**Dictionary: key+value**

D={‘one’:1,’two’:2} f

D=dict(one=1,two=2)

Connect two dictionary into one:

T=dict(three=3,four=4,\*\*D)

Merits of dictionary:

1:the key can be any types

2:don’t need initialization

x=[],x[3]=1 -> wrong –(list assignment index out of range)

x={},x[3]=1 ->right

shallow copy and deep copy comparison:

from copy import deepcopy

d={}

d[‘name’]=[‘peter’,’david’]

c=d.copy() #shallow copy

dc=deepcopy(d) #deep copy

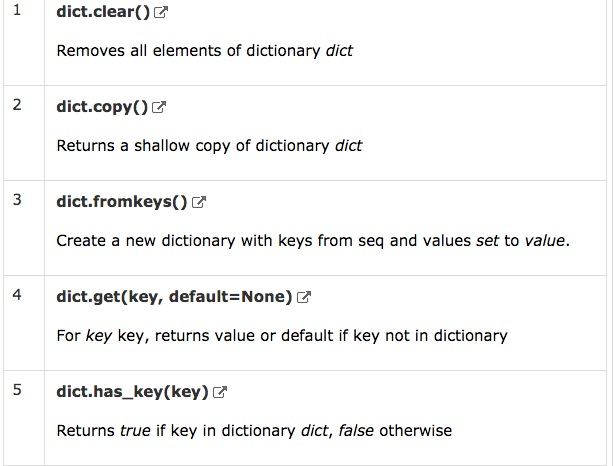
d[‘name’].append(‘mike’)

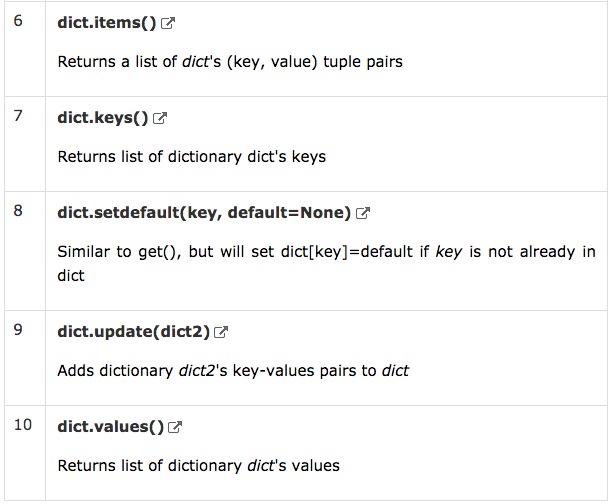
c->{‘name’:[‘peter’,’david’,’mike’]}

dc->{‘name’:[‘peter’,’david’]}

get：if we want to visit the terms which is not exiting in the dictionary, we won’t have exception, instead we will get the None value.

**access the key and value in dictionary:**





dictionary 也可以这样赋值：dict[2]=2,(我犯的错误，dict[two]=2，因为two 既不是str 也不是num,所以报错)， 事实上dict[key]=value, 其中key可以是value,

string, tuple ,但不能是list 和dictionary.

在python中，任何的不可变的对象都可以当作key,如 strings, number,tuple,相反的，包含可变对象的list, dict等不能当作key

if key in d: is just a call to a \_\_contains\_\_ method python查看某个元素是否存在于字典中

**Set:** 是一个**无序不重复**元素集, 基本功能包括关系测试和消除重复元素. 集合对象还支持union(联合), intersection(交), difference(差)和sysmmetric difference(对称差集)等数学运算

一个简单例子，set 可以解决列表中的重复元素 b =set(a)

1. t.add('x')            # 添加一项
3. s.update([10,37,42])  # 在s中添加多项
4. t.remove('H')
5. len(s)
6. set 的长度
7. x **in** s
8. 测试 x 是否是 s 的成员
9. s.issubset(t)
10. s <= t
11. 测试是否 s 中的每一个元素都在 t 中
12. s.issuperset(t)
13. s >= t
14. 测试是否 t 中的每一个元素都在 s 中
15. s.union(t)
16. s | t
17. 返回一个新的 set 包含 s 和 t 中的每一个元素
18. s.intersection(t)
19. s & t
20. 返回一个新的 set 包含 s 和 t 中的公共元素
21. s.difference(t)
22. s - t
23. 返回一个新的 set 包含 s 中有但是 t 中没有的元素
24. s.symmetric\_difference(t)
25. s ^ t
26. **返回一个新的 set 包含 s 和 t 中不重复的元素**

请注意：union(), intersection(), difference() 和 symmetric\_difference() 的非运算符（non-operator，就是形如 s.union()这样的）版本将会接受任何 iterable （如list[1,2,3]） 作为参数。相反，它们的运算符版本（operator based counterparts）要求参数必须是 sets (s&t)。

另外，update 可以配合运算符使用达到在原来的set上改变

1. s.update(t)
2. s |= t
3. 返回增加了 set “t”中元素后的 set “s”
5. s.intersection\_update(t)
6. s &= t
7. 返回只保留含有 set “t”中元素的 set “s”
9. s.difference\_update(t)
10. s -= t
11. 返回删除了 set “t”中含有的元素后的 set “s”
13. s.symmetric\_difference\_update(t)
14. s ^= t
15. 返回含有 set “t”或者 set “s”中有而不是两者都有的元素的 set “s”

集合是可变的，因此not hashable, 因此 a=set(),b=set(),a.add(b) -> 会报错

加上frozenset, a.add(frozenset(b)),frozenset构造函数为给丁集合创建副本。

但集合中的元素是不可变得，

**注意s.discard(item)和s.remove(item):**两者都是删除item,但是前者如果s中没有item不会报错，后者会 keyvalue异常**。**

**List:**

a 是一个list,

while a:

something you want to do with every character

a.pop()

可以遍历整个list，另一种方法遍历list:

for word in list:

do something with word

**deque (有时候比list 好用)：可以在左边pop 处值，适合queue 的数据结构。此外，有rotate()等有用的方法. https://docs.python.org/2.7/library/collections.html#collections.deque**

**from collections import deque**

appendleft(*x*), extendleft(*iterable*), popleft(),

**rotate**(*n*): Rotate the deque *n* steps to the right. If *n* is negative, rotate to the left, Rotating one step to the right is equivalent to: d.appendleft(d.pop()).

