大数据系统学习：十年Python大牛总结出的python基础知识实例（详解）

1、在Python 语言中，对象是通过引用传递的。

```
在赋值时，不管这个对象是新创建的，
还是一个已经存在的，都是将该对象的引用（并不是值）赋值给变量。
   如：x=1    1这个整形对象被创建，然后将这个对象的引用赋值给x这个变量
```

2、多元赋值，其实就是元组赋值

```
x,y,z=1,2,'string'    等价于    (x,y,z)=(1,2,'string')

#利用多元赋值实现的两个变量的值交换

>>> x, y = 1, 2
>>> x
1
>>> y
2
>>> x, y = y, x
>>> x
2
>>> y
1
```

3、编写模块

```python
# 1. 起始行
# -*- coding: cp936 -*-

# 2. 模块文档
"""This is a test module again"""

# 3. 模块导入
import sys
import os

# 4. 变量定义
debug=True

# 5. 类定义语句
class FooClass(object):
    """FooClass"""
    flag=1
    def foo(self):
        print "FooClass.foo() is invoked, "+str(self.flag)

# 6. 函数定义语句
def test():
    """test function"""
    foo=FooClass()

    if debug:
        print 'ran test()'
        foo.foo()
        print foo.flag

# 7. 主程序
if __name__ == '__main__':
    test()
    print "what are these?"
```

4、时刻记住一个事实

```
那就是所有的模块都有能力来执行代码，最高级别的Python 语句——也就是说。
那些没有缩进的代码行在模块被导入时就会执行， 不管是不是真的需要执行。
由于有这样一个“特性”，比较安全的写代码的方式就是除了那些真正需要执行
的代码以外， 几乎所有的功能代码都在函数当中。再说一遍， 通常只有主程
序模块中有大量的顶级可执行代码，所有其它被导入的模块只应该有很少的顶
级执行代码，所有的功能代码都应该封装在函数或类当中。
```

5、动态类型

```
变量赋值时，解释器会根据语法和右侧的操作数来决定新对象的类型。
在对象创建后，一个该对象的引用会被赋值给左侧的变量。
```

6、变量在内存中是通过引用计数来跟踪管理的

```
一个对象增加新的引用：对象被创建、对象的别名被创建、作为参数传递给函数、
方法或类、成为容器对象中的一个元素.
一个对象减少引用：变量赋值给另外一个对象、del显示删除一个变量、引用离开
了它的作用范围、对象被从一个窗口对象中移除、窗口对象本身被销毁 .
```

7、异常处理

```
#try-except-else语句，else 子句在try 代码块运行无误时执行
#异常处理最适用的场合，是在没有合适的函数处理异常状况的时候
try:
    fobj=open(fname,'r')
except IOError,e:
    print "file open error: ",e
else:
    for eachLine in fobj:
        print eachLine,
    fobj.close()
```

8、所有的Python 对像都拥有三个特性

```
    . 身份
    . 类型
    . 值
这三个特性在对象创建的时候就被赋值，除了值之外，其它两个特性都是只读的  。
```

9、布尔值

```
每个对象天生具有布尔 True 或 False 值。空对象、值为零的任何数字或者Null对象 None 的布尔值都是False
```

10、对象身份比较

```
>>> x=1.0
>>> y=1.0
>>> x is y
False
>>> x is not y
True
>>> id(x)
19094432
>>> id(y)
19094416
#比较两个变量是否指向同一个对象，但是整数和字符串有缓存机制，有可能指向同一个对象.
```

11、cmp()

```
内建函数cmp()用于比较两个对象obj1 和obj2， 如果obj1 小于obj2，则返回一个负整
数，如果obj1 大于obj2 则返回一个正整数， 如果obj1 等于obj2， 则返回0。它的行为非常
类似于C 语言的strcmp()函数。比较是在对象之间进行的，不管是标准类型对象还是用户自定
义对象。如果是用户自定义对象， cmp()会调用该类的特殊方法__cmp__()。
```

12、str()和repr()

```
str()函数得到的字符串可读性好， 而repr()函数得到的字符串通常可以用来重新获得该对象，
通常情况下 obj == eval(repr(obj)) 这个等式是成立的。
str()得到的字符串对人比较友好，而repr()得到的字符串对python比较友好
```

13、isinstance()和type()，主要体现的是代码的优化

```python
from types import *

def displayNumType0(num) :
    print num,'is',
    if type(num) is IntType :
        print 'an integer'
    elif type(num) is LongType :
        print 'a long'
    elif type(num) is FloatType :
        print 'a float'
    elif type(num) is ComplexType :
        print 'a complex'
    else :
        print 'not a number at all !!!'


def displayNumType1(num):
    print num, 'is',
    if(isinstance(num,(int,long,float,complex))):
        print 'a number of type: ',type(num).__name__
    else:
        print 'not a number at all !!!'
```

14、标准类型的分类

```
(1) 存储类型
   标量/原子类型： 数值（所有的数值类型），字符串（全部是文字）
   容器类型： 列表、元组、字典
(2) 更新类型
   可变类型： 列表， 字典
   不可变类型： 数字、字符串、元组
(3) 访问模型
根据访问我们存储的数据的方式对数据类型进行分类。在访问模型中共有三种访问方式：直接存取，顺序，和映射。
   直接访问： 数字
   顺序访问： 字符串、列表、元组
   映射访问： 字典


映射类型类似序列的索引属性，不过它的索引并不使用顺序的数字偏移量取值， 它的元素无序存放，
通过一个唯一的key 来访问， 这就是映射类型， 它容纳的是哈希键-值对的集合。


汇总：

数据类型      存储模型      更新模型      访问模型
数字          Scalar       不可更改      直接访问
字符串        Scalar       不可更改      顺序访问
列表          Container      可更改      顺序访问
元组          Container     不可更改      顺序访问
字典          Container      可更改      映射访问
```

15、不同数据类型之间的运算

```
在运算之前，要将两个操作数转换为同一数据类型，数字强制类型
转换原则是整数转换为浮点数， 非复数转换为复数。
```

16、python除法：

```
#（1）传统除法，若操作数是整数，则进行取整操作，若操作数是浮点数，则执行真正的除法
>>> 1/2
0
>>> 1.0/2
0.5

#（2）真正的除法，未来的除法，不管操作数是什么类型，都要进行真正的除法运算
>>> from __future__ import division
>>> 1/2
0.5
>>> 1.0/2.0
0.5

#（3）地板除，不管操作数是什么数据类型，都进行取整操作
>>> 1//2
0
>>> 1.0//2
0.0
```

17、工厂函数

```
#工厂函数就是指这些内建函数都是类对象， 当你调用它们时，实际上是创建了一个类实例，有以下这些工厂函数：
int(), long(), float(), complex(), bool()

>>> int('F',16)
15
>>> int('15')
15
```

18、内建函数

（1）适用于所有数据类型的内建函数：

```
abs(num)  #返回 num 的绝对值

coerce(num1, num2)  #将num1和num2转换为同一类型，然后以一个 元组的形式返回。

divmod(num1, num2)  #除法一取余运算的结合。返回一个元组(num1/num2,num1 %num2)。
#对浮点数和复数的商进行下舍入（复数仅取实数部分的商）

pow(num1, num2, mod=1)  #取 num1 的 num2次方，如果提供 mod参数，
#则计算结果再对mod进行取余运算

round(flt, ndig=0)  #接受一个浮点数 flt 并对其四舍五入，保存 ndig位小数。
#若不提供ndig 参数，则默认小数点后0位。
```

