
```

10.2 加强的生成器特性

#在 python2.5 中，一些加强特性加入到生成器中，所以除了 next()来获得下个生成的值，用户
#可以将值回送给生成器[send()]，在生成器中抛出异常，以及要求生成器退出[close()]
#由于双向的动作涉及到叫做 send()的代码来向生成器发送值（以及生成器返回的值发送回来），
#现在 yield 语句必须是一个表达式，因为当回到生成器中继续执行的时候，你或许正在接收一个进入的对象

#示例：

```
def counter(start_at=0):
    count = start_at
    while True:
        val = (yield count)
        if val is not None:
            count = val
        else:
            count += 1
```

#生成器带有一个初始化的值，对每次对生成器[next()]调用以 1 累加计数。用户已可以选择重
#置这个值，如果他们非常想要用新的值来调用 send()不是调用 next()。这个生成器是永远运行的，所以如果你想要终结它，调用 close()方法。

```
count = counter(5)
print(count.__next__())
print(count.__next__())
##-->
##5
##6
count.send(9)
print(count.__next__())
```

```
print(count.__next__())
#-->
#10
#11
count.close()
```