## Python1

网上链接： <http://www.runoob.com/python/python-install.html>

### 1节 简介

特点：优雅，简单，明确

适合领域：开发web网站和各种网络服务；系统工具和脚本；吧其他语言开发的模块包装起来方便使用。

不适合：贴近硬件的代码；移动开发不适合；游戏可发不适合

对比：

|  |  |  |  |
| --- | --- | --- | --- |
|  | 类型 | 运行速度 | 代码量 |
| C | 编译为机器码 | 非常快 | 非常多 |
| Java | 编译为字节码 | 快 | 多 |
| Python | 解释执行 | 慢 | 少 |

### 2节 版本

跨平台：2.7版本， 3.3版本

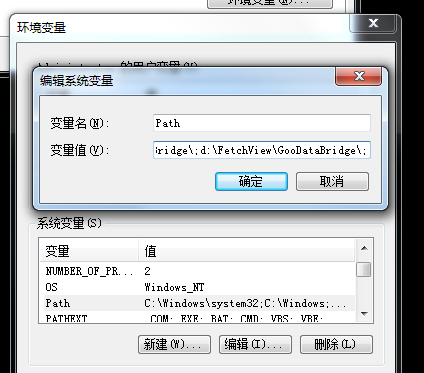
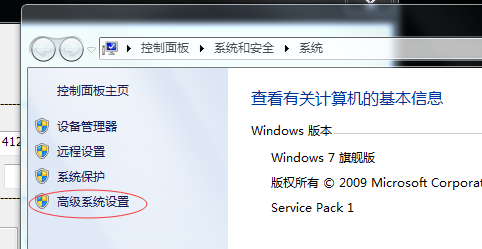
（2.7版本为基础）

<https://www.python.org/> 下载安装文件

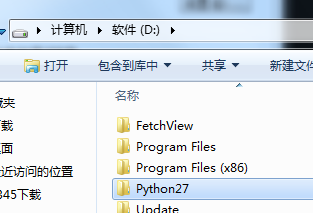
下载后，安装Python安装包。

Cmd命令行不能直接打开Python，需要配置环境变量：

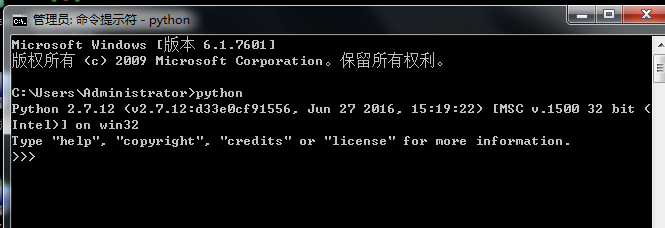
1.找到计算机属性-高级-环境变量

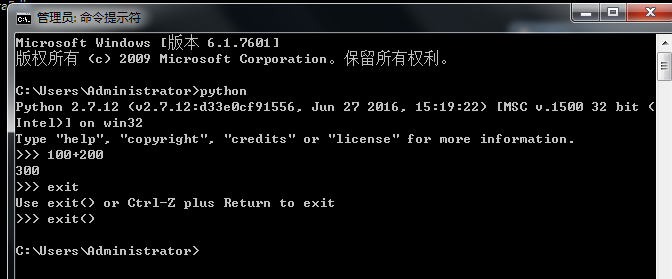


2.找到path后，在原有数据后面添加Python安装路径： D:\Python27 保存。



3.再次打开命令行，输入Python





### 3节

安装notepad++，设置首选项中新建文档的编码格式为utf-8（无BOM）格式，语言选择为Python语言。

编写代码后，保存为.py格式。可以用python直接运行 .py文件，输出结果。



在print语句中加逗号“，”分割时可以连成一串字符输出



Python的**注释**用“#”开头，后面的文字直到行尾都是注释部分。

**变量：**

Python中，变量用变量名表示，变量名必须是 **大小写英文、数字和下划线“\_”组成，且不能以数字开头**。

如： a=1 #变量a为整数

t\_001=’007’ #变量t\_001为字符串

同一个变量可以反复赋值，而且可以是不同类型的变量。 变量不固定----称为“动态语言”

定义字符串--转译

字Python中字符串可以用 ’ ‘ 或者 “ “括起来表示

两种都在字符串中时，需要用 **\** 进行转译。

如：Bob said “ I’m OK” .

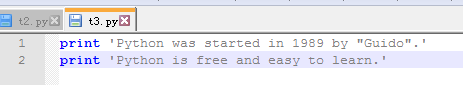
此时需要转义字符进行表示： ‘Bob said \ ”I \ ’m OK \” . ‘

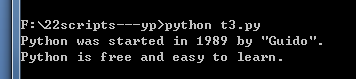
其他常用的转译字符：

\n 表示换行

\t 表示一个制表符

\\ 表示\字符本身





~ raw字符串和多行字符串

字符串中含有多个转义字符\时，可以在字符串前面加r 前缀，表示里面的字符不再需要转译。

如： r ‘\(~\_~) / \ (~\_~)/’

但是r‘…’不能表示多行字符串，也不能表示含有’’和””的字符串

**表示多行字符串时，可以用’’’…’’’表示：**

如

’’’Line1

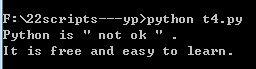
Line2

Line3…’’’

等同于表示：’Line1\nLine2 \nLine3’

也可以在多行字符串前加r，把多行字符串变成raw字符串。

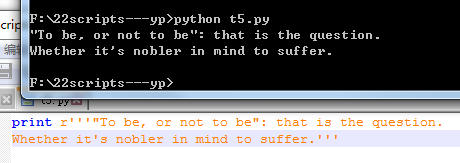




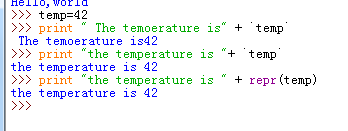
注意：**下面的r和’’’之间不能有空格！！！**







### repr()函数



用repr()函数，创建一个字符串，以合法的Python表达式的形式来表示。

repr()函数可以用反引号代替（` `）

>>> temp=42

>>> print " The temoerature is" + `temp`

The temoerature is42

>>> print "the temperature is "+ `temp`

the temperature is 42

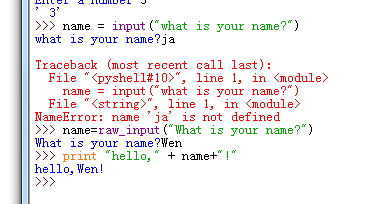
>>> print "the temperature is " + repr(temp)

the temperature is 42

### raw\_input()函数

input()函数会假设用户输入的是合法的Python表达式

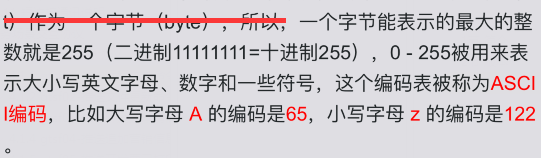
raw\_input()函数会把所有的输入当做原始数据（raw data）

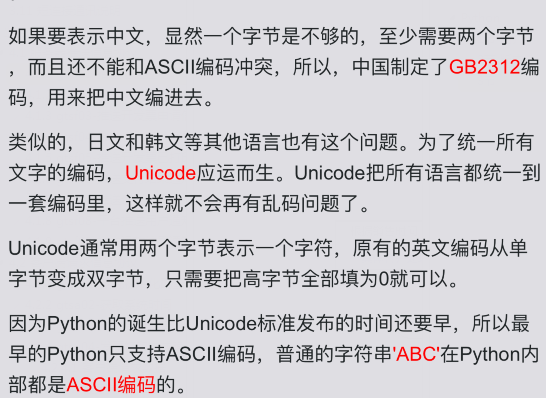


上述，input() 后输入ja时，加引号，则可以输入

除非input()有特别需要，否则尽可能使用raw\_input()

### Unicode字符串





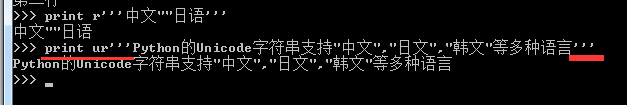
Python在后来添加了对Unicode的支持，以Unicode表示的字符串用u‘…’表示，如：

print u‘中文’

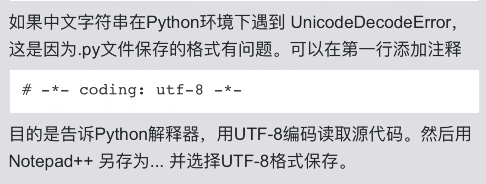


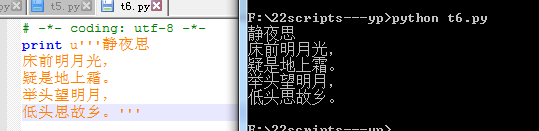
Unicode除了多加一个u之外，与普通字符没有区别，转译字符和多行表示法仍然有效：