（2）适用于整数的内建函数：

```
hex(num) #将数字转换成十六进制数并以字符串形式返回

oct(num) #将数字转换成八进制数并以字符串形式返回

chr(num) #将ASCII值的数字转换成ASCII字符，范围只能是0 <= num <= 255。

ord(chr) #接受一个 ASCII 或 Unicode 字符（长度为1的字符串），
#返回相应的ASCII或Unicode 值。

unichr(num) #接受Unicode码值，返回 其对应的Unicode字符。
#所接受的码值范围依赖于你的Python是构建于UCS-2还是UCS-4。
```

19、布尔值

```
#对于值为零的任何数字或空集（空列表、空元组和空字典等）在Python 中的布尔值都是False。
>>> bool('1')
True
>>> bool('0')
True
>>> bool('fdsafds')
True
>>> bool('')
False
>>> bool([])
False
>>> bool([3])
True
>>> bool(1)
True
>>> bool(0)
False
```

20、数字类型相关模块

```
decimal:  #十进制浮点运算类 Decimal
array:  #高效数值数组（字符，整数，浮点数等等）
math/cmath:  #标准C库数学运算函数。常规数学运算在match模块，复数运算在cmath模块
operator:  #数字运算符的函数实现。比如 tor.sub(m,n)等价于 m - n
random:  #多种伪随机数生成器
```

21、随机数，要导入random模块

```
randrange():  #它接受和 range() 函数一样的参数， 随机
              #返回range([start,]stop[,step])结果的一项
uniform():    #几乎和 randint()一样，不过它返回的是二者
              #之间的一个浮点数(不包括范围上限)。
random():     #类似 uniform() 只不过下限恒等于0.0，上限恒等于1.0
choice():     #随机返回给定序列（关于序列，见第六章）的一个元素
```

示例：

```
>>> import random
>>> for i in range(5):
    print random.randrange(0,100),

1 58 77 38 3

>>> for i in range(5):
    print random.uniform(0,10),

7.26383692825 2.76070616182 7.37142561958 7.8026850248 4.7771524698

>>> for i in range(5):
    print random.random(),

0.232856863437 0.0784714114799 0.238714810514 0.0698641200863 0.386250556331

>>> for i in range(5):
    print random.choice([1,2,3,'4','5','suo']),

3 5 1 suo 4
```

22、成员关系操作符 (in, notin)

```
成员关系操作符使用来判断一个元素是否属于一个序列的，返回值为True或False
```

23、len()函数，可以得到序列长度

24、访问序列中的元素可以使用负索引

```
范围是 -1 到序列的负长度，-len(sequence),
 -len(sequence) <= index <= -1.正负索引的区别在于正索引以
序列的开始为起点，负索引以序列的结束为起点
```

25、序列切片操作

```
>>> s = 'abcdefgh'
>>> s[0:4]
'abcd'
>>> s[1:5]
'bcde'
>>> s[::-1] # 可以视作"翻转"操作
'hgfedcba'
>>> s[::2] # 隔一个取一个的操作
'aceg'

>>> s = 'abcde'
>>> for i in [None] + range(-1, -len(s), -1):
... print s[:i]
...
abcde
abcd
abc
ab
a
```

26、字符串、序列、元组类型转换

```
list(iter):     #把可迭代对象转换为列表
str(obj):       #把obj 对象转换成字符串(对象的字符串表示法)
unicode(obj):   #把对象转换成Unicode 字符串(使用默认编码)，是str()函数的unicode版本
basestring():   #抽象工厂函数,其作用仅仅是为str 和unicode 函数提供父类，所以不能被
                #实例化，也不能被调用(详见第6.2 节)
tuple(iter):    #把一个可迭代对象转换成一个元组对象
```

```
一旦一个Python 的对象被建立，我们就不能更改其身份或类型了.如果你把一个列表
对象传给list（）函数，便会创建这个对象的一个浅拷贝，然后将其插入新的列表中。
所谓浅拷贝就是只拷贝了对对象的索引，而不是重新建立了一个对象！如果你想完全的
拷贝一个对象(包括递归，如果你的对象是一个包含在容器中的容器),你需要用到深拷贝。
```

27、序列类型可用的内建函数

```
enumerate(iter):  #接受一个可迭代对象作为参数，返回一个enumerate
                  #对象(同时也是一个迭代器)，该对象生成由iter 每
                  #个元素的index 值和item 值组成的元组(PEP 279)

len(seq):         #返回seq 的长度

max(iter,key=None)
max(arg0,arg1.key=None):  #返回iter 或(arg0,arg1,...)中的最大值
#如果指定了key,这个key 必须是一个可以传给sort()方法的,用于比较的回调函数.

min(iter, key=None)
min(arg0, arg1. key=None):  #返回iter 里面的最小值;或者返回(arg0,arg2,...)
#里面的最小值;如果指定了key,这个key 必须是一个可以传给sort()方法的,用于比较的回调函数.

reversed(seq):  #接受一个序列作为参数,返回一个以逆序访问的迭代器(PEP 322)

sorted(iter,
func=None,
key=None,
reverse=False):  #接受一个可迭代对象作为参数，返回一个有序的列表;可选参数func,
#key 和reverse 的含义跟list.sort()内建函数的参数含义一样.

sum(seq, init=0):  #返回seq 和可选参数init 的总和，效果等同于reduce(operator.add,seq,init)

zip([it0, it1,... itN]):  #返回一个列表，其第一个元素是it0,it1,...这
                          #些元素的第一个元素组成的一个元组，第二个...,类推.
```

28、字符串

```
Python 里面单引号和双引号的作用是相同的，Python 里面没有字符这个类型
,而是用长度为1 的字符串来表示这个概念，字符串是不可变的,所以你不能仅
仅删除一个字符串里的某个字符,你能做的是清空一个空字符串，或者是把剔
除了不需要的部分后的字符串组合起来形成一个新串。
```

29、字符串模块

```
>>> import string
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.atol('10')
10L

虽然对初学者来说string 模块的方式更便于理解，但出于性能方面的考虑，
我们还是建议你不要用string 模块。原因是Python 必须为每一个参加连接
操作的字符串分配新的内存，包括新产生的字符串。
```

30、字符串格式化

```
Python 支持两种格式的输入参数。第一种是元组,这基本上是一种的
Cprintf()风格的转换参数集;

Python 支持的第二种形式是字典形式.字典其实是一个哈希键-值对的
集合。这种形式里面，key 是作为格式字符串出现，相对应的value 值
作为参数在进行转化时提供给格式字符串.
>>> dic={'key1':'suo','key2':'dai','key3':80}
>>> '%(key1)s love %(key2)s %(key3)d years' % dic
'suo love dai 80 years'

格式字符串既可以跟print 语句一起用来向终端用户输出数据
，又可以用来合并字符串形成新字符串，而且还可以直接显示
到GUI(Graphical User Interface)界面上去.
```

31、原始字符串

```
#在原始字符串里，所有的字符都是直接按照字面的意思来使用，没有转义特殊或不能打印的字符
>>> import re
>>> m=re.search(r'\\[rtfvn]',r'Hello World!\n')
>>> if m is not None:
    m.group()

'\\n'
```

