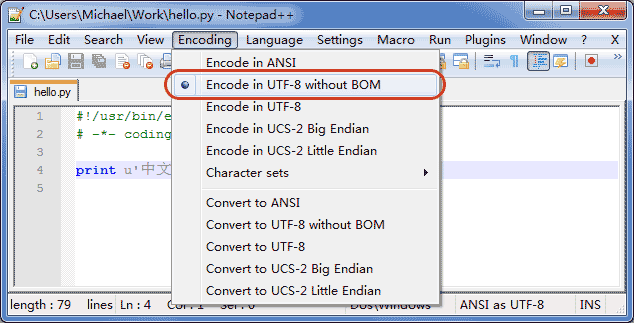
*#!/usr/bin/env python3*

*# -\*- coding: utf-8 -\*-*

第一行注释是为了告诉Linux/OS X系统，这是一个Python可执行程序，Windows系统会忽略这个注释；

第二行注释是为了告诉Python解释器，按照UTF-8编码读取源代码，否则，你在源代码中写的中文输出可能会有乱码。

申明了UTF-8编码并不意味着你的.py文件就是UTF-8编码的，必须并且要确保文本编辑器正在使用UTF-8 without BOM编码：



如果.py文件本身使用UTF-8编码，并且也申明了# -\*- coding: utf-8 -\*-，打开命令提示符测试就可以正常显示中文：

所以，我们把函数的参数改为可变参数：

**def** **calc**(\*numbers):

sum = 0

**for** n **in** numbers:

sum = sum + n \* n

**return** sum

定义可变参数和定义一个list或tuple参数相比，仅仅在参数前面加了一个\*号。在函数内部，参数numbers接收到的是一个tuple，因此，函数代码完全不变。但是，调用该函数时，可以传入任意个参数，包括0个参数：

Python允许你在list或tuple前面加一个\*号，把list或tuple的元素变成可变参数传进去：

>>> nums = [1, 2, 3]

>>> calc(\*nums)

14

位置参数  
  
在参数名之前使用一个星号，就是让函数接受任意多的位置参数。  
  
>>> def multiply(\*args):  
... total = 1  
... for arg in args:  
... total \*= arg  
... return total  
...  
>>> multiply(2, 3)  
6  
>>> multiply(2, 3, 4, 5, 6)  
720  
  
python把参数收集到一个元组中，作为变量args。显式声明的参数之外如果没有位置参数，这个参数就作为一个空元组。  
  
命名关键字参数python在参数名之前使用2个星号来支持任意多的关键字参数。  
  
>>> def accept(\*\*kwargs):  
... for keyword, value in kwargs.items():  
... print "%s => %r" % (keyword, value)  
...  
>>> accept(foo='bar', spam='eggs')  
foo => 'bar'  
spam => 'eggs'  
  
注意：kwargs是一个正常的python字典类型，包含参数名和值。如果没有更多的关键字参数，kwargs就是一个空字典。

**关键字参数（key-value）和命名关键字参数：**如果要限制关键字参数的名字，就可以用命名关键字参数

**命名关键字参数必须传入参数名**，这和位置参数不同。如果没有传入参数名，调用将报错：

>>> person('Jack', 24, 'Beijing', 'Engineer')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: person() takes 2 positional arguments but 4 were given

由于调用时缺少参数名city和job，Python解释器把这4个参数均视为位置参数，但person()函数仅接受2个位置参数。

命名关键字参数可以有缺省值，从而简化调用：

**def** **person**(name, age, \*, city='Beijing', job):

print(name, age, city, job)

由于命名关键字参数city具有默认值，调用时，可不传入city参数：

>>> person('Jack', 24, job='Engineer')

Jack 24 Beijing Engineer

使用命名关键字参数时，要特别注意，\*不是参数，而是特殊分隔符。如果缺少\*，Python解释器将无法识别位置参数和命名关键字参数：

**def** **person**(name, age, city, job):

*# 缺少 \*，city和job被视为位置参数*

**pass**

参数组合

在Python中定义函数，可以用必选参数、默认参数、可变参数、关键字参数和命名关键字参数，这5种参数都可以组合使用，除了可变参数无法和命名关键字参数混合。但是请注意，参数定义的顺序必须是：必选参数、默认参数、可变参数/命名关键字参数和关键字参数。

比如定义一个函数，包含上述若干种参数：

要注意定义可变参数和关键字参数的语法：

**\*args是可变参数，args接收的是一个tuple；**

**\*\*kw是关键字参数，kw接收的是一个dict。**

以及调用函数时如何传入可变参数和关键字参数的语法：

可变参数既可以直接传入：func(1, 2, 3)，又可以先**组装list或tuple**，再通过\*args传入：func(\*(1, 2, 3))；

关键字参数既可以直接传入：func(a=1, b=2)，又可以先组装dict，再通过\*\*kw传入：func(\*\*{'a': 1, 'b': 2})。

使用\*args和\*\*kw是Python的习惯写法，当然也可以用其他参数名，但最好使用习惯用法。

命名的关键字参数是为了限制调用者可以传入的参数名，同时可以提供默认值。

定义命名的关键字参数不要忘了写分隔符\*，否则定义的将是位置参数。

Example:L4=('12','34',[12,34]) 有\*和没\*的区别

>>> def p(l):

for i in l:

print (i)