Raw+多行

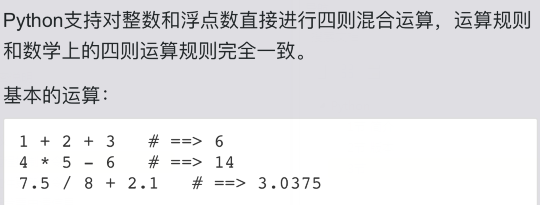


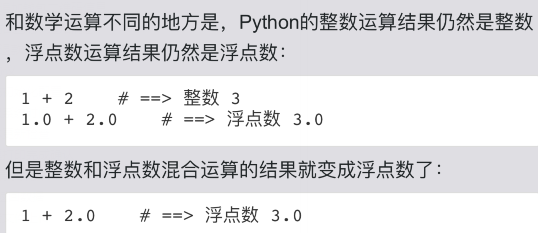
**# -\*- coding: utf-8 -\*-**

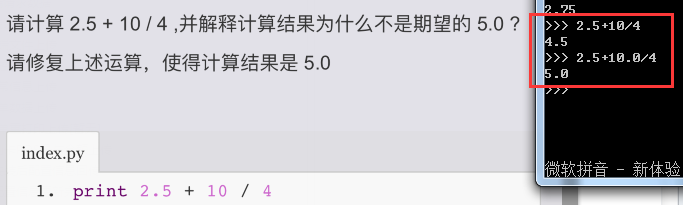




整数和浮点数

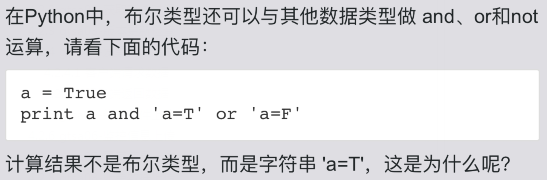


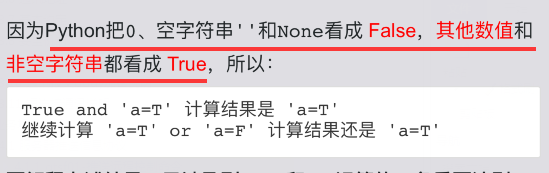


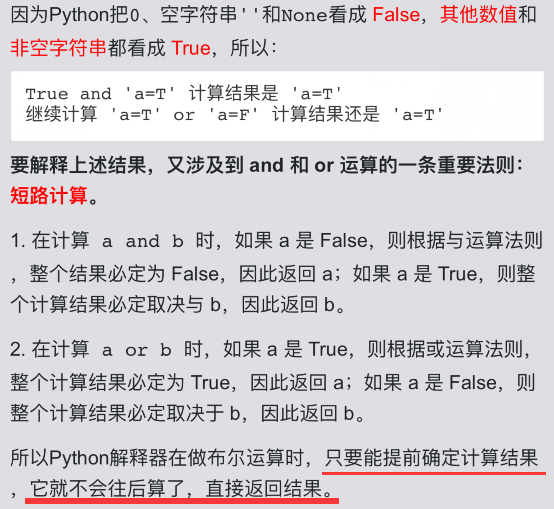


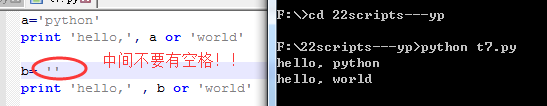
### 3．2节 布尔类型

布尔类型只有True和False两种值。





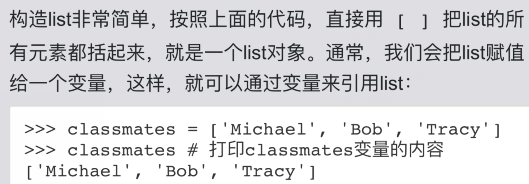




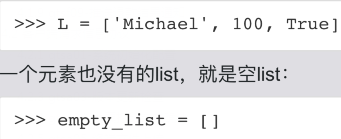
### 4节 创建list

Python中内置的一种数据类型是列表:list。 List是一种**有序的集合**，可以随时添加和删除其中的元素。



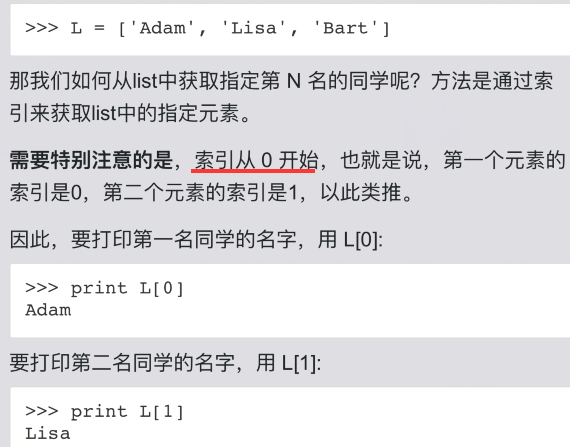


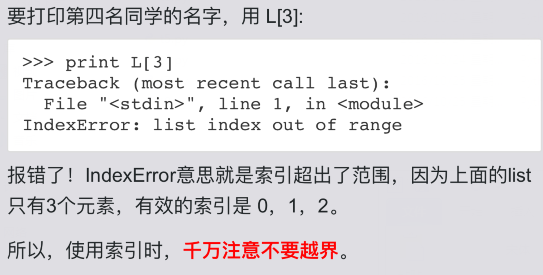
Python是动态语言，所以list中的元素可以包含各种数据：





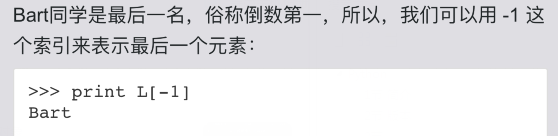
4.2 按照索引方式访问list

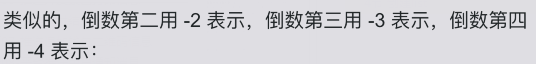




4-3 倒序访问list

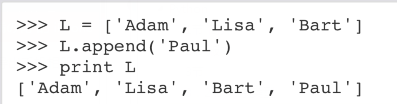
L=[‘Adam’, ‘Lisa’, ‘Bart’]





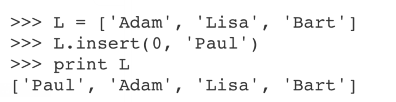
### 4-4 添加新元素

1. 用list的append()方法，把新同学追加到list的末尾：

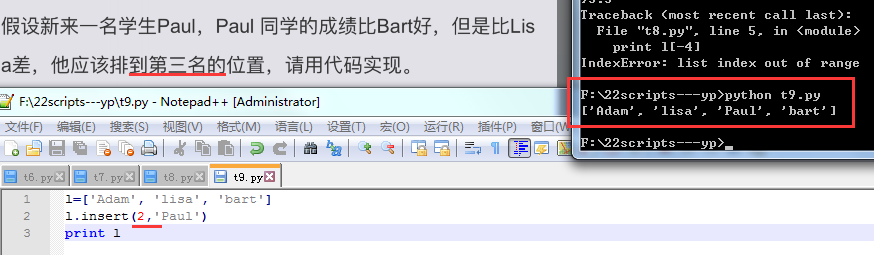


注意：append()总数把新元素添加到list尾部。

1. 用list的insert() 方法，它接受两个参数，第一个参数是索引号，第二个参数是待添加的元素：

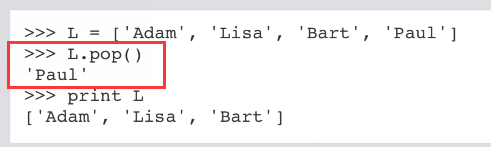


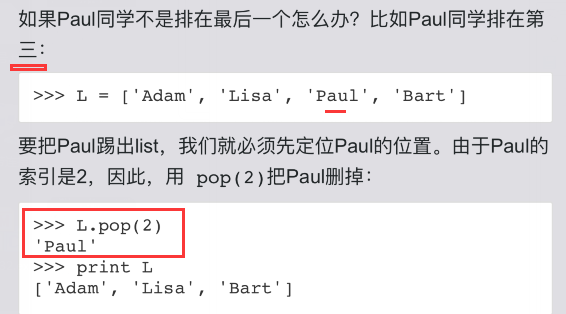
注： L.insert(0, ‘Paul’)：’Paul’将被添加到索引为0的位置，原来索引为0的元素以及后面的元素，都自动向后移动一位。