可以用这种办法实现删除和slicing功能：（删除d[n]）

**def** delete\_nth(d, n):

d.rotate(-n)

d.popleft()

d.rotate(n

**Joint method:**

str.join(sequence)

The method **join()** returns a string in which the string elements of sequence have been joined by *str* separator, . （就是用s来分开seq 字符串）

Eg: s = "-"; seq = ("a", "b", "c"); # This is sequence of strings. print s.join( seq )

a-b-c

此时seq 可以是tuple, list, 或者单个的字符串如’dqdw’,若是单个的字符串，join 方法则会自动默认将取但字符分开，即’d-q-d-w’

**Eval method:**

The eval method is to evaluate the content of the string:

Like eval(‘2+3’)->5 eval(‘2+1==4’) -> False

**List comprehensions:**

squares = [x\*\*2 **for** x **in** range(10)]

=

**>>>** squares = [] **>>> for** x **in** range(10): **...**  squares.append(x\*\*2) **...** **>>>** squares [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

contains (if):

**>>>** [(x, y) **for** x **in** [1,2,3] **for** y **in** [3,1,4] **if** x != y]

[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]

=

**>>>** combs = [] **>>> for** x **in** [1,2,3]: **...**  **for** y **in** [3,1,4]: **...**  **if** x != y: **...**  combs.append((x, y)) **...** **>>>** combs [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]

note with tuple we can.t ignore the parenthesis:

**>>>** [x, x\*\*2 **for** x **in** range(6)] File "<stdin>", line 1 [x, x\*\*2 **for** x **in** range(6)]

^ SyntaxError: invalid syntax

code below transpose the matrix:

**>>>** [[row[i] **for** row **in** matrix] **for** i **in** range(4)]

[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

**注意(x+1 for x in list) 和 [x+1 for x in list]：**

**前者是生成一个generator,需要调用.next() 依次处理其中的各个元素，后者是生成一个list, 如果我们不是一定要返回list的话，这样比较占内存。**

**这两者都可以嵌套更多的for 语句以及加上if 语句进行筛选。**

**Append**: list.append(v1) (v1 is the value we want to add)

**Extend**: a.extend(b) (将链表b加在链表a),append 只能加一个值

**Index**: a.index(v) (v: 想找索引的值)

**Count**: a.count(v) (v:想计数的值)

**Insert**: a.insert(index,value)

**Pop**: a.pop(index) (默认的是最后一个,并且此方法返回pop出来的值)

**Remov**e: a.remove(value) (此种方法没有返回值)

**Pop 和 remove 区别:**

1: pop 括号内是索引，remove 括号内是要删除的值

2: pop 返回值，remove 无返回值

reverse: a.reverse()

Sort the list: list.sort**()** (注意括号不能少), 注意sort() 是无返回值的，如：

x=[4,6,1,2]

y=x.sort() -> y 则是None

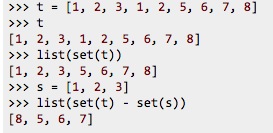
若要保留原来的列表，则需要深复制(简单的复制整个列表的方法)，y=x[:](注意此种方法仅使用与无嵌套list的情况，若要深复制嵌套的list，需要deepcopy方法)，

若简单的y=x, 则x和y都直接指向同一个列表，

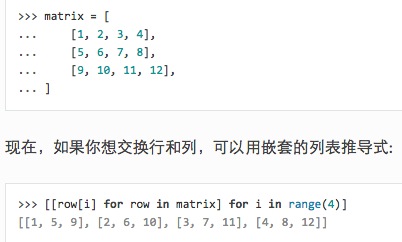
另一种方法，用sorted(), y=sorted(x), x保持不变，y是排完序后的列表

**怎样将一个列表中的重复元素删掉：**

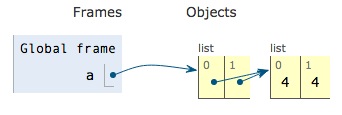
The common approach to get a unique collection of items is to use a [set](http://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset). Sets are *unordered*collections of *distinct* objects. To create a set from any iterable, you can simply pass it to the built-in [set()](http://docs.python.org/3/library/functions.html#func-set) function. If you later need a real list again, you can similarly pass the set to the [list()](http://docs.python.org/3/library/functions.html#func-list)function.

****

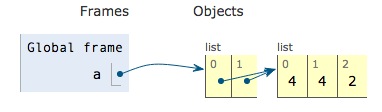
**转至一个矩阵（一行代码）**：



**关于初始化的reference 问题：**

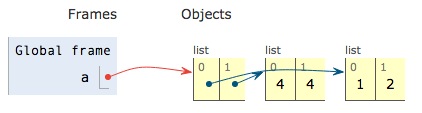
a=[[4]\*2]\*2

a[0].append(2)



注意后面的＊使得a[0] 和 a[1] sharing 一个reference了。因此我们对其中任何一个list 操作都会改变整个list, 但是如果我们改为：

a[0]=[1,2]

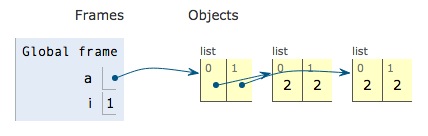


或者使用iterator:

a=[]

for i in range(2):

a.append([2]\*2)



# [Fastest way to check if a value exist in a list](http://stackoverflow.com/questions/7571635/fastest-way-to-check-if-a-value-exist-in-a-list):

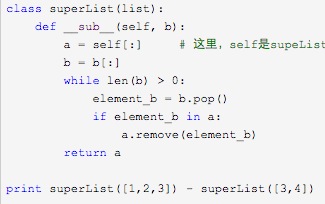
a = [4,2,3,1,5,6]

if 7 in a:

a.index(7)

# 

由于list 是一个class,我们可以继承甚至补充该class的method,例如对象的运算符重载，list 中无‘－’操作的方法（有＋的方法），我们继承list然后创建一个拥有此类的superlist():



当然我们可以重写\_\_add\_\_方法。

**class:**

private attribute(not accessiable outside of the class):双下划线加名字

example: def \_\_inaccessible(self):

其实上述方法还是可以在类外进行访问的，因为所有双下划线开始的名字都是被翻译成单下划线加名字：

s=Secretive()

s.\_Secretive\_\_inaccessible()

**constructor(构造方法)：**将init方法改为\_init\_即可：

def \_init\_(self):

self.x=32

**overwrite the constructor:** 有时候我们重新复写子类的构造方法，同时我们不想失去父类的构造方法，有两种方法：

1: 未绑定的超类构造方法，

2: 调用super 函数

example:

class Bird:

def \_int\_ (self):

self.hungry=1

class SongBird(Bird):

def \_int\_ (self):

Bird.\_int\_(self) **#method 1**

Super(SongBird,self).\_int\_() **#method 2**

Self.sound=’aaa’

**destructor(析够方法)：** \_del\_

inheritance(继承)：

class SPAMfilter(Filter):SPAMfilter 是filter 的子类

检查类的继承：issubclass(SPAMfilter,Filter)