32、Unicode 字符串操作符(u/U)

```
它用来把标准字符串或者是包含Unicode 字符的字符串转换成完全地Unicode 字符串对象。

```

33、字符串内建函数

- 各种内建函数，参见API

34、字符串三引号

```
它允许一个字符串跨多行，字符串中可以包含换行符、制表符以及其他
特殊字符.三引号让程序员从引号和特殊字符串的泥潭里面解脱出来，
自始至终保持一小块字符串的格式是所谓的WYSIWYG(所见即所得)格式的。

>>> hi='''''''hi \n are you'''
>>> hi
'hi \n are you'
>>> print hi
hi
 are you
>>>
```

35、字符串不可变性

```
字符串是一种不可变数据类型，就是说它的值是不能被改变或修改的。
这就意味着如果你想修改一个字符串，或者截取一个子串，或者在字符
串的末尾连接另一个字符串等等，你必须新建一个字符串。

>>> s='xingruiping'
>>> s[2]='K'

Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    s[2]='K'
TypeError: 'str' object does not support item assignment
```

36、Unicode字符串

```
ASCII码：str()和chr()
    UNICODE：unicode()和unichr()

    程序中出现字符串时一定要加个前缀 u.

    不要用 str()函数，用unicode()代替.
    不要用过时的 string 模块 -- 如果传给它的是非ASCII 字符，它会把一切搞砸。
    不到必须时不要在你的程序里面编解码 Unicod 字符.只在你要写入文件或数据
    库或者网络时，才调用encode()函数;
    相应地，只在你需要把数据读回来的时候才调用decode()函数.
```

37、列表

```
#添加新的元素，注意不能使用+号来添加元素，+号的使用级别是列
#表，即两个操作数都是列表的情况下才可以连接：

>>> s=[1,2,3,4]
>>> s.append(5)
>>> s
[1, 2, 3, 4, 5]
>>>
```

```
#序列类型函数
 sorted() and reversed():
 sorted()只是对s进行排序，并没有改变s的真实顺序
 >>>s=[4,5,43,2,543,2,432,64]
 >>> sorted(s)
 [2, 2, 4, 5, 43, 64, 432, 543]
 >>> sum(s)  #求和
 1095
```

```
list()和tuple()

list()函数和tuple()函数接受可迭代对象(比如另一个序列)作为参数,并通过浅拷贝
数据来创建一个新的列表或者元组。更多的情况下,它们用于在两种类型之间进行转换

sort()/reverse()和sorted()/reversed()的区别

sort()/reverse()会在原地操作，没有返回值，是通过对象的点调用的，对象被改变
sorted()/reversed()相当于表达式，将一个对象排序之后生成一个新的对象，原来
的对象并不改变，返回这个新的对象
```

38、元组

```
#元组是一种不可变类型.正因为这个原因,元组能做一些列表不能做的事情...
#用做一个字典的key.另外当处理一组对象时,这个组默认是元组类型.

#元组不可变也是有一定的限度的，若是元组中含有可变的元素，那么在这个
#层面上这个元组也是可以改变的，如：

>>> atuple=(['xyz',123],'abc',456)
>>> atuple
(['xyz', 123], 'abc', 456)
>>> atuple[0][1]
123
>>> atuple[0][1]='opq'
>>> atuple
(['xyz', 'opq'], 'abc', 456)
>>>
```

默认集合类型

#所有的多对象的，逗号分隔的，没有明确用符号定义的，等等这些集合默认的
#类型都是元组，所有函数返回的多对象（不包括有符号封装的）都是元组类型。

```
>>> def foo():
    return 1,2,3

>>> foo()
(1, 2, 3)
```

但不推荐这么做，还是显示定义的比较好

单元素元组

#由圆括号包裹的一个单一元素首先被作为分组操作，而不是作为元组的分界符。一个变
#通的方法是在第一个元素后面添一个逗号（,)来表明这是一个元组而不是在做分组操作.

```
>>> ('xyz')
'xyz'
>>> ('xyz',)
('xyz',)
>>>
```

深拷贝和浅拷贝

我觉得《核心编程》上关于深拷贝和浅拷贝的说法有一些错误：p241

首先，对象拷贝不是简单的把对象的引用赋值给另外一个变量，这样
不是拷贝，拷贝分为浅拷贝和深拷贝，不管是哪种拷贝，在最初都会
生成一个新的对象，虽然对象中有可能会存在引用到源对象。

其次，序列类型对象的浅拷贝是默认类型拷贝,并可以以下几种方式实施:
(1)完全切片操作[:]
(2)利用工厂函数,比如list(),*dict*()等
(3)使用copy 模块的copy 函数.

深拷贝通过copy模块的deepcopy()函数来实现，深拷贝的是列表元素，元组元素不进行深拷贝

以下有几点关于拷贝操作的警告。

第一：非容器类型(比如数字,字符串和其他"原子"类型的对象,像代码,类型和
xrange 对象等)没有被拷贝一说,浅拷贝是用完全切片操作来完成的.

第二：如果元组变量只包含原子类型对象,对它的深拷贝将不会进行。

深拷贝示例：

```
>>> person=['name',['savings',100.00]]
>>> import copy
>>> hubby=copy.deepcopy(person)
>>> wifey=copy.deepcopy(person)
>>> [id(x) for x in person]
[22135968, 27744536]
>>> [id(x) for x in hubby]
[22135968, 32582712]
>>> [id(x) for x in wifey]
[22135968, 32582952]
>>> hubby[0]='suo'
>>> wifey[0]='piao'
>>> person,hubby,wifey
(['name', ['savings', 100.0]], ['suo', ['savings', 100.0]], ['piao', ['savings', 100.0]])
>>> hubby[1][1]=50.00
>>> person,hubby,wifey
(['name', ['savings', 100.0]], ['suo', ['savings', 50.0]], ['piao', ['savings', 100.0]])
>>>
```