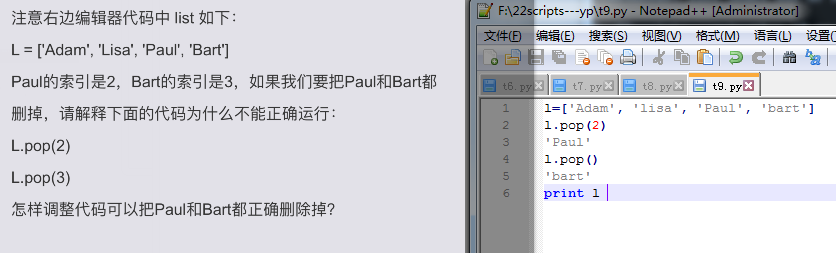


### 4-5 从list删除元素

若删除最后一个元素，可以用list的pop()方法删除

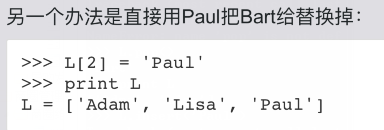


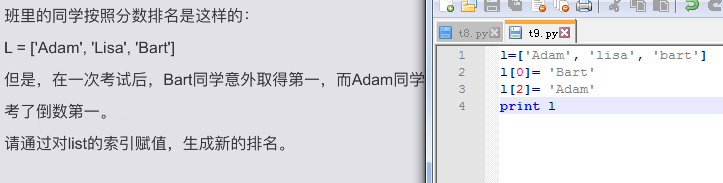




### 4-6替换元素

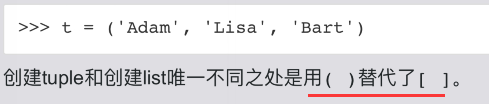
1. 删除原来元素，添加新元素
2. 直接替换：对list中的某个索引直接赋值，可以用新元素替换原来元素，list包含元素个数保持不变。





### 4-7 创建tuple

tuple是另一种有序的列表，中文翻译为“元组”。Tuple和list类似，区别：tuple一旦创建完毕就不能修改了。



创建之后，tuple不能修改，所以tuple 没有append()方法，也没有insert() 和pop()方法。

获取tuple元素的方式和list一样，用t[0], t[1]… t[-1]等索引方式访问元素。

### 4-8 创建单元素tuple

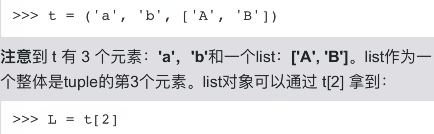
Tuple可以含有0个，1个和任意多个元素

包含0个元素的为：空tuple，直接用()表示：

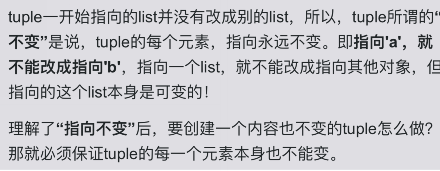
用（）定义单元素的tuple有歧义，所以Python规定，单元素tuple要多加一个逗号“,”



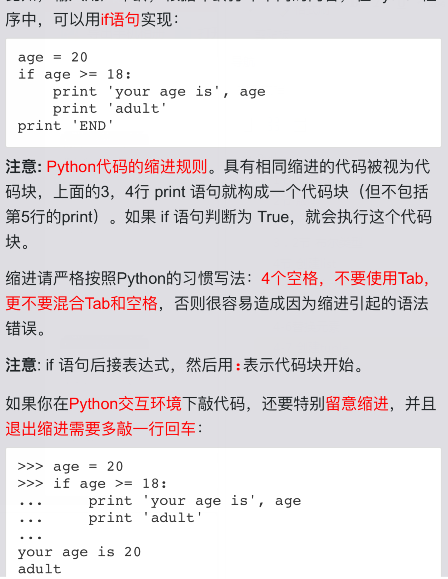
### 4-9 “可变的”tuple

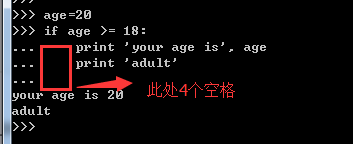


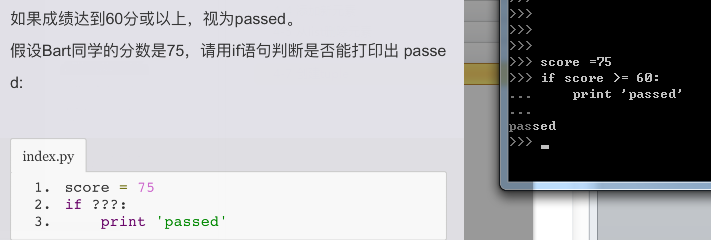




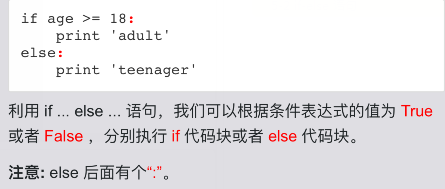
### 5-1 if语句



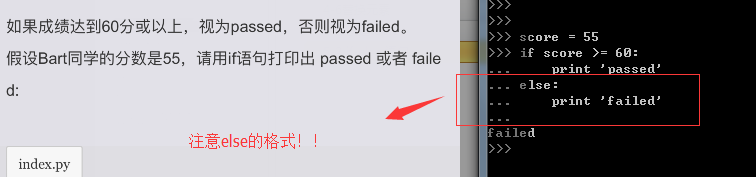




### 5-2 if-else 语句

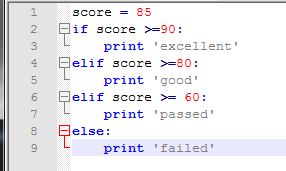


注意else后面的冒号！！！



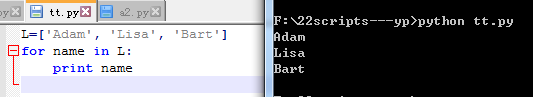
### 5-3 if-elif-else语句

嵌套多层判断语句时，使用if-elif-else语句



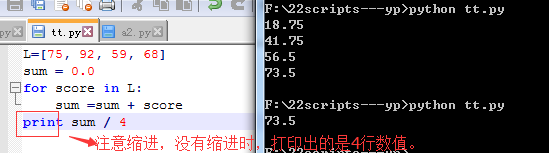
### 5-4 for循环

Python中的for循环可以依次把list或tuple中的每个元素迭代出来：



注： name这个变量是在for循环中定义的：取出list中的每一个元素，并赋值给name, 然后执行for循环体。

下例子：求L数列的平均值：

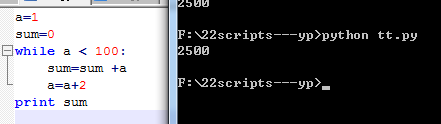


5-5 while循环

For循环：迭代list或tuple元素

While循环：根据表达式判断循环是否结束。

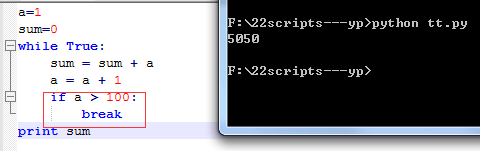
例子：用while循环计算100以内的奇数和。



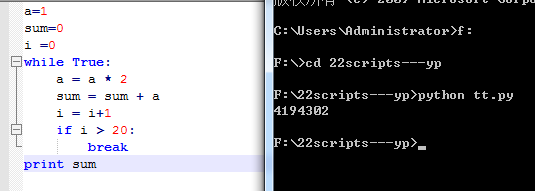
### 5-6 break退出循环

用for循环或者while循环时，如果在循环体内直接退出循环，可以使用break语句。

例子：计算100以内的整数和。

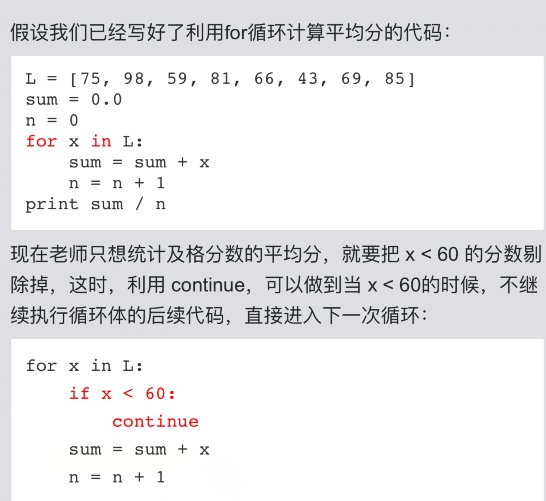


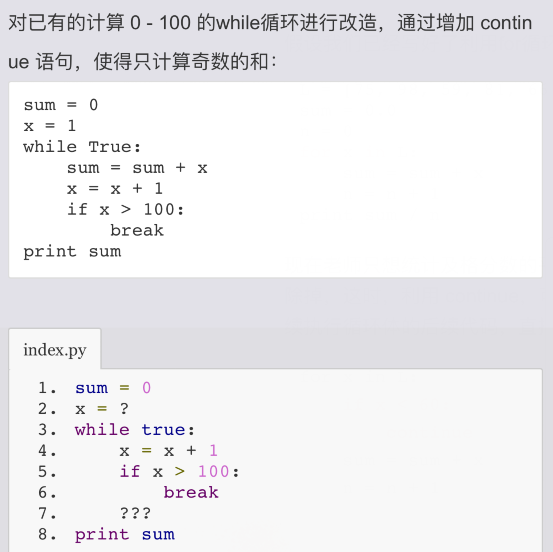
例子：利用while True无限循环配合break语句，计算1+2+4+8+16的前20项的和