检查一个对象属于哪个类：\_class\_ : s.\_class\_

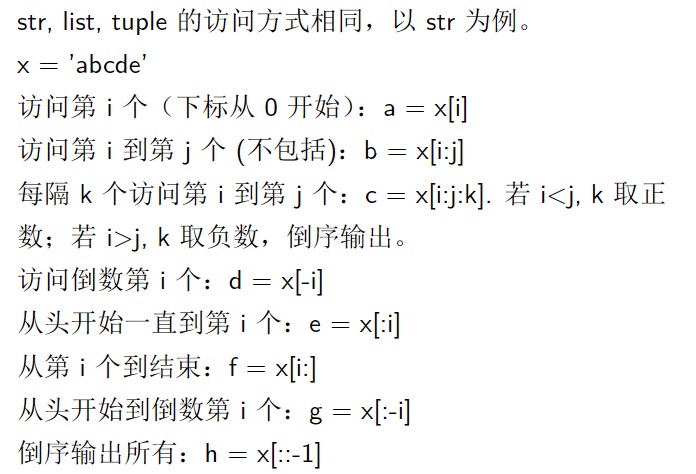
检查一个对象是否是一个类的实例：isinstance(s,SPAMfilter)

查阅已知的类的基类：\_bases\_ :SPAMfilter.\_bases\_

Multiple inheritance(多重继承):

如果一个方法从多个超类继承（有两个相同名字但不同内容的方法，先继承的类中方法会重写后继承的类中的方法）

**数据类型：**

****

**三目运算符：**

y if cond else x 相当于C 语言cond?y:x

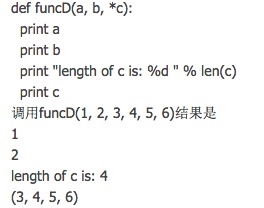
**列表推倒：** a = [x+3 for x in range(3)]

sorted(a) 直接返回列表，

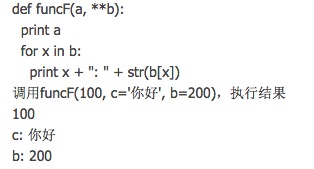
reversed(a)返回一个object,两者都可以配合迭代器使用

a.reverse() 将a 列表直接颠倒了

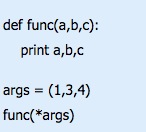
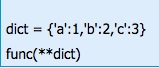
**定义参数数量未知的函数(packing)：**

\*: 代表将未知数量的**无名参数**放到tuple里面，例如：此时3，4，5，6都存在了tuple里

\*\*:所有正常行参之外的其他**关键字参数**都将存放在\*\*keyarg 传递给函数，最典型的例子是传递dict ，

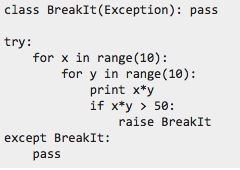
此时传递的对象b是个dict 实例。

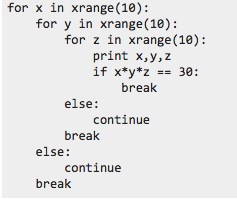
**解包裹传递（unpacking）:**

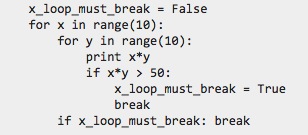
 

在定义或者调用参数时，参数的几种传递方式可以混合。但在过程中要小心前后顺序。基本原则是，先位置，再关键字，再包裹位置，再包裹关键字，并且根据上面所说的原理细细分辨。

**how to jump out of nested loops**:

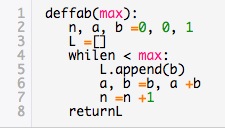
1: 

2:

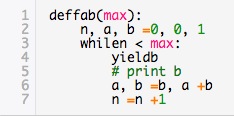
3:

**Yield-Generator:**

简单的例子，输出斐波那契数列的元素：



fab 函数通过返回 List 能满足复用性的要求，但是更有经验的开发者会指出，该函数在运行中占用的内存会随着参数 max 的增大而增大，如果要控制内存占用，最好不要用 List来保存中间结果，而是通过 iterable 对象来迭代。



　　简单地讲，yield 的作用就是把一个函数变成一个 generator，带有 yield 的函数不再是一个普通函数，Python 解释器会将其视为一个 generator，调用 fab(5) 不会执行 fab 函数，而是返回一个 iterable 对象！在 for 循环执行时，每次循环都会执行 fab 函数内部的代码，执行到 yield b 时，fab 函数就返回一个迭代值，下次迭代时，代码从 yield b 的下一条语句继续执行，而函数的本地变量看起来和上次中断执行前是完全一样的，于是函数继续执行，直到再次遇到 yield。

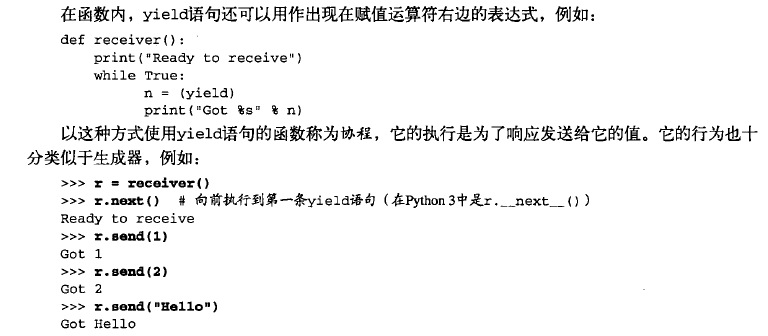
　　也可以手动调用 fab(5) 的 next() 方法（因为 fab(5) 是一个 generator 对象，该对象具有 next() 方法）

\_\_iter\_\_ 方法返回一个迭代器，一般情况下，\_\_iter\_\_ 方法会放到其他的会在for 循环中使用的对象中，这样，我们就能返回所需的迭代器了，此外，推荐是迭代器实现自己的\_\_iter\_\_ 方法，然后就能直接在for 循环中直接使用迭代器了。