注意：进行拷贝的时候，一定要考虑到各种引用，不能想当然怎么样就怎么样

39、字典

```
键值对、映射类型

keys():        #返回字典中所有的键，以列表形式
values():      #返回字典中所有的值，以列表形式
items():       #返回字典中所有的元素，返回类型为以键值对组成的元组为元素的列表
has_key(key): #判断字典中是否存在key这个键，返回布尔值，类似的可以使用in或者是not in来判断

字典中的键是不可以改变的，所以数字和字符串可以作为字典中的键，但是列表和其他字典则不行。

不允许一个键对应多个值，键必须是可以哈希的

字典可以和所有的标准类型操作符一起工作,但却不支持像拼接（concatenation）
和重复(repetition)这样的操作。这些操作对序列有意义，可对映射类型行不通。
```

```
增加新元素，如下：

>>> dict2={x:'15364072939','piao':'18946799512'}
>>> dict2['dai']='15200859293'
>>> dict2
{'suo': '15364072939', 'piao': '18946799512', 'dai': '15200859293'}


映射类型的相关函数

dict([container]):   #创建字典的工厂函数。如果提供了容器类(container)，
                     #就用其中的条目填充字典，否则就创建一个空字典。
len(mapping):        #返回映射的长度(键-值对的个数)
hash(obj):           #返回obj 的哈希值
```

```
字典类型方法

dict.clear() #删除字典中所有元素
dict.copy()  #返回字典(浅复制)的一个副本
dict.fromkeys(seq,val=None) #创建并返回一个新字典，以seq 中的元素做该字典的键，
                            #val 做该字典中所有键对应的初始值(如果不提供此值，则默认为None)
dict.get(key,default=None)  #对字典dict 中的键key,返回它对应的值value，如果字典中不存在此
                            #键，则返回default 的值(注意，参数default 的默认值为None)
dict.has_key(key) #如果键(key)在字典中存在，返回True，否则返回False。在Python2.2
                  #版本引入in 和not in 后，此方法几乎已废弃不用了，但仍提供一个可工作的接口。
dict.items()      #返回一个包含字典中(键，值)对元组的列表
dict.keys()       #返回一个包含字典中键的列表
dict.iter()       #方法iteritems(), iterkeys(), itervalues()与它们对应的非迭代方法一样，不
                  #同的是它们返回一个迭代子，而不是一个列表。
dict.pop(key[, default]) #和方法get()相似，如果字典中key 键存在，删除并返回dict[key]，
                         #如果key 键不存在，且没有给出default 的值，引发KeyError 异常。
dict.setdefault(key,default=None) #和方法set()相似，如果字典中不存在key 键，由dict[key]=default 为它赋值。
dict.update(dict2)                #将字典dict2 的键-值对添加到字典dict
dict.values()                     #返回一个包含字典中所有值的列表

iteritems(), iterkeys(), 和itervalues()
#这些函数与返回列表的对应方法相似，只是它们返回惰性赋值的迭代器，所以节省内存。数据集如果很大会导致很难处理。
```

40、集合

集合(sets)有两种不同的类型，可变集合(*set*) 和 不可变集合(*frozenset*)。可变集合(*set*)不是可哈希的，因此既不能用做字典的键也不能做其他集合中的元素。不可变集合(*frozenset*)则正好相反，即，他们有哈希值，能被用做字典的键或是作为集合中的一个成员。

集合没有特定的语法格式，只能使用工厂方法set()、*frozenset*()来创建集合

set()和frozenset()工厂函数分别用来生成可变和不可变的集合。如果不提供任何参数，默认会生成空集合。如果提供一个参数，则该参数必须是可迭代的，即，一个序列，或迭代器，或支持迭代的一个对象。

例如：一个文件或一个字典。

```
>>>s=set('suoo')
s
>>> s
set(['s', 'u', 'o'])
>>> s=frozenset('piaodatou')
>>> s
frozenset(['a', 'd', 'i', 'o', 'p', 'u', 't'])
```

遍历集合

```
>>>for i in s:
    print i,

a d i o p u t
```

更新集合

```
#用各种集合内建的方法和操作符添加和删除集合的成员，注意：只有可变集合可以被修改
>>> s=set('piaodatou')
>>> s
set(['a', 'd', 'i', 'o', 'p', 'u', 't'])
>>> s.add('z')
>>> s.add('xyz')
>>> s
set(['a', 'd', 'i', 'xyz', 'o', 'p', 'u', 't', 'z'])
>>> s.update('gh')
>>> s
set(['a', 'd', 'g', 'i', 'h', 'xyz', 'o', 'p', 'u', 't', 'z'])
>>> s-=set('gh')
>>> s
set(['a', 'd', 'i', 'xyz', 'o', 'p', 'u', 't', 'z'])
```

集合等价与不等价

等价/不等价被用于在相同或不同的集合之间做比较。两个集合相等是指，对每个集合而言，当且仅当其中一个集合中的每个成员同时也是另一个集合中的成员。
你也可以说每个集合必须是另一个集合的一个子集， 即，s <= t 和 s >= t 的值均为真(True)，或(s <= t and s>= t) 的值为真(True)。集合等价/不等价与集合的类型或集合成员的顺序无关，只与集合的元素有关。

这与数学中的集合的概念非常类似

```
集合的交、并、补、对称差分（适用于所有的集合类型，可变和不可变的）

>>> s=set('suo')
>>> t=set('piao')
>>> s
set(['s', 'u', 'o'])
>>> t
set(['i', 'p', 'a', 'o'])
>>> s|t     #并集
set(['a', 'p', 's', 'u', 'i', 'o'])
>>> s&t     #交集
set(['o'])
>>> s-t     #差集
set(['s', 'u'])
>>> t-s     #差集
set(['i', 'p', 'a'])
>>> s^t     #对称差分
set(['a', 'i', 'p', 's', 'u'])

如果左右两个操作数的类型相同，既都是可变集合或不可变集合，则所产生的结果类型是相同
的，但如果左右两个操作数的类型不相同(左操作数是set，右操作数是frozenset，或相反情况)，
则所产生的结果类型与左操作数的类型相同
只适用于可变集合的类型操作符

s|=t   s&=t    s-=t   s^=t

集合类型内建方法

适用于所有的集合类型：

s.issubset(t)             #如果s 是t 的子集，则返回True,否则返回False
s.issuperset(t)           #如果t 是s 的超集，则返回True,否则返回False
s.union(t)                #返回一个新集合，该集合是s 和t 的并集
s.intersection(t)         #返回一个新集合，该集合是s 和t 的交集
s.difference(t)           #返回一个新集合，该集合是s 的成员，但不是t的成员
s.symmetric_difference(t) #返回一个新集合，该集合是s 或t 的成员，但不是s和t共有的成员
s.copy()                  #返回一个新集合，它是集合s 的浅复制

适用于可变集合：

add(), remove(), discard(), pop(), clear().


集合模块
```

41、条件和循环

```
条件表达式，即三元操作符

>>> x=4
>>> y=8
>>> smaller=x if x<y else y
>>> smaller
4

与序列相关的内建函数

sorted(), reversed(), enumerate(), zip()
其中两个函数( sorted() 和 zip() )返回一个序列(列表)，而另外两个函数
( reversed() 和 enumerate() )返回迭代器(类似序列)

continue

continue 语句并不是"立即启动循环的下一次迭代"。实际上，当遇到continue 语句时，程
序会终止当前循环,并忽略剩余的语句，然后回到循环的顶端。在开始下一次迭代前,如果是条件循
环，我们将验证条件表达式.如果是迭代循环,我们将验证是否还有元素可以迭代。只有在验证成功
的情况下，我们才会开始下一次迭代。
```

```
pass

不做任何事情，类似c语言中的空的大括号

while-else和for-else

# -*- coding: cp936 -*-
#求一个数的最大约数，练习使用while-else语句
#当循环全部正常执行完毕之后，才会执行else中的语句，break也算是非正常结束循环
def showMaxFactor(num):
    count=num/2
    while count>1:
        if num%count==0:
            print 'largest factor of %d is %d' % (num,count)
            break
        count-=1
    else:
        print num,'is prime'

for eachNum in range(10,21):
    showMaxFactor(eachNum)
```

迭代器

迭代器就是有一个 next() 方法的对象，而不是通过索引来计数．当你或是一个循环机制(例如 for 语句)需要下一个项时，调用迭代器的 next() 方法就可以获得它．条目全部取出后，会引发一个StopIteration 异常，这并不表示错误发生，只是告诉外部调用者，迭代完成。如下：

```
>>> s=(123,'suo',56.78)
>>> s
(123, 'suo', 56.78)
>>> i=iter(s)
>>> i.next()
123
>>> i.next()
'suo'
>>> i.next()
56.78
>>> i.next()

Traceback (most recent call last):
  File "<pyshell#359>", line 1, in <module>
    i.next()
StopIteration
```