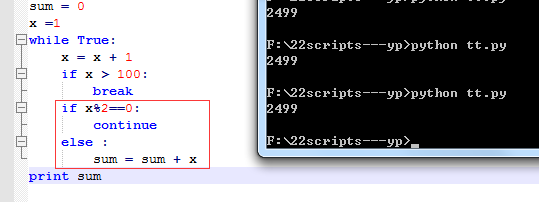


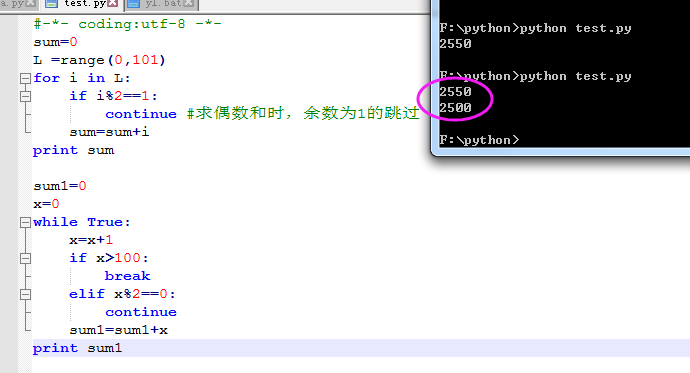
### 5-7 continue继续循环

用continue**跳过后续循环代码，继续下一次循环**。



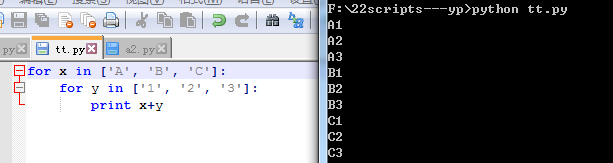




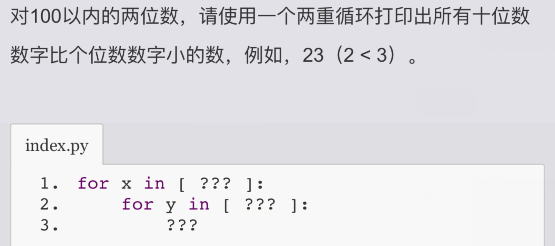


### 5-8 多重循环

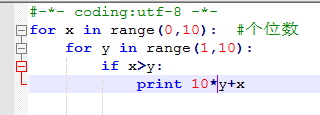
在循环内部可以嵌套循环：



例题：

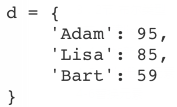






### 6-1 dict介绍

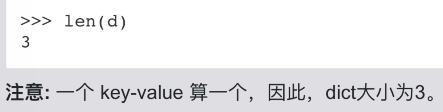
用dict表示“名字”-“成绩”的查找表如下：



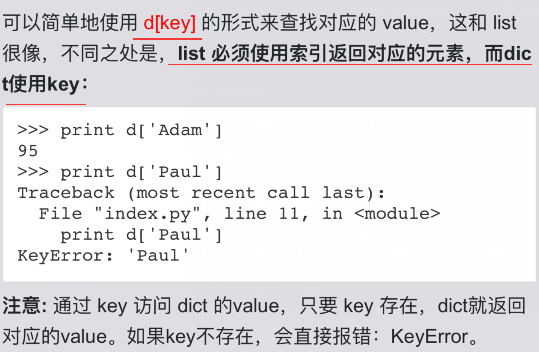
其中，把名字称为Key，对应成绩称为 Value，**dict就是通过key查找value**。

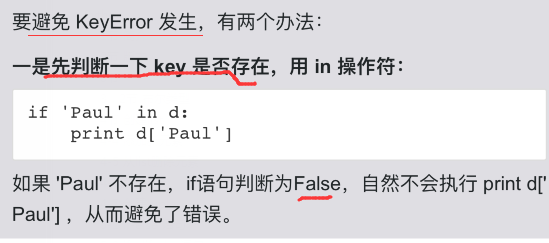


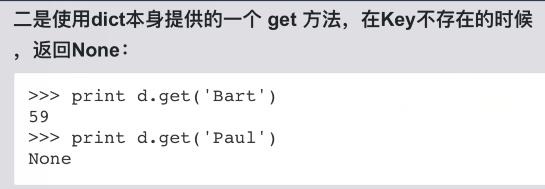
由于dict也是集合，len()函数可以计算任意集合的大小：



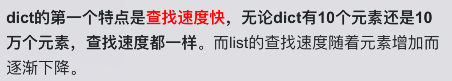
### 6-2 访问dict





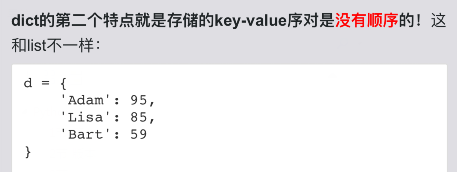


### 6-3 dict特点

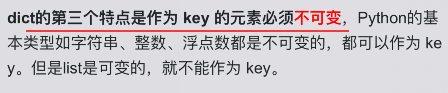
1）

但是查找速度快以牺牲内存为代价

由于dict是按照Key查找，所以**dict中的key不能重复**。

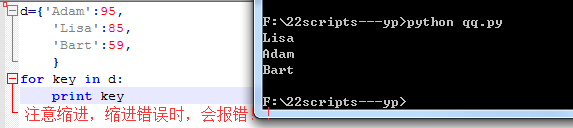


**Dict中存储的key-value 是没有顺序的，不能用dict存有序的集合。**

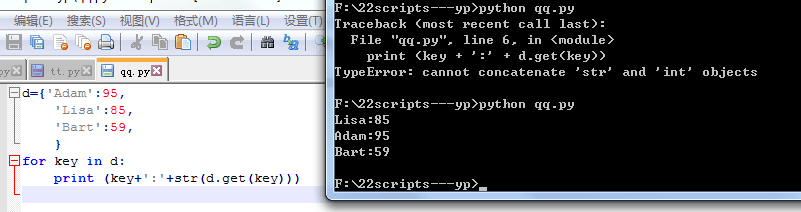


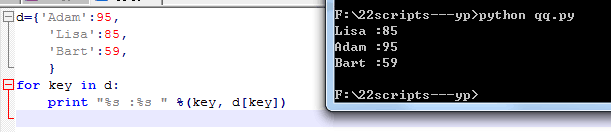
### 6-5 遍历dict

遍历dict可以通过for循环实现

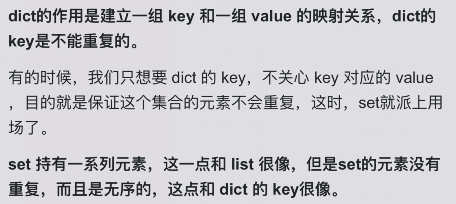


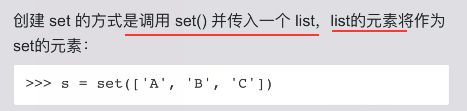
以key:value的方式遍历出d



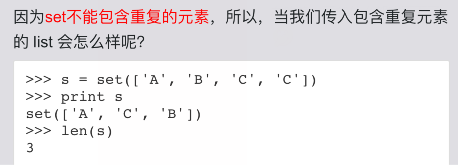


### 6-6 什么是set

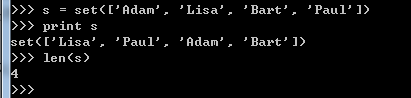




**Set中元素也是无序的！**



说明：set会自动去掉重复的元素！

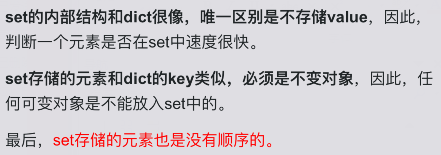


### 6-7 访问set

用in操作符判断：



### 6-8 set的特点：



示例：用户输入周一至周日的某天，判断用户是否输入值有效：

1. 用if语句判断时，非常繁琐。

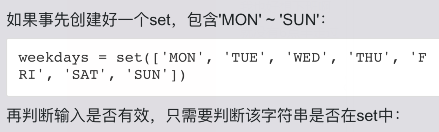
if x ! = ‘ MON’ and x ! = ‘TUE’ and x != ‘WED’ and ```

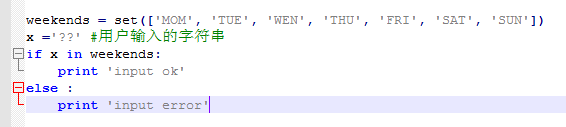
print ‘input error’

else

print ‘input ok’

2)直接用set语句输入：



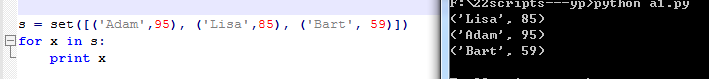


示例：判断输入月份是否有效：



### 6-9 遍历set

通过for语句实现：



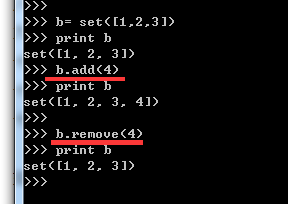
6-10 更新set

原因：set中存储的是不重复的无序元素。

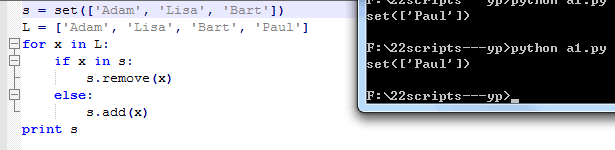
更新时：新元素添加到set中，已有元素从set中删除。

添加元素时，用set的add()方法：

删除元素时，用set的remove()方法：



示例：对于L中的元素，s中有的话删除，没有的话，添加。



### 7-1 函数介绍

函数是最基本的一种代码抽象的方式。

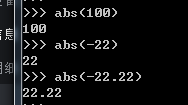
Python可以灵活的定义函数，而且本身内置了很多有用的函数，可以直接调用。

### 7-2 调用函数

调用函数时，需要知道函数的名称和参数。

如：求绝对值的函数abs:

调用abs函数

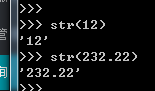


其他函数如：

比较函数 cmp(x, y) 需要两个参数

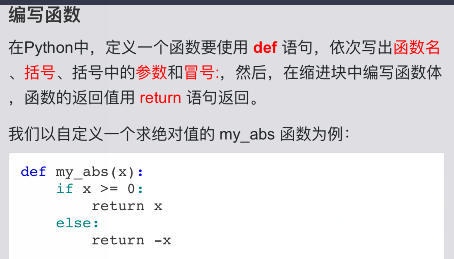
数据类型转换函数 ： 其他数据类型转换为整数 int()

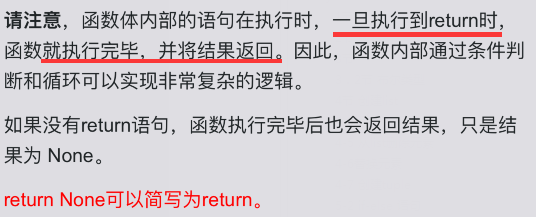
其他类型转换为str()



?????本小节例题没解决

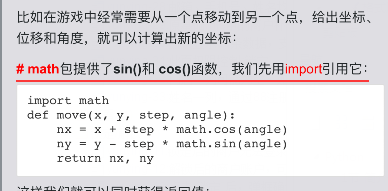
### 7-3 编写函数





定义默认参数时，def中必须有返回值，要用return返回，不能用print，否则显示None

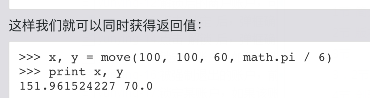
### 7-4 返回多值

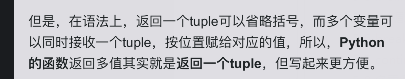


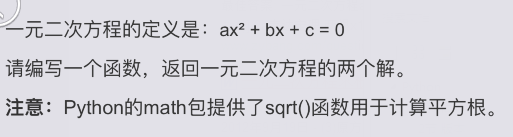
# import math

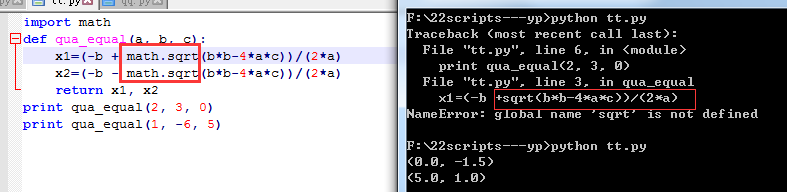
Or #

注意此处写为： **math.cos() 和math.sin()**



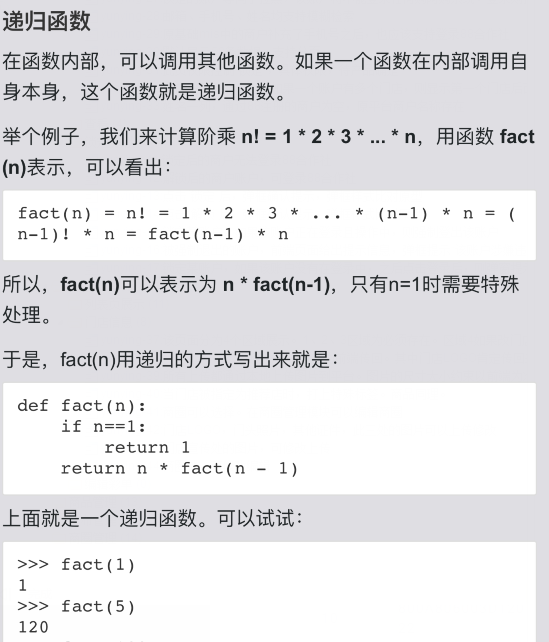




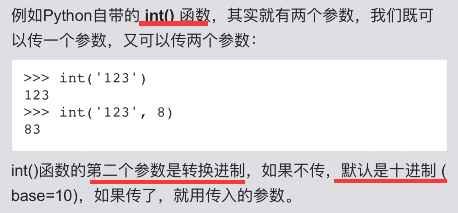


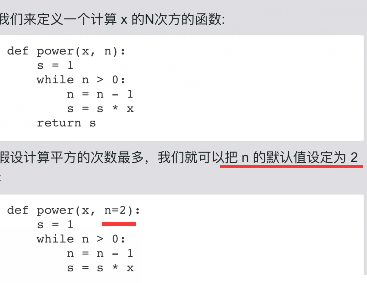
注意调用时，要用math.sqrt()

### 7-5 递归函数



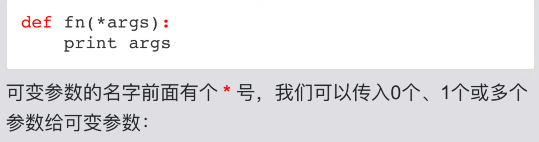
### 7-6 定义默认参数

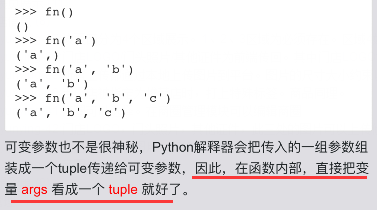




定义默认参数时，def中必须有返回值，要用return返回，不能用print

7-7 定义可变参数





### 8-1 对list进行切片

以往状况下用列举或for语句进行切片。

但是Python中提供了切片（Slice）操作符，可以简化操作。

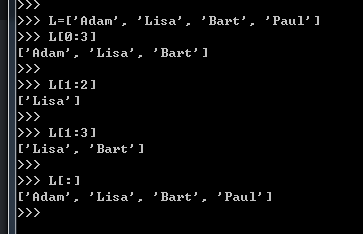
如取L中的前三个元素，可以表示为**L[0:3]**

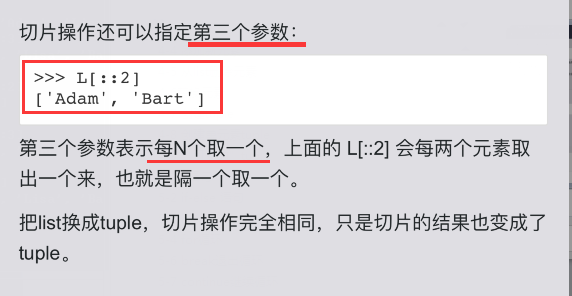
表示从索引0开始，知道索引3为止，但不包括索引3.

第一个索引为0时，还可以表示为**L[:3]**

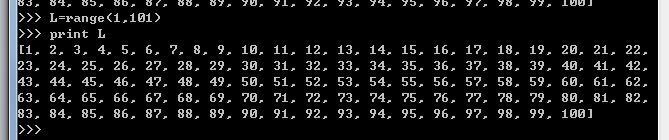
L从1开始，取出第2个元素，表示为：L[1:2]

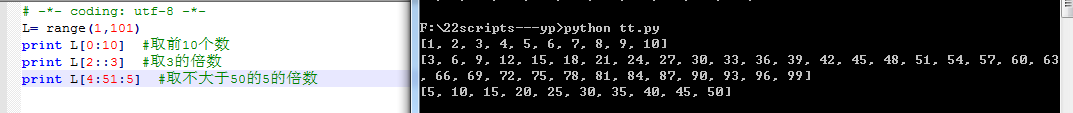
只用：表示时，表示从头到尾 L[:]





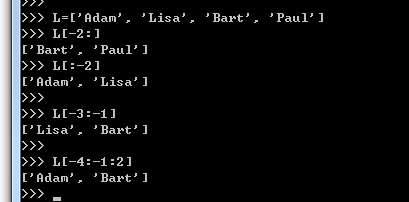






### 8-2 倒序切片

在list中，Python支持[-1]取倒数第一个元素，同样支持倒数切片：

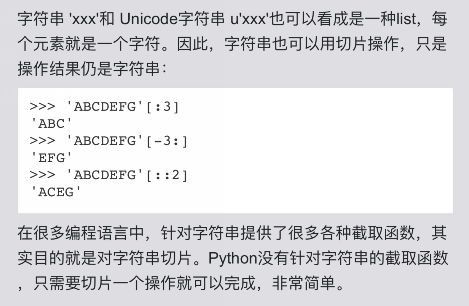


倒数第一个元素的索引是-1,。倒序切片包括起始索引，不包含结束索引。

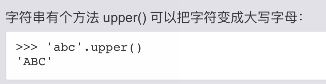
例子：



### 8-3 对字符串切片



大写字母转换方法： upper()