生成器函数本身是一个函数，因此可以有参数，但是次函数有默认的返回值，即生成器，不能返回其他值了。

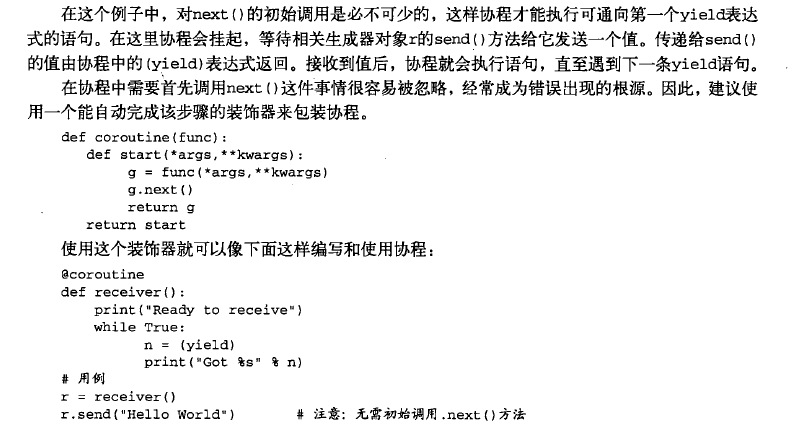
其他关于生成器的有用方法：

yield 如果出现在赋值运算符右边，则就是**协程(corrountines):**

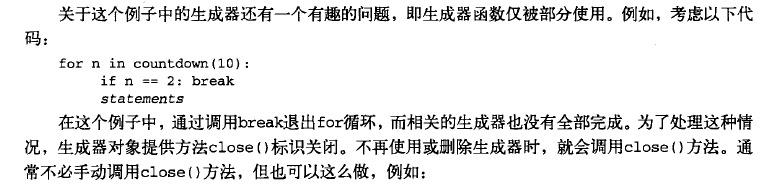
****

????**send()**: 如r.send(10),即上次生成器挂起的地方，我们可以传递10给此生成器然后在进行iteration.

上述例子需要第一次需要自己手动调动r.next()，若不想手动调动，可以写一个装饰器：



如果我们在生成器未引发StopIteration之前退出了相关程序，就会引发GeneratorExit的异常，则会调用close()方法：

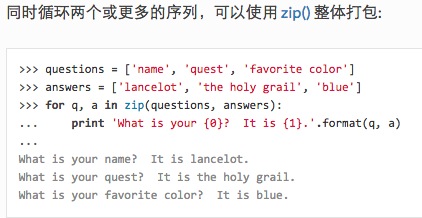


有时候我们需要索引进行一些操作，这时候**enumerate** 就派上用场了。

For idx,ele in enumerate(a):

Print idx,ele

如果我们要遍历循环多个序列，用**zip**方便：



**Notice:**

Zip 和 itertools.product(iterable,repeat) 的区别、

前者是将打包的对象一一对应，但是后者是将前者的每个对象与后者的每个对象一一对应，相当于两个for loop.

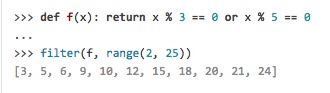
如：for x,y in itertools(range(5),range(3)):

<=> for x in range(5):

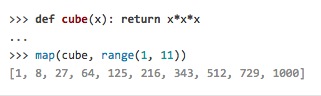
for y in range(3)

### 函数式编程工具-filter() , map(),reduce()

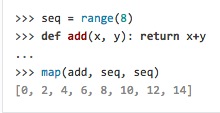
filter(function, sequence) 返回一个 sequence(序列)，包括了给定序列中所有调用 function(item) 后返回值为 true 的元素(如果可能的话，会返回相同的类型)。如果该 *序列* (sequence)是一个 [str](https://docs.python.org/2.7/library/functions.html#str),[unicode](https://docs.python.org/2.7/library/functions.html#unicode) 或者 [tuple](https://docs.python.org/2.7/library/functions.html#tuple)，返回值必定是同一类型，否则，它总是 [list](https://docs.python.org/2.7/library/functions.html#list)。例如，以下程序可以计算一个被 3 或者 5 整除的序列:



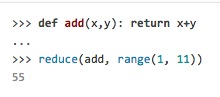
map(function, sequence) 为每一个元素依次调用 function(item) 并将返回值组成一个链表返回。例如，以下程序计算立方:



可以传入多个序列，函数也必须要有对应数量的参数，执行时会依次用各序列上对应的元素来调用函数(如果某些序列比其它的短，就用 None 来代替)。如果把 None 做为一个函数传入，则直接返回参数做为替代。

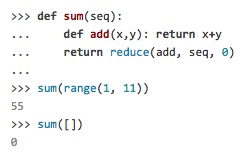


reduce(function, sequence) 返回一个**单值**，它是这样构造的：首先以序列的前两个元素调用函数function，再以返回值和第三个参数调用，依次执行下去。例如，以下程序计算 1 到 10 的整数之和:



如果序列中只有一个元素，就返回它，如果序列是空的，就抛出一个异常。

可以传入第三个参数作为初始值。如果序列是空的，就返回初始值，否则函数会先接收初始值和序列的第一个元素，然后是返回值和下一个元素，依此类推。例如:



一般list comprehension 可以替代绝大部分需要用map 和filter 的问题.

**String : Chr(),unichr() and ord():**

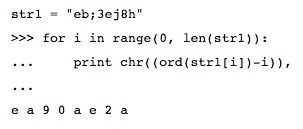
**http://blog.csdn.net/xiaoxiaoniaoer1/article/details/8542834(string 的相关操作)**

Chr(i): 输入i~[0~255],输出是相应的ASCII 码,单字节的转换。

Unichr():与chr()很类似，只是返回unicode的字符。

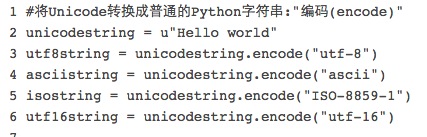
Ord(c): 输入：ascii字符，输出是对应的十进制整数.[ord(‘10’)-ord(‘0’)=10]

一般可以将chr() 和ord()连用，就可以对字符串进行相关运算的转换。如：将字符串1转换成字符串2, 使得str[i] = str [i] – i

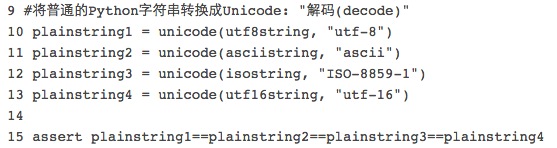


关于unicode:

Unicode 字符串可以转换成普通的python 字符串，即encode:



decode:



**python强大的处理字符的能力：**

**split()** 用法：将字符串按照我们想要的要求分开，如s.split(‘.’,3)是将字符串 s在‘.’处分离三次，s=’www.google.com.cn’,分离后：’www’,’google’,’com’,’cn’

注意**分离后是一个list**，s.split(‘.’,3)=[‘www’,’google’,’com’,’cn’],

>>> **str.center**(20)      #生成20个字符长度，str排中间，其他空格填充

对应的str.ljust(20) 和 str.rjust(20) 表示左对其和右对其，str.zfill(20) 表示右对其，左边填充0

>>>**str.upper**() #转大写

相应的str.lower()转小写，str.swapcase()大小写转换，str.captalize()首字符大写，其余的小写。

>>> **str.isalnum()**  #是否全是**字母和数字**，并至少有一个字符，

>>> **str.isdigit()**  #是否全是**数字**，并至少有一个字符

>>> **str.isalpha()**  #是否全是字母，并至少有一个字符

>>> **str.islower()**   #是否全是小写，当全是小写和数字一起时候，也判断为True ，类似的str.isuper()

类似的 str.isspace() 是否全是空格，并至少一个字符，str.istitle() 是否所有单词首字母都是大写，

>>> **str.startswith('str')** #判断字符串以'str'开头，类似的str.endswith(‘arn’)判断’arn’结尾.