自己写的for循环的迭代器的内部机制：

```
seq=[123,'suo',45.67,7896L]

fetch=iter(seq)

while True:
    try:
        i=fetch.next()
    except StopIteration:
        break
    print i
```

```
1）迭代序列
2）迭代字典
字典的迭代器会遍历它的键(keys).
语句 for eachKey in myDict.keys() 可以缩写为 for eachKey in myDict

Python 还引进了三个新的内建字典方法来定义迭代: myDict.iterkeys() (通过 keys 迭
代), myDict.itervalues() (通过 values 迭代), 以及 myDicit.iteritems() (通过 key/value 对
来迭代).

3）迭代文件
文件对象生成的迭代器会自动调用 readline() 方法. 这样, 循环就可以访问文本文件的所有行.
程序员可以使用 更简单的 for eachLine in myFile 替换 for eachLine in myFile.readlines()

4）创建迭代器
iter(obj)
如果你传递一个参数给 iter() , 它会检查你传递的是不是一个序列, 如果是, 那么很简单:
根据索引从 0 一直迭代到序列结束. 另一个创建迭代器的方法是使用类, 一个实现了 __iter__()
和 next() 方法的类可以作为迭代器使用.

iter(func, sentinel )
如果是传递两个参数给 iter() , 它会重复地调用 func , 直到迭代器的下个值等于
sentinel .
```

列表解析（List comprehensions）

只用一行代码就可以创建包含特定内容的列表.

```
>>> [x ** 2 for x in range(6)]
[0, 1, 4, 9, 16, 25]

>>> seq = [11, 10, 9, 9, 10, 10, 9, 8, 23, 9, 7, 18, 12, 11, 12]
>>> [x for x in seq if x % 2]
[11, 9, 9, 9, 23, 9, 7, 11]

>>> [(x+1,y+1) for x in range(3) for y in range(5)]
[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (2, 1), (2, 2), (2,
3), (2, 4), (2, 5), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5)]
```

注：列表解析支持多重嵌套for 循环以及多个 if 子句.
在同一个列表中的这两个for循环，他们的关系是上下级的关系，即后一个for循环是在前一个for循环中的，并且变量的定义是在for循环中定义的，并不是在for循环前面进行的定义

```
>>> f = open('hhga.txt', 'r')
>>> len([word for line in f for word in line.split()])
91
```

生成器表达式

生成器表达式是列表解析的一个扩展，生成器是特定的函数，允许你返回一个值，然后"暂停"代码的执行，稍后恢复.

列表解析的一个不足就是必要生成所有的数据，用以创建整个列表．这可能对有大量数据的迭代器有负面效应．生成器表达式通过结合列表解析和生成器解决了这个问题.

生成器表达式与列表解析非常相似，而且它们的基本语法基本相同；不过它并不真正创建数字列表，而是返回一个生成器，这个生成器在每次计算出一个条目后，把这个条目“产生”(yield)出来．生成器表达式使用了"延迟计算"(lazy evaluation)，所以它在使用内存上更有效

```
它是一个表达式，和列表解析有根本的不同，它返回一个表达式，即生成器。

示例：

>>> sum(len(word) for line in data for word in line.split())
408

f = open('/etc/motd', 'r')
longest = max(len(x.strip()) for x in f)
f.close()
return longest

注：生成器表达式用起来，和列表解析还是一样的，只是处理的机制不一样了，生成器表达式更加的高效，生成器表达式生成一个“懒惰”的序列表达式，可以将它传递给函数，进行进一步操作。

迭代器模块：itertools
```

42、异常

```
通用的异常和异常捕获，使用多个except

def safe_float(obj):
    try:
        retval = float(obj)
    except ValueError:
        retval = 'could not convert non-number to float'
    except TypeError:
        retval = 'object type cannot be converted to float'
    return retval
```

```
一个except处理多个异常，except 语句在处理多个异常时要求异常被放在一个元组里

def safe_float(obj):
    try:
        retval = float(obj)
    except (ValueError, TypeError):
        retval = 'argument must be a number or numeric string'
    return retval
```

```
异常参数

异常参数将会是一个包含来自导致异常的代码的诊断信息的类实例.
def safe_float(object):
    try:
        retval = float(object)
    except (ValueError, TypeError), diag:
        retval = str(diag)
    return retval

else子句

在try 范围中没有异常被检测到时,执行else 子句.
```

```
finally子句

finally 子句是无论异常是否发生,是否捕捉都会执行的一段代码


下面是try-except-else-finally 语法的示例:
try:
    A
except MyException,e:
    B
else:
    C
finally:
    D
```

43、函数

```
若函数没有返回值，则默认的返回值是None

装饰器

装饰器实际就是函数，它接受函数对象。我们在执行函数之前，可以运行些预备代码，也可以在执行代码之后做些清理工作。这类似于java中的AOP，即面向切面编程，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
```

```
1. 无参装饰器:

@deco2
@deco1
def func(arg1, arg2, ...):
    pass

这和创建一个组合函数是等价的:

def func(arg1, arg2, ...):
    pass
func = deco2(deco1(func))
```

```
2. 有参装饰器:

@deco1(deco_arg)
@deco2
def func():
    pass

#这等价于:

func = deco1(deco_arg)(deco2(func))

#示例:
from time import ctime,sleep

def tsfunc(func):
    def wrappedFunc():
        print '[%s] %s() called' % (ctime(), func.__name__)
        return func()
    return wrappedFunc

@tsfunc
def foo():
    print 'foo() is invoked !'

foo()
sleep(4)

for i in range(2):
    sleep(1)
    foo()
```

```
运行结果:

[Tue Jul 17 22:45:54 2012] foo() called
foo() is invoked !
[Tue Jul 17 22:45:59 2012] foo() called
foo() is invoked !
[Tue Jul 17 22:46:00 2012] foo() called
foo() is invoked !
```

传递函数

函数也是phthon对象的一种，也是一个对象，也可以将函数对象的引用赋值给一个变量，通过这个变量，也相当于是这个函数的别名，来调用这个函数，基于这种机制，就有了传递函数，即可以将一个函数名通过形参传递给另外一个函数，在这个函数中调用传递进来的函数。如下示例：

```
>>> def foo():
    print 'in foo()'

>>> bar=foo
>>> bar()
in foo()

>>> def bar(argfunc):
    argfunc()

>>> bar(foo)
in foo()
>>>
```

可变长度的参数

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的方法。

1. 非关键字可变长参数（元组）

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):pass
```

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。

之前，只要在函数调用时给出不正确的函数参数数目，就会产生一个TypeError异常。通过末尾增加一个可变的参数列表变量，我们就能处理当超出数目的参数被传入函数的情形，因为所有的额外（非关键字）参数会被添加到变量参数元组。

示例：

```
>>> def tupleVarArgs(arg1,arg2='defaultB',*rest):
    print 'formal arg1: ',arg1
    print 'formal arg2: ',arg2
    for eachXarg in rest:
        print 'another arg: ',eachXarg