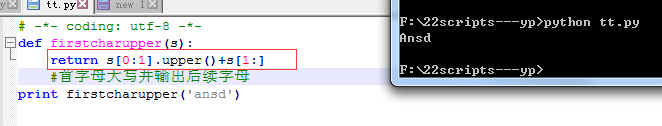


例题：输出中首字母大写，其他字母不变

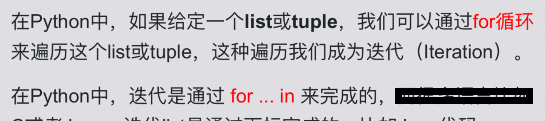
定义函数;

**def firstcharupper(s):**

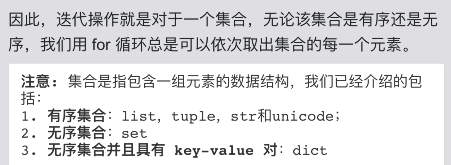
**return s[0:1].upper()+s[1:]**



### 9-1 什么是迭代？

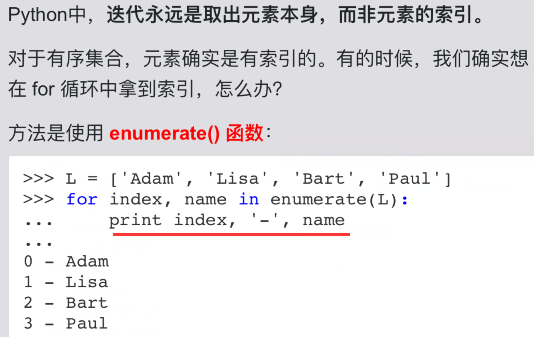


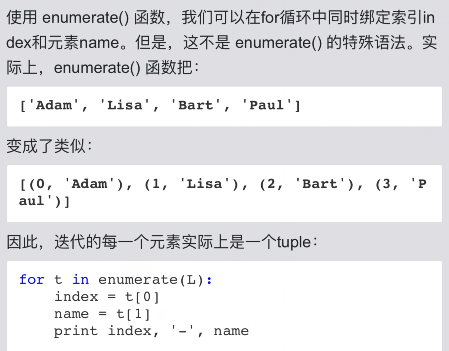
Python中的for循环，不仅可以用在list或tuple上，还可以作用在其他任何可迭代对象上。

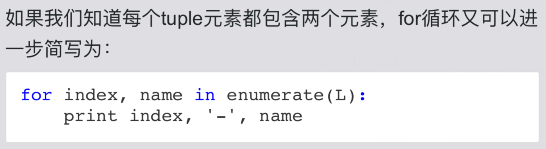


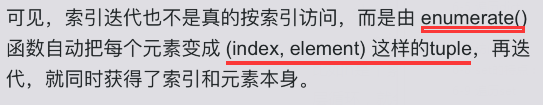
### 9-2 索引迭代

在for循环中取得索引办法：用enumerate()函数

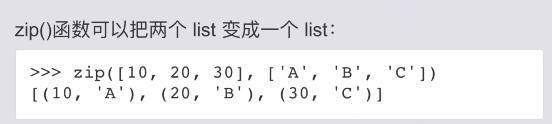


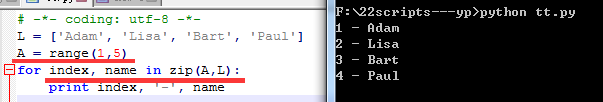






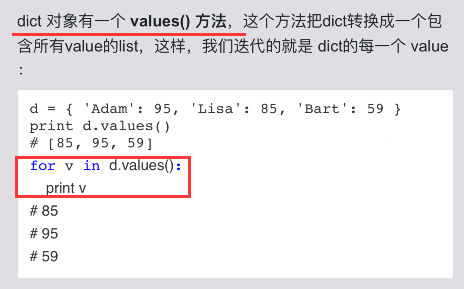
zip()函数：



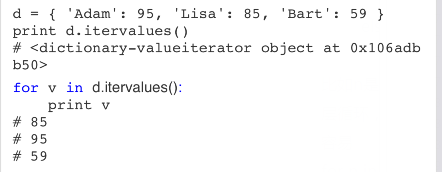


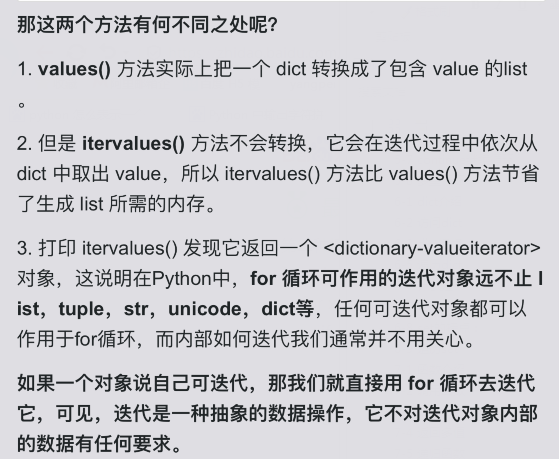
### 9-3 迭代dict的value

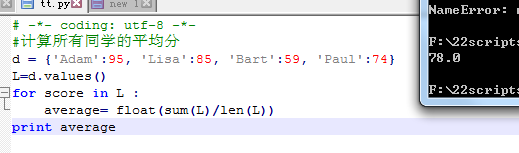




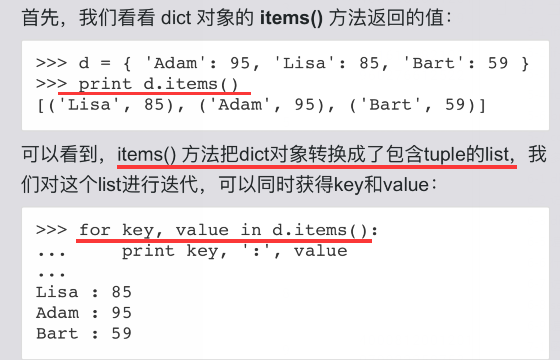
dict除了values()方法，还有一个itervalues()方法，迭代效果完全一样：

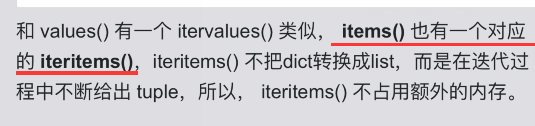


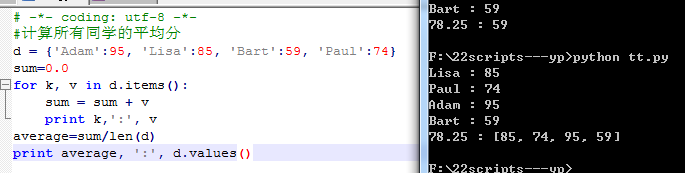




9-4 迭代dict的key和value





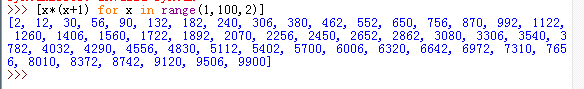


10-1 生成列表



以上代码繁琐，可以用一行语句代替，是Python特有的列表生成式

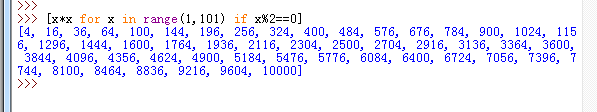




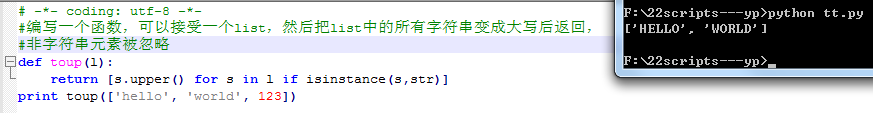
### 10-2 复杂表达式

### 10-3 条件过滤

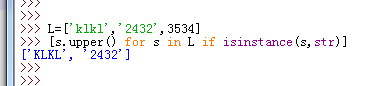
列表生成式的for循环还可以加上if判断：





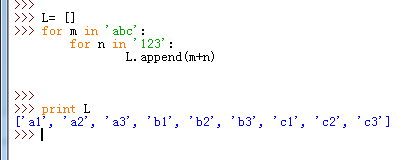


只输出L中的字符串，并大写，如下：

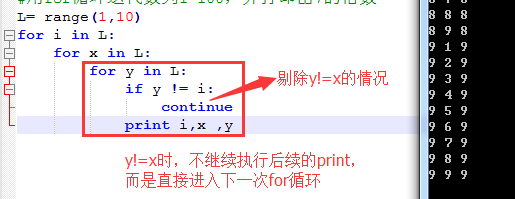


### 10-4 多层表达式

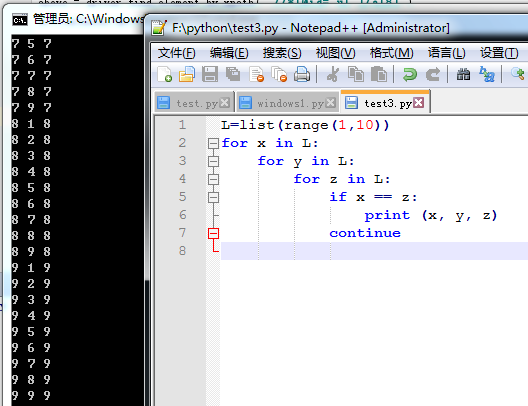




例子：用3层for循环的列表生成式，找出对称的3位数：



或者：



## Py2

产生一个随机的整数，就在程序的最开头写上：  
**from random import randint**

如果你想要**把一个数字加到文字后面输出**，比如这样  
num = 18  
print( 'My age is' + num)  
程序就会报错。因为**字符和数字不能直接用+相加**。