>>> p(L4)

1

2

3

4

5

>>> def d(\*l):

for i in l:

print (i)

>>> d(L4)

('12', '34', [12, 34])

**def** **f1**(a, b, c=0, \*args, \*\*kw):

print('a =', a, 'b =', b, 'c =', c, 'args =', args, 'kw =', kw)

**def** **f2**(a, b, c=0, \*, d, \*\*kw):

print('a =', a, 'b =', b, 'c =', c, 'd =', d, 'kw =', kw)

**1. 栈和栈帧**  
栈（stack）相对整个系统而言，调用栈（Call stack）相对某个进程而言，栈帧（stack frame）则是相对某个函数而言，调用栈就是正在使用的栈空间，由多个嵌套调用函数所使用的栈帧组成。具体来说，Call stack就是指存放某个程序的正在运行的函数的信息的栈。Call stack 由 stack frames 组成，每个 stack frame 对应于一个未完成运行的函数。

Stack frame 组织方式的重要性和作用体现在两个方面：

第一，它使调用者和被调用者达成某种约定。这个约定定义了函数调用时函数参数的传递方式，函数返回值的返回方式，寄存器如何在调用者和被调用者之间进行共享；

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

斐波拉契数列用列表生成式写不出来，但是，用函数把它打印出来却很容易：

def fib(max):

n, a, b = 0, 0, 1

while n < max:

print(b)

a, b = b, a + b

n = n + 1

return 'done'

上面的函数可以输出斐波那契数列的前N个数：

>>> fib(6)

1

1

2

3

5

8

'done'

仔细观察，可以看出，fib函数实际上是定义了斐波拉契数列的推算规则，可以从第一个元素开始，推算出后续任意的元素，这种逻辑其实非常类似generator。

也就是说，上面的函数和generator仅一步之遥。要把fib函数变成generator，只需要把print(b)改为yield b就可以了：

def fib(max):

n, a, b = 0, 0, 1

while n < max:

yield b

a, b = b, a + b

n = n + 1

return 'done'

这就是定义generator的另一种方法。如果一个函数定义中包含yield关键字，那么这个函数就不再是一个普通函数，而是一个generator：

>>> f = fib(6)

>>> f

<generator object fib at 0x104feaaa0>

这里，最难理解的就是generator和函数的执行流程不一样。函数是顺序执行，遇到return语句或者最后一行函数语句就返回。而变成generator的函数，在每次调用next()的时候执行，遇到yield语句返回，再次执行时从上次返回的yield语句处继续执行。

举个简单的例子，定义一个generator，依次返回数字1，3，5：

def odd():

print('step 1')

yield 1

print('step 2')

yield(3)

print('step 3')

yield(5)

调用该generator时，首先要生成一个generator对象，然后用next()函数不断获得下一个返回值：

>>> o = odd()

>>> next(o)

step 1

1

>>> next(o)

step 2

3

>>> next(o)

step 3

5

>>> next(o)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

StopIteration

可以看到，odd不是普通函数，而是generator，在执行过程中，遇到yield就中断，下次又继续执行。执行3次yield后，已经没有yield可以执行了，所以，第4次调用next(o)就报错。

回到fib的例子，我们在循环过程中不断调用yield，就会不断中断。当然要给循环设置一个条件来退出循环，不然就会产生一个无限数列出来。

同样的，把函数改成generator后，我们基本上从来不会用next()来获取下一个返回值，而是直接使用for循环来迭代

>>> ord('A') ASCII value

65

#导入mode，**import与from…import**的不同之处在于，简单说：  
# 如果你想要直接输入argv变量到你的程序中而每次使用它时又不想打sys，  
# 则可使用：from sys import argv  
# 一般说来，应该避免使用from..import而使用import语句，  
# 因为这样可以使你的程序更加易读，也可以避免名称的冲突  
###########################

声明：s为字符串，rm为要删除的字符序列

s.strip(rm) 删除s字符串中开头、结尾处，位于 rm删除序列的字符

s.lstrip(rm) 删除s字符串中开头处，位于 rm删除序列的字符

s.rstrip(rm) 删除s字符串中结尾处，位于 rm删除序列的字符

>>> a = '123abc'  
>>> a.strip('21')  
'3abc' 结果是一样的  
>>> a.strip('12')  
'3abc'

**Python swapcase()** 方法用于对字符串的大小写字母进行转换。

***lambda***函数也叫匿名函数，即，函数没有具体的名称。先来看一个最简单例子：

def f(x):  
return x\*\*2  
print f(4)

Python中使用lambda的话，写成这样

g = lambda x : x\*\*2  
print g(4)