>>> **str.find('a')**      #查找字符串，没有则返回-1，有则返回查到到第一个匹配的索引,str.rfind(‘a’)类似，有则返回查到的最后一次匹配的。

类似的str.index(‘a’),如果没有匹配则报错，有就返回第一次匹配的的index, str.rindex(‘a’)同上

>>> **str.replace('EAR','ear')**  #匹配替换，

>>>**str.replace('n','N',1)，**将’n’用’N’代替一次

>>> **str.strip('n')**   #删除字符串首尾匹配的字符，通常用于默认删除回车符,类似的有lstrip(‘n’) 和rstrip(‘n’)

>>> **str.expandtabs()**  #把制表符转为空格, str.expandtabs(2)指定空格数

>>>**'-'.join(str)**，例如：str='Learn string' ，之后：'L-e-a-r-n- -s-t-r-i-n-g'

如果是一个列表字符，如，l1=['Learn','string'] ，>>> '-'.join(l1)

'Learn-string'

>>> **str.split('n')** ，str='Learn string'，['Lear', ' stri', 'g'],默认的是‘’(空格)，str.split(‘n’,1),顺序分离一次，类似的还有str.rsplit(‘n’,1)倒叙分离一次.

>>>**str.partition('n')**,与split 类似，但保留‘n’,即('Lear', 'n', ' string')，

split 和partition **区别**：

1:前者不包括分离符，后者包括

1:前者产生一个list,后者产生tuple,

2:前者可以分离多次，后者只能一次。

c=’123456’,c[::1]=’123456’,c[::2]=’135’,c[::3]=’14’,c[::-1]=’654321’,c[::-2]=’642’ –此种方法可以适用于将单个数字符转换成响应的数字.

**字符相加：**

‘2’ +’3’ =’23’ ，我们可以用这种方法将character array 转换成 string:

a=[‘2’, ‘3’,’4’]

b=’’

for i in a:

b+=i

另一种方法: b=’’.join(a)

**Prefix and postfix:**

a++: meaning the value of a will get change after the evaluation of expression.

++a: meaning the value of a will get change before the evaluation of expression.

a = 4; b = a++; // first b will be 4, and after this a will be 5 // now a value is 5 c = ++a; // first a will be 6, then b will be assigned to 6

**if a :**

if a 等价于if a is not zero or if a is not None

if not a 等价于 if a is zero or if a is None

**shallow copy and deep copy:**

当你对一个对象赋值的时候(做为参数传递,或者做为返回值),Python和Java一样,总是传递原始对象的引用,而不是一个副本,

shallow copy:

(1)、使用切片[:]操作进行拷贝

(2)、使用工厂函数(如list/dir/set)等进行拷贝

(3)、copy.copy()

>>> jack = ['jack',['age',20]]

>>> tom = jack[:]

>>> anny = list(jack)

>>> jack

['jack', ['age', 20]]

>>> tom

['jack', ['age', 20]]

>>> anny

['jack', ['age', 20]]

>>> print id(jack),id(tom),id(anny)

13457088 18487376 18489136

使用copy.copy(), 进行浅复制，如果我们复制的只是单纯的列表（即没有内嵌的列表），浅复制后的对象修改后不会影响愿对象。如：

a=[1,2,3]

b=copy.copy(a)

b.append(4)

结果： a=[1,2,3], b=[1,2,3,4]

如果希望复制一个容器对象,以及它里面的所有元素(包含元素的子元素),使用copy.deepcopy,这个方法会消耗一些时间和空间,不过,如果你需要完全复制,这是唯一的方法.

deepLis = copy.deepcopy(lis)

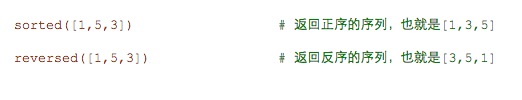
deepLis[1].append('foo')

print deepLis

#[['a'], [1, 2,'foo'], ['z', 23]]

print lis

#[['a'], [1, 2], ['z', 23]]

**序列操作：** 

**匿名函数 lambda:**

lambda 语句中，冒号前面是函数的参数，用逗号隔开，冒号右边是返回值，

g=lambda x,y: x+y

可以将lambda用在filter,map 和reduce 函数中，如：

a=[1,2,3,4,5]

b=filter(lambda x: x%2==0,a) ->b=[2,4]

但是filter 等这些方法返回的都是列表，我们可以同样在iterator 中利用这些功能，如：

itertools.ifilter(lambda x: x%2==0,range(10)) [range(10)可以改为xrange(10),又或者是［1，3，5］列表]

**if \_\_name\_\_ == '\_\_main\_\_':**

相当于此文件的测试代码，当我们直接运行该程序，此段是main,当我们import 这段程序时，此段无用。

**一些小技巧：**

**关于*import*几点：**

import Testlib as t

from Testlib import lib\_func #只引用lib\_func，减少模块占有的内存

from Testlib import \* #调用Testlib所有对象，并跳过Testlib引用字段。更确切的说，dir(模块名称) 返回对象的所有特性（所有的函数，类，变量）,其中有个叫\_\_all\_\_的函数，这样 \* 就可以直接调用并且跳过testlib引用字段了。

**Help** (copy.copy):返回相关的注释。

查询某个函数的的输入参数:在inspect模块，调用

**inspect.getargspec(func)**

**查询对象的类和类名称：**

**a.\_\_class\_\_**

**a.\_\_class\_\_.\_\_name\_\_**

**查询父类：**

**list.\_\_base\_\_**

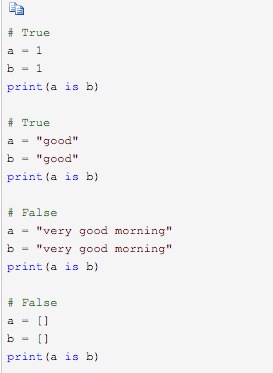
**dir():可以查询对象的所有特殊方法。**

**关于super的使用：super(子类的名称，self).调用父类的方法**

**例如：super(singbird,self).\_\_init\_\_()**

**对象的内存：**

**数字和短字符串(没有空格)每个对象只有一个，但长字符串和其他对象可以有多个对象。**

****

可以用sys 包里面的getrefcount() 查询对象的引用次数,但是由于getrefcount()本身也引用了对象，实际上应该减去1.