>>> tupleVarArgs('abc')
formal arg1:  abc
formal arg2:  defaultB

>>> tupleVarArgs('abc',25.0)
formal arg1:  abc
formal arg2:  25.0

>>> tupleVarArgs('abc',25.0,'suo','piao',999999L)
formal arg1:  abc
formal arg2:  25.0
another arg:  suo
another arg:  piao
another arg:  999999
```

精要：多余的参数保存到一个元组中，传递给函数

2.关键字变量参数（Dictionary）

```
#语法：

def function_name([formal_args,][*vargst,] **vargsd):pass

#示例：

>>> def dicVarArgs(arg1,arg2='defaultB',**rest):
    print 'formal arg1: ',arg1
    print 'formal arg2: ',arg2
    for key in rest:
        print 'xtra arg %s: %s ' % (key,rest[key])
```

```
>>> dicVarArgs('abc')
formal arg1:  abc
formal arg2:  defaultB

>>> dicVarArgs('abc',123)
formal arg1:  abc
formal arg2:  123

>>> dicVarArgs('abc',123,c='suo',d=456,e=9999L)
formal arg1:  abc
formal arg2:  123
xtra arg c: suo
xtra arg e: 9999
xtra arg d: 456
>>>
```

精要：多余的关键字参数，将其放到字典中，传递给函数

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，如下示例：

```
>>> def varArgs(arg1,arg2='defaultB',*tupleRest,**dicRest):
    print 'formal arg1: ',arg1
    print 'formal arg2: ',arg2
    for each in tupleRest:
        print 'non-key arg: ',each
    for key in dicRest:
        print 'xtra arg %s: %s ' % (key,dicRest[key])

>>> varArgs('abc')
formal arg1:  abc
formal arg2:  defaultB

>>> varArgs('abc',123)
formal arg1:  abc
formal arg2:  123

>>> varArgs('abc',123,'suo',456,c='piao',d='love',e=9999)
formal arg1:  abc
formal arg2:  123
non-key arg:  suo
non-key arg:  456
xtra arg c: piao
xtra arg e: 9999
xtra arg d: love
```

其实，也可以这样来调用带有可变长参数的函数：

```
>>> mytuple=('suo','love','piao',9999)
>>> mydic={'c':'how','d':'are','e':'you','f':88888L}
>>> varArgs('abc',123,*mytuple,**mydic)
formal arg1:  abc
formal arg2:  123
non-key arg:  suo
non-key arg:  love
non-key arg:  piao
non-key arg:  9999
xtra arg c: how
xtra arg e: you
xtra arg d: are
xtra arg f: 88888
>>>
```

这样的调用，更加清晰一些。

（5）函数式编程

1. *Lambda 匿名函数*

python 允许用lambda 关键字创造匿名函数，语法为：

lambda [*arg1*[, *arg2*, ... *argN*]]: expression

lambda必须放在一行中，就像是单行版的函数一样，但应该叫它lambda表达式

它返回一个函数对象，这个函数执行的操作，就是expression中的内容

示例：

```
>>> a=lambda x,y=2:x+y
>>> a(4)
6
>>> b=lambda *z : z
>>> b(1,2,3)
(1, 2, 3)
```

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 表达式没有在变量第一*

```
个函数，当被调用时，创建一个框架对象。


2.内建函数
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
的结果和下面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。


（6）变量作用域
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。

问题引出：
在函数中定义了一个局部的bar变量，在主函数体中，定义了一个全局的bar变量，但是有可能在函数体
内的bar变量会覆盖掉全局的bar变量，这样全局的bar变量在函数体内就失效了。。

如下示例：
bar=100

def foo():
    print 'calling foo()...'
    bar=200
    print 'in foo(), bar is ', bar


print 'bar=',bar
foo()
print 'bar=',bar

运行结果如下：
bar= 100
calling foo()...
in foo(), bar is  200
bar= 100

问题解决：
为了解决上面的问题，我们可以使用global关键字，明确的引用一个已命名的全局变量

bar=100

def foo():
    print 'calling foo()...'
    global bar
    bar=200
    print 'in foo(), bar is ', bar


print 'bar=',bar
foo()
print 'bar=',bar

运行结果如下：
bar= 100
calling foo()...
in foo(), bar is  200
bar= 200


（7）递归
求阶乘：

>>> def factorial(n):
    if n==0 or n==1:
        return 1
    else:
        return (n*factorial(n-1))


>>> factorial(5)
120


(8)生成器，暂时跳过
```

44、模块

```
修改搜索路径

在交互模式下修改sys.path，将自己写的模块所在的目录放到这个搜索路径中，那么
解释器就可以找到在这个路径下所定义的模块了，示例：

>>>import sys
>>> sys.path.append(r'E:\My DBank\python\py\mymodule')

若要移除这个路径，那么可以使用列表中的pop()方法
```

名称空间

名称空间是名称(标识符)到对象的映射，向名称空间添加名称的操作过程涉及到绑定标识符到指定对象的操作(以及给该对象的引用计数加 1)。

在执行期间有两个或三个活动的名称空间。 这三个名称空间分别是局部名称空间，全局名称空间和内建名称空间，但局部名称空间在执行期间是不断变化的，所以我们说"两个或三个"。

Python 解释器首先加载内建名称空间。 它由 __builtins__ 模块中的名字构成。 随后加载执行模块的全局名称空间，它会在模块开始执行后变为活动名称空间。如果在执行期间调用了一个函数，那么将创建出第三个名称空间，即局部名称空间。

通过 globals() 和 locals() 内建函数判断出某一名字属于哪个名称空间

在遇到名称空间的时候想想"它存在吗?"，遇到变量作用域的时候想想"我能看见它吗?"

使用类实例来创建名称空间

有点晕啊，python怎么那么随便呢？竟然可以这样来写类，而且可以在外部动态的给类添加属性，有点接受不了啊，使用类实例可以充当命名空间，这个。。。不知道该怎么说啊

```
>>> class MyClass(object):
    pass

>>> bag=MyClass()
>>> bag.x=100
>>> bag.y=200
>>> bag.version=0.1
>>> bag.completed=False

>>> bag.y
200
>>> bag.version
0.1
```

推荐的模块导入顺序

我们推荐所有的模块在 Python 模块的开头部分导入。 而且最好按照这样的顺序：
- Python 标准库模块
- Python 第三方模块
- 应用程序自定义模块

扩展的模块导入语句(as)

使用as关键字，可以将导入的模块的名字或者是导入的模块的属性的名字换成你想要的名字如下：

```
>>>import MyModule as m
>>> m.test()
ran test()

>>> from MyModule import test as t
>>> t()
ran test()
```

导入到当前名称空间的名称

一个模块只被加载一次，不管它被导入多少次。
调用 from-import 可以把名字导入当前的名称空间里去。

有时候使用from-import语句导入的模块中的属性，会和程序之前所定义的变量相冲突，遇到这种情况，唯一的解决办法，就是不要使用from-import语句，而是通过模块名字来调用属性，以防止冲突

模块内建函数

1. __import__()

import语句是通过调用__import__()函数来完成导入的工作的，语法为：

```
__import__(module_name[, globals[, locals[, fromlist]]])
```

2. globals()和locals()
globals() 和 locals() 内建函数分别返回调用者全局和局部名称空间的字典
在一个函数内部，局部名称空间代表在函数执行时候定义的所有名字，locals() 函数返回的就是包含这些名字的字典。 globals() 会返回函数可访问的全局名字。
在全局名称空间下，globals() 和 locals() 返回相同的字典，因为这时的局部名称空间就是全局空间
在局部名称空间下，globals()返回的是全局的的名称空间的字典，而locals()返回局部的名称空间的字典。

示例：