解决方法是，用str()把数字转换成字符串

print ('My age is' + str(18))

或  
num = 18

print ('My age is' + str( num))

### python中的格式符汇总

%s 字符串 (采用str()的显示)

%r 字符串 (采用repr()的显示)

%c 单个字符

%b 二进制整数

%d 十进制整数

%i 十进制整数

%o 八进制整数

%x 十六进制整数

%e 指数 (基底写为e)

%E 指数 (基底写为E)

%f 浮点数

%F 浮点数，与上相同

%g 指数(e)或浮点数 (根据显示长度)

%G 指数(E)或浮点数 (根据显示长度)

%% 字符'%'

%[(name)][flags][width].[precision]typecode

(name)为命名

flags可以有+,-,' '或0。+表示右对齐。-表示左对齐。' '为一个空格，表示在正数的左侧填充一个空格，从而与负数对齐。0表示使用0填充。

width表示显示宽度

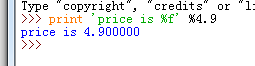
precision表示小数点后精度

example:

print('%6.3f' % 2.3)

还有一种方法，就是用%对字符串进行格式化

num = 18

print ('My age is %d' %num)  
输出的时候，%d会被%后面的值替换。输出  
My age is 18  
  
  
这里，%d只能用来替换整数。如果你想格式化的数值是小数，要用%f  
print ('Price is %f' % 4.99)  
输出  
Price is 4.990000  


如果你想保留两位小数，需要在f前面加上条件：%.2f

print ('Price is %.2f' % 4.99)

输出

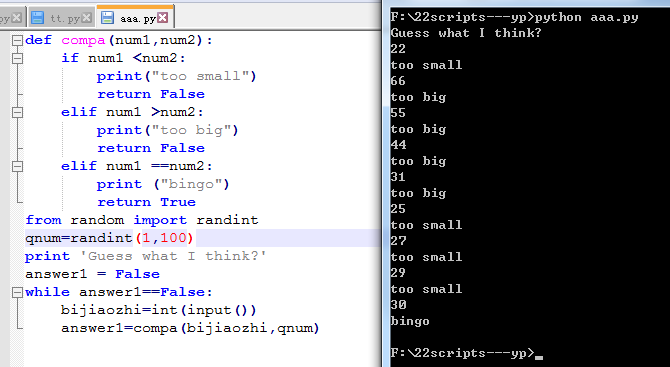
Price is 4.99

注意区分：有引号的表示一段字符，没有引号的就是一个变量，这个变量可能是字符，也可能是数字，但一定要和%所表示的格式相一致。

print ‘Hello’+1  
print ‘hello%d’ % '123'  
  
  
程序运行时会报错。因为第一句里，字符串和整数不能相加；第二句里，%d需要的是一个整数，而'123'是字符串。

在python中，以下数值会被认为是False：  
为0的数字，包括0，0.0  
空字符串，包括''，""  
表示空值的None  
空集合，包括()，[]，{}; 注释：{}表示字典，[]是数组，()是元组；  
其他的值都认为是True。

小程序：



def compa(num1,num2):

if num1 <num2:

print("too small")

return False

elif num1 >num2:

print("too big")

return False

elif num1 ==num2:

print ("bingo")

return True

from random import randint

qnum=randint(1,100)

print 'Guess what I think?'

answer1 = False

while answer1==False:

bijiaozhi=int(input())

answer1=compa(bijiaozhi,qnum)

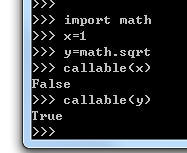
## Python3 进阶

<http://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/0014318230588782cac105d0d8a40c6b450a232748dc854000>

### 函数：

变量可以指向函数

内建的callable函数可以判断函数是否可以调用 （callable在3.0版本中不再可用）

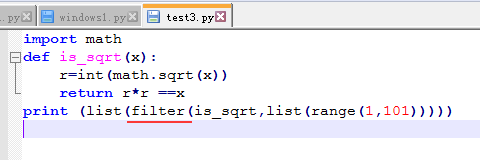


### 2. filter 函数 filter(f, list)

函数f作用：对每个元素进行判断，返回True或False

返回又符合条件的元素组成的新的 list

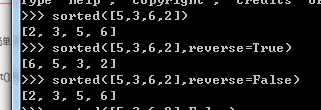
在Python3中使用时加list 才可以展示为列表形式

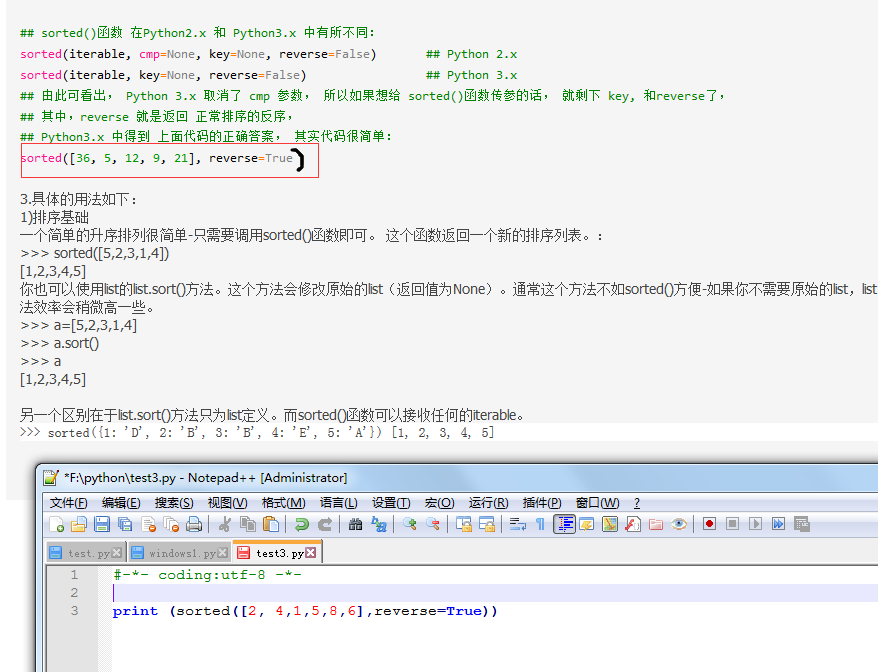




### 3. sorted()函数：Python内置的自定义排序函数

sorted([36, 5, 12, 9, 21], reverse=True)





<http://blog.csdn.net/windows_2015/article/details/51498505>

>>> sorted(['bob', 'about', 'Zoo', 'Credit'], key=str.lower)

['about', 'bob', 'Credit', 'Zoo']



### 4. 返回函数

高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回。

我们来实现一个可变参数的求和。通常情况下，求和的函数是这样定义的：

def calc\_sum(\*args):

ax = 0

for n in args:

ax = ax + n

return ax

但是，如果不需要立刻求和，而是在后面的代码中，根据需要再计算怎么办？可以不返回求和的结果，而是返回求和的函数：

def lazy\_sum(\*args):

def sum():

ax = 0

for n in args:

ax = ax + n

return ax

return sum

当我们调用lazy\_sum()时，返回的并不是求和结果，而是求和函数：

>>> f = lazy\_sum(1, 3, 5, 7, 9)

>>> f

<function lazy\_sum.<locals>.sum at 0x101c6ed90>

调用函数f时，才真正计算求和的结果：

>>> f()

25

在这个例子中，我们在函数lazy\_sum中又定义了函数sum，并且，内部函数sum可以引用外部函数lazy\_sum的参数和局部变量，当lazy\_sum返回函数sum时，相关参数和变量都保存在返回的函数中，这种称为“**闭包（Closure）**”的程序结构拥有极大的威力。

请再注意一点，当我们调用lazy\_sum()时，每次调用都会返回一个新的函数，即使传入相同的参数：

>>> f1 = lazy\_sum(1, 3, 5, 7, 9)

>>> f2 = lazy\_sum(1, 3, 5, 7, 9)

>>> f1==f2

False

f1()和f2()的调用结果互不影响。

### 闭包

注意到返回的函数在其定义内部引用了**局部变量****args**，所以，当一个函数返回了一个函数后，其内部的局部变量还被新函数引用，所以，闭包用起来简单，实现起来可不容易。