**分代回收：**Python同时采用了分代(generation)回收的策略。这一策略的基本假设是，存活时间越久的对象，越不可能在后面的程序中变成垃圾。我们的程序往往会产生大量的对象，许多对象很快产生和消失，但也有一些对象长期被使用。出于信任和效率，对于这样一些“长寿”对象，我们相信它们的用处，所以减少在垃圾回收中扫描它们的频率

Python将所有的对象分为0，1，2三代。所有的新建对象都是0代对象。当某一代对象经历过垃圾回收，依然存活，那么它就被归入下一代对象。垃圾回收启动时，一定会扫描所有的0代对象。如果0代经过一定次数垃圾回收，那么就启动对0代和1代的扫描清理。当1代也经历了一定次数的垃圾回收后，那么会启动对0，1，2，即对所有对象进行扫描。

这两个次数即上面get\_threshold()返回的(700, 10, 10)返回的两个10。也就是说，每10次0代垃圾回收，会配合1次1代的垃圾回收；而每10次1代的垃圾回收，才会有1次的2代垃圾回收。

同样可以用set\_threshold()来调整，比如对2代对象进行更频繁的扫描。

**import gc**

**gc.set\_threshold(700, 10, 5)**

**十进制和二进制的呼唤：**

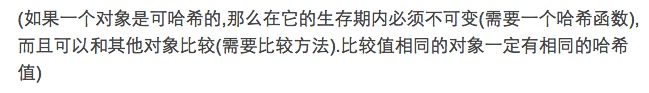
一般的bin(int) 会返回一个含有头两个字符为’0b’的字符串，但是如果我们不特意规定返回补码，负数的二进制表达将会是‘-0b（正数的表达）’，如-5:bin(-5) -> ‘-0b101’,如果我们想要相应的32 位数的步码表示，

bin(-5 % (1<<32)) -> '0b11111111111111111111111111111011'

或者 bin(-5 & 0xffffffff)

但上述方法对正数不工作，即bin(5 % (1<<32)) -> ‘0b101’

**关于hashable 和 hashable object:**

An object is hashable if it has a hash value which never changes during its lifetime (it needs a \_\_hash\_\_() method), and can be compared to other objects (it needs an \_\_eq\_\_() or \_\_cmp\_\_() method). Hashable objects which compare equal must have the same hash value.

简单的说，mutable的对象都**不是**hashable的，因为他们可以改变，而在hash 函数中，如果ID一样，hash值必须一样，这样的对象有list,dict,set.

另外：

Objects which are instances of user-defined classes are hashable by default; they all compare unequal, and their hash value is their id()

**文件和流：**

f=open(namem,[mode],[buffering])

其中model 是包括’r’,(读)’w’(写)’a’(追加)’b’(二进制)’r+’(读写模式)

buffering 如果是0或者false,即是无缓冲的，直接针对硬盘.

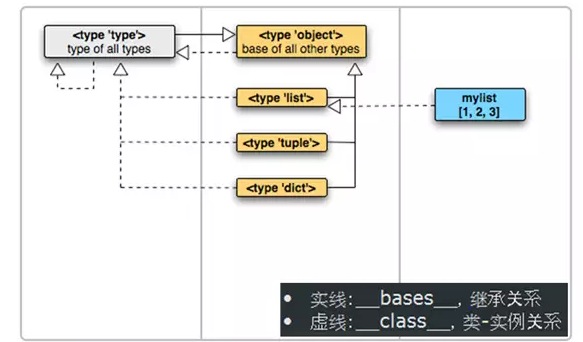
1:有缓存 ，大于1的数表示缓存区字节大小，-1代表默认的缓存大小。

注意当我们写入文件，我们需要flush或者close来更新硬盘上的数据。

注意我们打开文件后，一般都要f.close().(with 可以自动帮助我们关闭文件。With open(‘filename’)as f:)

还有f.readline() 和f.readlines()

**关于python中对象的基本概念：**

****

如果一个对象是<type ‘type’>的实例， 那它就是Type Object（类型对象）， 否则是Non-type Object（非类型对象）

通过<type ‘type’>生成的对象（类型对象）

Type Object（类型对象）有两个重要的特性：

可以被子类化（subclassed）

可以被实例化（instantiated）

**<type ‘type’>**：对所有的类型对象进行type()或者获取”\_\_class\_\_”属性都将得到 <type ‘type’>。比如type(list) -><type 'type'>

**<type ‘object’>:**是一个类型对象（因为type(object)是<type ‘type’>）；同时，<type ‘object’>是所有类型对象（除去<type ‘object’>本身）的父类

**方法（method）：首先，方法是在类定义中定义的函数，有三种－instance\_method(实例方法),class\_method(类方法)，static—method（静态方法）**

1:其中实例方法将指定类的实例操作，第一个参数一般是self。

**绑定方法(bound method)**:将一个函数和一个相关实例进行了绑定，如，f=foo() mech=f.instance\_method mech(37)

**非绑定方法(unbounded method):**封装了方法函数的的可调用对象，需要一个实例作为第一个参数如：

umech = foo.instance\_method umech(f,37)

2:类方法将类本身当作一个对象操作，第一个参数是cls

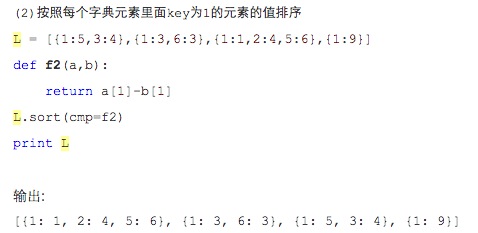
3:静态方法为打包在类中的函数。

1. **sort(cmp,key,reverse):、**

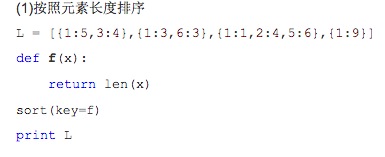
cmp 接受一个函数，

默认的cmp和key都是none, reverse 是1.