```
>>> def foo():    #局部
```

```
        print 'calling foo()...'
        aString='bar'
        anInt=42
        print 'foo() globals: ', globals().keys()
        print 'foo() lobals: ', locals().keys()


#全局<span style="white-space:pre">    </span>
>>> print '__main__ globals: ', globals().keys()
__main__ globals:  ['__builtins__', '__package__', '__name__', 'foo', '__doc__']


>>> print '__main__ locals: ',locals().keys()
__main__ locals:  ['__builtins__', '__package__', '__name__', 'foo', '__doc__']


>>> foo()
calling foo()...
foo() globals:  ['__builtins__', '__package__', '__name__', 'foo', '__doc__']
foo() lobals:  ['anInt', 'aString']


包
包是一个有层次的文件目录结构，它定义了一个由模块和子包组成的 Python 应用程序执行环境。

就好像java中的包的概念是类似的，而且与类和模块相同，包也使用句点属性标识来访问他们的元素。

示例：
import Phone.Mobile.Analog
from Phone import Mobile
from Phone.Mobile.Analog import dial
from package.module import *


相对导入和绝对导入

import 语句总是绝对导入的，所以相对导入只应用于 from-import 语句。


# -*- coding: UTF-8 -*-

使用指定的编码格式来解析模块文件
```

45、面向对象编程

```
类最简单的使用情况就是使用类作为命名空间，这就像c++中的结构体一样，但是
这样在类外加进去的属性属于类实例的属性，并不是类的属性，这些属性实质上是
动态的，不需要在其他的地方预先声明


类通常在一个模块的顶层进行定义，以便类实例能够在类所定义的源代码文件中的
任何地方被创建


类属性

属性就是属于一个对象的数据或者函数元素，包括数据属性和函数属性，即数据和
函数都属于属性的范畴


1. 实例数据属性：

2. 类数据属性：仅当需要有更加"静态"数据类型时才变得有用，它和任何实例都无
关，它相当于java中的静态变量，即用static标志的成员变量。如下：

>>> class C(object):
foo=100     #foo即为C类的数据属性

3. 方法
类中的方法必须通过类实例的点调用来访问，即方法必须绑定（到一个实例）才能
直接被调用。

>>> class MyClass(object):
        def myNoActionMethod(self):
            pass
>>> mc=MyClass()
>>> mc.myNoActionMethod()


4.查看类的属性
两种方法，使用内建函数dir()或者是查看类的特殊属性__dict__，dir()返回的是
所有的类属性的一个名字列表，而__dict__返回的是一个字典，键是属性名，值是
属性的数据值。


示例：
>>> class MyClass(object):
        version=1.0
        def func(self):
            pass
```

```
>>> dir(MyClass)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribu

>>> MyClass.__dict__
dict_proxy({'__module__': '__main__', 'version': 1.0, 'func': <function func at

5. 类的特殊属性
__name__、__doc__、__bases__、__module__、__dict__、__class__是所有类都
具备的特殊类属性。

__dict__：
属性包含一个字典，由类的数据属性组成。访问一个类属性的时候，Python 解释
器将会搜索字典以得到需要的属性。如果在__dict__中没有找到，将会在基类的字
典中进行搜索，采用"深度优先搜索"顺序。

__module__：
这个属性说明了这个类是位于哪个模块中的，python 支持模块间的类继承，如
>>> MyClass.__module__
'__main__'


说明MyClass这个类是在__main__模块中的，如果将这个类定义到别的模块中，那
么__module__存储的就是这个类所在的模块。如下：


>>> from ClassModule import MyModule
>>> MyModule
<class 'ClassModule.MyModule'>
>>> MyModule.__module__
'ClassModule'
>>>

__bases__：对任何类，它是一个包含其父类的集合的元组，只包含直接父类。



实例

实例化一个类就像调用一个函数一样，不需要new等关键字，如
myClass=MyClass()
类的__init__()和__del__()方法：
__init__()是类的初始化方法，也相当于类的构造方法，因为它可以指定类的
参数，但是它确实是一个类的初始化方法，毕竟类的实例的创建工作，不是
由__init__()来做的。若在自定义的类中没有定义__init__()方法，那么这个
方法依然会被调用，但是没有做任何的工作，如果定义了这个方法，那么就覆
盖了默认的这个方法，这个方法就会被调用，做你想让它做的事。
__del__()方法是python类中的析构方法，类似于c++中的析构方法，但不同的
是，这个函数要直到该实例对象所有的引用都被清除掉后才会执行。


示例：
#类的实例的跟踪计数
class MyClass(object):
    count=0                 #静态成员变量，被所有的实例所共享


    def __init__(self):
        MyClass.count+=1    #访问类的数据属性，一定要加上命名空间，否则访问出错


    def __del__(self):
        MyClass.count-=1


    def howMany(self):
        return MyClass.count


>>> a=MyClass()
>>> b=MyClass()
>>> c=MyClass()   #注意，这里生成了3个类的实例，每一个实例只有一个引用，所以删
<span style="white-space:pre">      </span>  #之后，__del__()方法就会执行了
>>> MyClass.count
3
>>> a.howMany()
3
>>> del a
>>> MyClass.count
2
>>> del c
>>> del b
>>> MyClass.count
0
```

实例属性和类属性

类属性包括两种：数据属性和方法属性

实例属性只包括数据属性，方法属性是属于类属性中的，实例属性的数据属性也包括两种：类属性中的数据属性和自定义的数据属性

实例仅拥有数据属性（方法严格来说是类属性），实例的数据属性只是与某个类的实例相关联的数据值，并且可以通过句点属性标识法来访问。这些值独立于其它实例或类。当一个实例被释放后，它的属性
同时也被清除了。

python能够在"运行时"创建实例属性。构造器__init()__是设置这些属性的关键点之一。

动态的只是数据属性，方法属性还是用的类中的方法属性，这也是类和实例唯一有关系的地方。

python的这种特性让我一时难以接受，如果这样可以的话，那么我想问一句：这还是面向对象吗？类还是实例的"蓝本"吗？实例具有的属性，类竟然不具有，这有点像用同一个模具创出来的东西，虽然大体上看起来很像，但是总会有那么几个小小的细节是不一样的，俗话说的好，没有两片相同的叶子。python这样的特性，好像是更加的具有哲理，更加的符合常理了。不由得惊叹。。。。。。这也颠覆了多年java和c↔培养起来的面向对象的概念。。。。。。但是应用这种特性也要格外的小心，不然会变得很乱。。。。。

__init__()方法是初始化实例的，所以将实例属性放到这个地方来进行初始化，是最合适不过的了。同时可以在__init__()方法中传入默认的参数，那么在实例化类时，就可以不传递这些参数，而使用默认的值了，如下：