另一个需要注意的问题是，返回的函数并没有立刻执行，而是直到调用了f()才执行。

def count():

fs = []

for i in range(1, 4):

def f():

return i\*i

fs.append(f)

return fs

f1, f2, f3 = count()

在上面的例子中，每次循环，都创建了一个新的函数，然后，把创建的3个函数都返回了。

你可能认为调用f1()，f2()和f3()结果应该是1，4，9，但实际结果是：

>>> f1()

9

>>> f2()

9

>>> f3()

9

全部都是9！原因就在于返回的函数引用了变量i，但它并非立刻执行。等到3个函数都返回时，它们所引用的变量i已经变成了3，因此最终结果为9。

返回闭包时牢记的一点就是：返回函数不要引用任何循环变量，或者后续会发生变化的变量。

如果一定要引用循环变量怎么办？方法是再创建一个函数，用该函数的参数绑定循环变量当前的值，无论该循环变量后续如何更改，已绑定到函数参数的值不变：

def count():

def f(j):

def g():

return j\*j

return g

fs = []

for i in range(1, 4):

fs.append(f(i)) # f(i)立刻被执行，因此i的当前值被传入f()

return fs

再看看结果：

>>> f1, f2, f3 = count()

>>> f1()

1

>>> f2()

4

>>> f3()

9

一个函数可以返回一个计算结果，也可以返回一个函数。

返回一个函数时，牢记该函数并未执行，返回函数中不要引用任何可能会变化的变量。

### 匿名函数

当我们在传入函数时，有些时候，不需要显式地定义函数，直接传入匿名函数更方便。

在Python中，**对匿名函数提供了有限支持**。还是以map()函数为例，计算f(x)=x2时，除了定义一个f(x)的函数外，还可以直接传入匿名函数：

>>> list(map(lambda x: x \* x, [1, 2, 3, 4, 5, 6, 7, 8, 9]))

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

通过对比可以看出，匿名函数**lambda x: x \* x**实际上就是：

def f(x):

return x \* x

关键字lambda表示匿名函数，冒号前面的x表示函数参数。

匿名函数有个限制，**就是只能有一个表达式，不用写return，返回值就是该表达式的结果。**

用匿名函数有个好处，因为函数没有名字，不必担心函数名冲突。此外，匿名函数也是一个函数对象，也可以把匿名函数赋值给一个变量，再利用变量来调用该函数：

>>> f = lambda x: x \* x

>>> f

<function <lambda> at 0x101c6ef28>

>>> f(5)

25

同样，也可以把匿名函数作为返回值返回，比如：

def build(x, y):

return lambda: x \* x + y \* y

Python对匿名函数的支持有限，只有一些简单的情况下可以使用匿名函数。

### 装饰器

由于函数也是一个对象，而且函数对象可以被赋值给变量，所以，通过变量也能调用该函数。

>>> def now():

... print('2015-3-25')

...

>>> f = now

>>> f()

2015-3-25

函数对象有一个\_\_name\_\_属性，可以拿到函数的名字：

>>> now.\_\_name\_\_

'now'

>>> f.\_\_name\_\_

'now'

其中f.\_\_name\_\_中，name前后各有**2个下划线**

现在，假设我们要增强now()函数的功能，比如，在函数调用前后自动打印日志，但又不希望修改now()函数的定义，这种**在代码运行期间动态增加功能的方式，称之为“装饰器”（Decorator）。**

本质上，decorator就是一个返回函数的高阶函数。所以，我们要定义一个能打印日志的decorator，可以定义如下：

def log(func):

def wrapper(\*args, \*\*kw):

print('call %s():' % func.\_\_name\_\_)

return func(\*args, \*\*kw)

return wrapper

观察上面的log，因为它是一个decorator，所以接受一个函数作为参数，并返回一个函数。我们要借助Python的@语法，把decorator置于函数的定义处：

@log

def now():

print('2015-3-25')

调用now()函数，不仅会运行now()函数本身，还会在运行now()函数前打印一行日志：

>>> now()

call now():

2015-3-25

把@log放到now()函数的定义处，相当于执行了语句：

now = log(now)

由于log()是一个decorator，返回一个函数，所以，原来的now()函数仍然存在，只是现在同名的now变量指向了新的函数，于是调用now()将执行新函数，即在log()函数中返回的wrapper()函数。

wrapper()**函数的参数定义是(\*args, \*\*kw)，因此，wrapper()函数可以接受任意参数的调用。**在wrapper()函数内，首先打印日志，再紧接着调用原始函数。

如果decorator本身需要传入参数，那就需要编写一个返回decorator的高阶函数，写出来会更复杂。比如，要自定义log的文本：

def log(text):

def decorator(func):

def wrapper(\*args, \*\*kw):

print('%s %s():' % (text, func.\_\_name\_\_))

return func(\*args, \*\*kw)

return wrapper

return decorator

这个3层嵌套的decorator用法如下：

@log('execute')

def now():

print('2015-3-25')

执行结果如下：

>>> now()

execute now():

2015-3-25

和两层嵌套的decorator相比，3层嵌套的效果是这样的：

>>> now = log('execute')(now)

我们来剖析上面的语句，首先执行log('execute')，返回的是decorator函数，再调用返回的函数，参数是now函数，返回值最终是wrapper函数。

以上两种decorator的定义都没有问题，但还差最后一步。因为我们讲了函数也是对象，它有\_\_name\_\_等属性，但你去看经过decorator装饰之后的函数，它们的\_\_name\_\_已经从原来的'now'变成了'wrapper'：

>>> now.\_\_name\_\_

'wrapper'

因为返回的那个wrapper()函数名字就是'wrapper'，所以，需要把原始函数的\_\_name\_\_等属性复制到wrapper()函数中，否则，有些依赖函数签名的代码执行就会出错。

不需要编写wrapper.\_\_name\_\_ = func.\_\_name\_\_这样的代码，Python内置的functools.wraps就是干这个事的，所以，一个完整的decorator的写法如下：

import functools

def log(func):

@functools.wraps(func)

def wrapper(\*args, \*\*kw):

print('call %s():' % func.\_\_name\_\_)

return func(\*args, \*\*kw)

return wrapper

或者针对带参数的decorator：

import functools

def log(text):

def decorator(func):

@functools.wraps(func)

def wrapper(\*args, \*\*kw):

print('%s %s():' % (text, func.\_\_name\_\_))

return func(\*args, \*\*kw)

return wrapper

return decorator

import functools是导入functools模块。模块的概念稍候讲解。现在，只需记住在定义wrapper()的前面加上@functools.wraps(func)即可。

### 小结

在面向对象（OOP）的设计模式中，decorator被称为装饰模式。OOP的装饰模式需要通过继承和组合来实现，而Python除了能支持 OOP的decorator外，直接从语法层次支持decorator。**Python的decorator可以用函数实现，也可以用类实现**。

decorator可以增强函数的功能，定义起来虽然有点复杂，但使用起来非常灵活和方便。

请编写一个decorator，能在函数调用的前后打印出'begin call'和'end call'的日志。

再思考一下能否写出一个@log的decorator，使它既支持：

@log

def f():

pass

又支持：

@log('execute')

def f():

pass

当函数的参数不确定时，可以使用\*args 和\*\*kwargs

# 模块

在Python中，一个.py文件就称之为一个模块（Module）。

为了避免模块名冲突，Python又引入了按目录来组织模块的方法，称为包（Package）。

举个例子，一个abc.py的文件就是一个名字叫abc的模块，一个xyz.py的文件就是一个名字叫xyz的模块。引入了包以后，只要顶层的包名不与别人冲突，那所有模块都不会与别人冲突。现在，abc.py模块的名字就变成了mycompany.abc，类似的，xyz.py的模块名变成了mycompany.xyz。

**每一个包目录下面都会有一个\_\_init\_\_.py的文件，这**个文件是必须存在的，否则，Python就把这个目录当成普通目录，而不是一个包。\_\_init\_\_.py可以是空文件，也可以有Python代码，因为\_\_init\_\_.py本身就是一个模块，而它的模块名就是mycompany。

## 使用模块



第1行和第2行是标准注释，第1行注释可以让这个hello.py文件直接在Unix/Linux/Mac上运行，第2行注释表示.py文件本身使用标准UTF-8编码；

第4行是一个字符串，表示模块的文档注释，任何模块代码的第一个字符串都被视为模块的文档注释；

第6行使用\_\_author\_\_变量把作者写进去，这样当你公开源代码后别人就可以瞻仰你的大名；

以上就是Python模块的标准文件模板，当然也可以全部删掉不写

使用sys模块的第一步，就是导入该模块：

导入sys模块后，我们就有了变量sys指向该模块，利用sys这个变量，就可以访问sys模块的所有功能。

sys模块有一个argv变量，用list存储了命令行的所有参数。argv至少有一个元素，因为第一个参数永远是该.py文件的名称，例如：

运行python3 hello.py获得的sys.argv就是['hello.py']；

运行python3 hello.py Michael获得的sys.argv就是['hello.py', 'Michael]。

## 水电费三