**Cmp** 决定比较的函数，



**key** 决定选取列表中的那个值惊醒排序，比如，列表中元素都是tuple,我们可以规定按照那个值进行排序，



或者根据每个值与其他值的关系，例如我们按照与10的偏差大小排序：

1. sort(key=lambda x: abs(x-10))

**reverse** 设为1时代表倒叙排列

sorted 也可以使用上述方法，不过第一个参数必须为要排序的本体，并且不是基于本体的排序，即不会改变原序列。

其实如果要调用的时对象的属性时侯，可以使用**operator** 模块，如下：



关于super 的调用：

super().\_\_init\_\_(frob, 34) #调用了父类的\_\_init\_\_(),注意括号不能丢super().

**特殊方法（Magic Method）**

**对象创建与销毁：** \_\_new\_\_(cls,[,\*args[,\*\*kwargs]]), \_\_init\_\_(self, [,\*args[,\*\*kwargs]]) , \_\_del\_\_(self)

**对象的比较与排序**：

\_\_lt\_\_(self,other) -> self < other

\_\_le\_\_(self,other) -> self <= other

\_\_gt\_\_(self,other) -> “>”

\_\_ge\_\_(self,other) -> “>=”

\_\_eq\_\_(self,other) -> “==”

\_\_ne\_\_(self,other)-> “!=”

**序列与映射方法：**

a.\_\_len\_\_() -> len(a)

a.\_\_getitem\_\_(2) -> a[2]

a.\_\_setitem\_\_(7) -> a[1]=7

a\_\_delitem\_\_(2) -> del a[2]

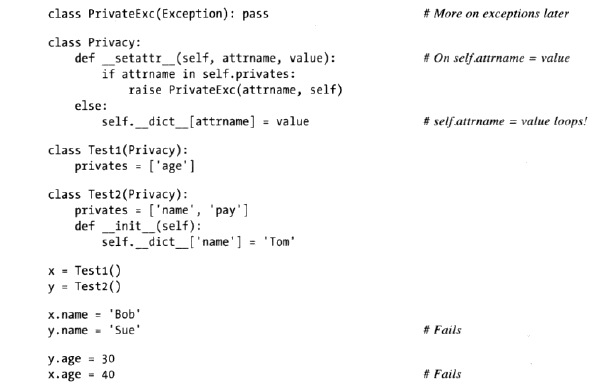
a.\_\_contains\_\_(5) -> 5 in a

\_\_getattr\_\_ and \_\_setattr\_\_:用于拦截属性点号运算符



**注意：**\_\_getattribute\_\_拦截所有的属性获取，而不是那些未定义的属性。

关于instance attribute 是私有的情况模拟：



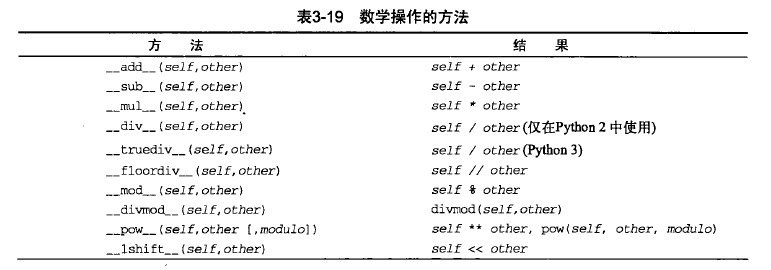
**关于切片的强大功能：**

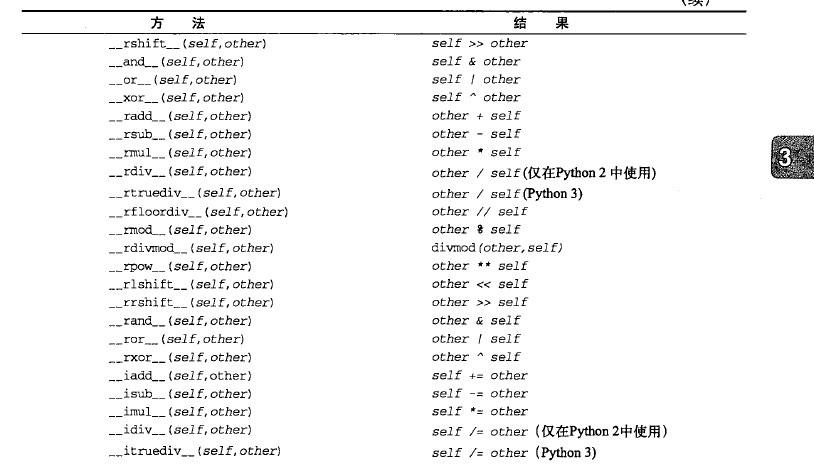
多维的切片表达： b=m[1:10,3:20]

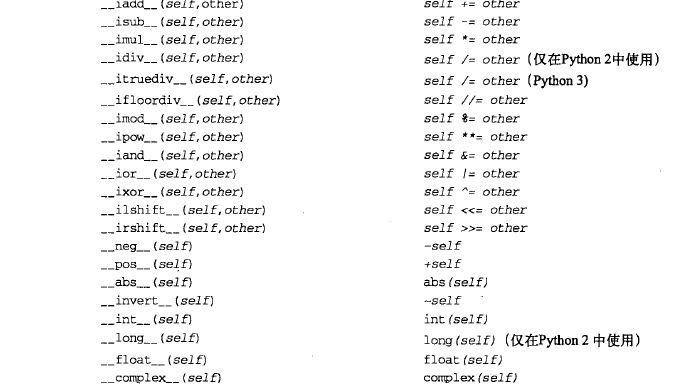
c=m[1:10:2,5:30:5]

a=m[…,10:20] (省略号表示任意的维数)

**数学操作：**

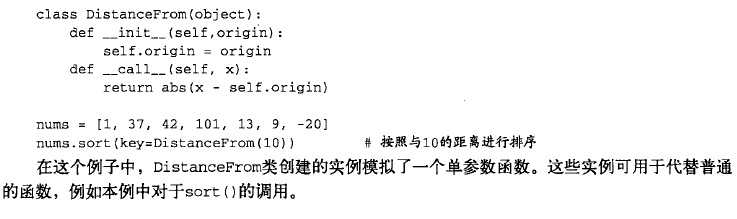
****





**可调用接口：**

x.\_\_call\_\_(self,arg1,arg2…) -> x(arg1,arg2…)



复写\_\_call\_\_后，先实例化对象，再从该对象直接调用，换句话说，我们可以把这个类型的对象当作函数来使用，相当于重载了括号运算符

如：

a=Solution(),

a(2,3..) (可以直接调用)，类似于a.\_\_call\_\_(self,2,3..)