```
class AttrClass(object):


    def __init__(self,name,age=23,sex='male'):
        self.name=name
        self.age=age
        self.sex=sex       #这里使用self表示当前实例的数据属性，不用在类中进行声

    def show(self):
        print self.name, self.age, self.sex



>>> suo=AttrClass('suo')
>>> suo.show()
suo 23 male
>>> piao=AttrClass('piao',21,'female')
>>> piao.show()
piao 21 female
>>>
```

实例的dir()和__dict__的区别：
__dict__只显示实例属性，即只显示实例的数据属性，而dir()显示类属性和实例属性，即显示实例所属类中的数据属性和方法属性，也显示实例中的数据属性。

内建类型

内建类型也属于类，也有属性，不过内建类型的实例没有__dict__属性，内建类型就是complex、int、float等

类属性和实例属性的区别

类属性说的是类内定义的数据属性（静态）和方法，而实例属性只包括"动态"添加到实例中的数据属性。

类属性仅是与类相关的数据值，和实例属性不同，类属性和实例无关。

类和实例都是名字空间。类是类属性的名字空间，实例则是实例属性的。

类属性可通过类或实例来访问。类属性就好像java中的静态成员变量。

当通过实例来访问类属性的话，会首先在实例属性中找这个属性，如果没找到才会去类属性中找，如果还没找到，那么就到基类中去找

注意，如果要更新类属性的值，只能通过类引用类属性才能更新，如果通过实例去更新的话，只会对这个实例生成一个新的实例属性，这个实例属性的名字和类属性的名字是相同的，这样的话，就遮蔽了实例对类属性的访问，更改的也只是实例名字空间内的这个属性，而不是类域的这个属性，只能显示的del掉这个实例属性之后，才能正常的通过实例来访问类属性，可以通过查看实例的__dict__变量来查看当前实例的实例属性
如下：

```
>>> class C(object):
    version=1.0    #类属性


    >>> C.version    #通过类名访问
    1.0
    >>> c=C()
    >>> c.version   #通过实例访问
    1.0
    >>> c.version=2.0   #想通过实例更改类属性
    >>> c.version       #但是只是为这个实例多生成了一个实例属性，并没有更改类属性
    2.0
    >>> C.version
    1.0
    >>> c.__dict__
    {'version': 2.0}
    >>> del c.version   #删除掉这个实例属性之后，才可以正常的通过实例来访问类属性
    >>> c.version
    1.0
    >>> C.version+=1
    >>> C.version
    2.0
    >>> c.version
    2.0
    >>> c.__dict__
    {}
    >>>
```

绑定和方法调用

方法仅仅是类内部定义的函数。(这意味着方法是类属性而不是实例属性)。

方法只有在其所属的类拥有实例时，才能被调用。当存在一个实例时，方法才被认为是绑定到那个实例了。没有实例时方法就是未绑定的。

任何一个方法定义中的第一个参数都是变量self，它表示调用此方法的实例对象。也就是代表这些方法所绑定的实例。这个self是必须要作为第一个参数传递的，它表示要将哪个实例和类方法进行绑定。

静态方法和类方法

类方法和静态方法的区别我还没有弄清楚。。。。。不过类方法和静态方法都可以通过类名来调用。

类方法，需要类而不是实例作为第一个参数，它是由解释器传给方法。类不需要特别地命名，类似self，不过很多人使用cls 作为变量名字。

示例：

```
# -*- coding: cp936 -*-
class AttrClass(object):


    def __init__(self,name,age=23,sex='male'):
        self.name=name
        self.age=age
        self.sex=sex

    def show(self):
        print self.name, self.age, self.sex


    #使用staticmethod()内建方法，将一个方法设置为静态方法
    def staticFoo1():
        print 'calling static method foo1()'
    staticFoo1=staticmethod(staticFoo1)


    #使用classmethod()内建方法，将一个方法设置为类方法
    def classMethod1(cls):
        print 'calling class method1'
    classMethod1=classmethod(classMethod1)

    #使用修饰符将一个方法设置为静态方法
    @staticmethod
    def staticFoo2():
        print 'calling static method foo2()'


    #使用修饰符将一个方法设置为类方法
    @classmethod
    def classMethod2(cls):
```

```
        print 'calling class method2'


继承

一个子类可以继承它的基类的任何属性，不管是数据属性还是方法。
文档字符串对类，函数/方法，还有模块来说都是唯一的，所以特殊属性__doc__不会
从基类中继承过来
# -*- coding: cp936 -*-
class AddrBookEntry(object):
    'address book entry class'

    '''
    (1)这个__init__方法不是类的构造方法，而是一个在返回类实例之前的一个初始
    化方法，
       会被隐式的调用
    (2)方法的第一个参数为self，它代表实例对象本身，这个参数不用显示的传递进
    来，是自动传入的
    '''
    def __init__(self,nm,ph):
        self.name=nm
        self.phone=ph
        print 'Created instance for: ',self.name

    def updatePhone(self,newph):
        'update...'
        self.phone=newph
        print 'Update phone for: ',self.name



'''''''
(1)这个类继承自AddrBookEntry类，父类是写在类名后的小括号中的，如果有多个父类
要继承，那么就可以把所有的父类以逗号相隔，写在括号中

(2)子类也要有这个__init__方法，并且在子类的__init__方法中，要用父类的类名调
用父类的__init__方法，并且一定要显示的写上第一个参数为self

(3)子类继承了父类的所有的方法和属性
'''
class EmplAddrBookEntry(AddrBookEntry):
    'Employee Address Book Entry class extend from AddrBookEntry'
    def __init__(self,nm,ph,id,em):
        #AddrBookEntry.__init__(self,nm,ph)
        super(EmplAddrBookEntry,self).__init__(nm,ph)#两种调用父类方法的方法
                                                     #推荐使用<span style="whit
                                                     #</span>#super(), 因为它不

        self.empid=id
        self.email=em


    def updateEmail(self,newem):
        self.email=newem
        print 'Update e-mail address for:', self.name


内建函数

issubclass(sub, sup): : #给出的子类sub 确实是父类sup 的一个子类，则返回True，否
isinstance(obj1, obj2): #在obj1 是类obj2 的一个实例，或者是obj2 的子类的
                        #一个实例时，返回True（反之，则为False）


hasattr(), getattr(),setattr(), delattr()

当使用这些函数时，你传入你正在处理的对象作为第一个参数，但属性名，也就是这些
函数的第二个参数，是这些属性的字符串名字。换句话说，在操作obj.attr 时，就相
当于调用*attr(obj,'attr'....)系列函数


>>> class myClass(object):
... def __init__(self):
... self.foo = 100
...
>>> myInst = myClass()
>>> hasattr(myInst, 'foo')
True
>>> getattr(myInst, 'foo')
100
>>> hasattr(myInst, 'bar')
False
>>> getattr(myInst, 'bar')
Traceback (most recent call last):
File "<stdin>", line 1, in ?
getattr(myInst, 'bar')
```

```
AttributeError: myClass instance has no attribute 'bar'
>>> getattr(c, 'bar', 'oops!')
'oops!'
>>> setattr(myInst, 'bar', 'my attr')
>>> dir(myInst)
['__doc__', '__module__', 'bar', 'foo']
>>> getattr(myInst, 'bar') # same as myInst.bar #等同于 myInst.bar
'my attr'
>>> delattr(myInst, 'foo')
>>> dir(myInst)
['__doc__', '__module__', 'bar']
>>> hasattr(myInst, 'foo')
False
```

```
input()和raw_input()的区别


#input 不同于raw_input()，因为raw_input()总是以字符串的形式，逐字地返回用
#户的输入。input()履行相同的的任务；而且，它还把输入作为python 表达式进行求值。这意味着
#input()返回的数据是对输入表达式求值的结果：一个python 对象

```