**‘==’ 和 ’is ’ 的区别：**

**‘＝＝’是检验值是否相同，但’is ’是判断两者的引用对象是否一样，换言之，检验两者的id 是否一样。例如：**

x=[1,2],y=[1,2]

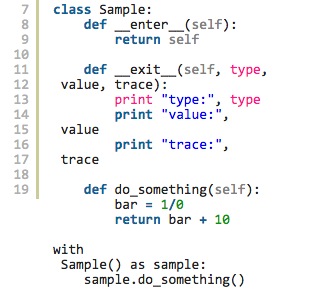
x==y -> True

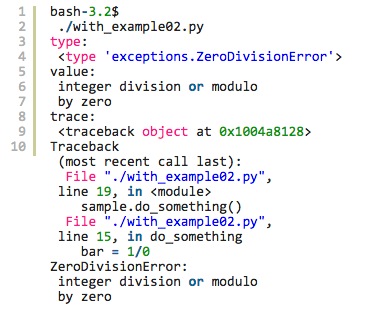
x is y -> False

但是若上面的x,y是num 或者str,则x,yID一样，因为num和str都是immutable

**‘ With’ 的相关用法 --- Context Management：**

with obj as f: 执行with obj 语句时，它执行obj.\_\_enter\_\_()来表示进入了一段上下文，并且将obj.\_\_enter\_\_()的返回值保存在f 中（之一obj不一定给f 赋值）。当控制流离开上下文时，调用obj.\_\_exit\_\_(type,value,traceback),若没有异常，则三个参数都是none,否则这三个参数将保存异常的类型，值以及跟踪信息。\_\_exit\_\_()返回true 或者false, 若为false,则引发的异常将被传递出上下文。





在with后面的代码块抛出任何异常时，\_\_exit\_\_()方法被执行。正如例子所示，异常抛出时，与之关联的type，value和stack trace传给\_\_exit\_\_()方法，因此抛出的ZeroDivisionError异常被打印出来了。开发库时，清理资源，关闭文件等等操作，都可以放在\_\_exit\_\_方法当中。

如果要处理多个项：

with open("x.txt") as f1, open('xxx.txt') as f2:

    do something with f1,f2

**闭包(closure)和装饰器(decorator):**

要了解闭包，首先要了解函数的作用域范围，一般位于嵌套内的函数是不能修改嵌套外函数的值，在python 3.x 中可以在嵌套内函数中nonlocal n代表n是区域外的变量，[global n d代表是全局变量]但在2.x 的版本中，只能通过列表或者字典来代表要修改外部的值，例如：

**def line\_conf(a,b):**

**# v={}#use dict to store**

**# v['0']=a\*b**

**v=[0]**

**v[0]=a\*b**

**def line(x):**

**# nonlocal i #only can be used in python 3 which means this variable is not in this local function**

**# v['0']=v['0']+x**

**v[0]=v[0]+x**

**return v[0]\*x+b**

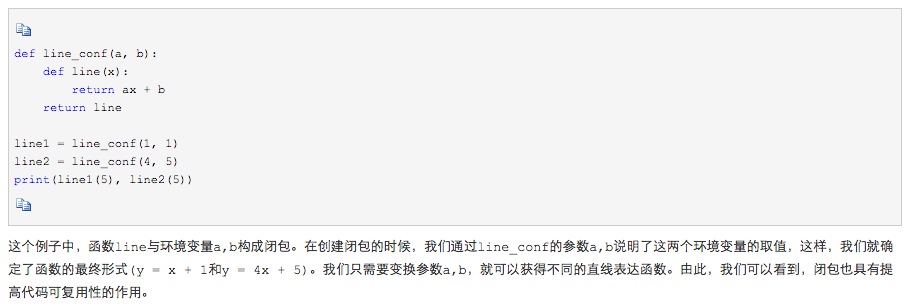
**return line**

**l1=line\_conf(4,5)**

**l1(5) -> 结果是130**

上述例子也是closure的例子，一般的下面的两个例子更能说明闭包的特性：





一个函数和它的环境变量合在一起，就构成了一个闭包(closure)。在Python中，所谓的闭包是一个包含有环境变量取值的函数对象。环境变量取值被保存在函数对象的\_\_closure\_\_属性中。

关于lazy evaluation 和闭包提高有关运行速度的例子，see python reference p80~p81.

**装饰器：闭包的一个特例，用来包装另一个函数或者类，目的是透明地修改被包装对象地行为。注意，其本身也是一个函数。**

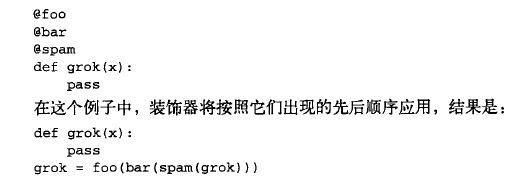
**例如：**

@trace def square(x):

def square(x): => return x\*x

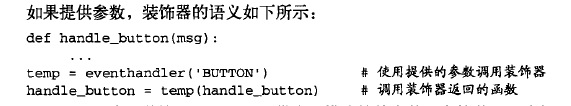
return x\*x square=trace(square)

如果有多个语法糖，装饰器按顺序应用:



若装饰器有参数：





同样的，若装饰一个类，要确保装饰器返回了类对象

**查看当行命令的运行时间：**

from timeit import timeit

timeit(“s=dict(one=2)”)

**一些常见地错误:**

1: 将以一个可变的用作默认值，

|  |  |  |  |
| --- | --- | --- | --- |
| 1  2  3 | def foo(numbers=[]):      numbers.append(9)  print numbers   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8 | >>> foo() # first time, like before  [9]  >>> foo() # second time  [9, 9]  >>> foo() # third time...  [9, 9, 9]  >>> foo() # WHAT IS THIS BLACK MAGIC?!  [9, 9, 9, 9] |   我们发现每次foo()无参数传入时，都会在numbers 里面加9，与我们的预期不一样。为什么在调用函数的时候这个默认值却被赋予了不同的值？因为在你每次给函数指定一个默认值的时候，Python都会存储这个值。如果在调用函数的时候重写了默认值，那么这个存储的值就不会被使用。当你不重写默认值的时候，那么Python就会让默认值引用存储的值（这个例子里的numbers）。  要摆脱这样的行为，在函数开始的地方用切片或者方法来创建默认参数的副本，或者将默认值的表达式移到函数里面；只要每次函数调用时这些值在函数里，就会每次都得到一个新的对象：  如果改成整形，或者其他不可变的对象，我们就不会遇到这种问题。  2: 在except语句中捕捉多个异常的正确做法是将所有想要捕捉的异常放在一个**元组**（[tuple](https://docs.python.org/2/tutorial/datastructures.html" \l "tut-tuples" \t "_blank)）里并作为第一个参数给except语句。并且，为移植性考虑，使用as关键字  except (ValueError, IndexError) as e:  关于flask 的笔记： |