Python包含6种内建序列：列表，元组，字符串，Unicode字符串，buffer对象和xrange对象。

**列表和元组主要的区别在于：列表可以修改，元组则不能。**

**通用序列操作：**

序列(如列表和元组)都可以进行特定测操作，如：索引，分片，加，乘以及检查某个元素是否属于序列的成员。Python还有计算序列长度，找出最大元素和最小元素的内建函数。

**索引：**

greeting = 'Hello'

print greeting[0] #结果为'H'

print greeting[-1] #结果为'o'

可以通过索引获取元素。最后一个元素的位置编号是-1。

字符串字面值(其他序列字面量亦可)能够直接使用索引，而不需要一个变量引用他们。

实例：

'Hello'[1] #结果为'e'

实例：

根据输入的年月日以数字形式打印出日期

#!/usr/bin/python

#coding: utf-8

months = [

'January',

'February',

'March',

'April',

'May',

'June',

'July',

'August',

'September',

'October',

'November',

'December'

]

endings = ['st', 'nd', 'rd'] + 17 \* ['th'] \

+ ['st', 'nd', 'rd'] + 7 \* ['th'] \

+ ['st']

#此处的这些东西表示序数词后缀，在英文中使用，具体规律如下：

#除了11,12,13是th之外,逢1就是st,２就是nd,３就是rd,其他都是th

#以200-300之内为例

#201,221,231,241,251,261,271,281,291st

#202,222,232,342,252,262,272,282,292nd

#203,223,233,243,253,363,273,283,293rd

#其他包括211,212,213都是th。

#17 \* ['th']表示生成一个列表。里面元素全为th，共17个元素

year = raw\_input('Year: ')

month = raw\_input('Month (1-12): ')

day = raw\_input('Day (1-31): ')

month\_number = int(month)

day\_number = int(day)

month\_name = months[month\_number - 1]

ordinal = day + endings[day\_number - 1]

print month\_name + ' ' + ordinal + '. ' + year

**分片：**

分片操作来访问一定范围内的元素。分片通过冒号相隔的两个索引来实现。

实例：

tag = '<a href="http://www.python.org">Python web site</a>'

tag[9:30] #结果为'http://www.python.org'

tag[32:-4] #结果为'Python web site'

分片的第一个索引是需要提取部分的第一个元素的编号，而最后的索引则是分片之后剩下部分的第一个元素的编号，也就是说分片包含头步索引所指向的元素而不包含尾部索引所指向的元素。

实例：

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

numbers[3:6] #结果是[4, 5 ,6]

numbers[0:1] #结果是[1]

如果要访问最后的三个元素，显式操作：

numbers[7:10] 结果是[8, 9 ,10]

其中索引10指向的是第11个元素，这个元素并不存在，但是却是在最后一个元素之后

从列表的结尾开始计数：

实例：

numbers[-3:-1] #结果是[8, 9]

numbers[-3:0] #结果是[]

这种方式不能访问最后的元素。

只要分片最左边的索引比他右边的晚出现在序列中，结果就是一个空的序列。

分片所得部分包含序列结尾的元素，只需要置空最后一个索引即可：

实例：

numbers[-3:] #结果是[8, 9, 10]

这个方法同样适用于序列开始的元素：

实例：

numbers[:3] #结果是[1, 2, 3]

如果复制整个数组，可以将两个索引都置空：

实例：

numbers[:] #结果是[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

**分片中的步长：**

指定第三个参数

实例：

numbers[0:10:1] #结果是[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

numbers[0:10:2] #结果是[1, 3, 5, 7, 9]

numbers[3:6:3] #结果是[]

实例：将每4个元素朱红的第一个提取出来，那么只要将步长置为4即可。

numbers[::4] #结果是[1, 5, 9]

当步长为负数时，从右到左提取元素。

实例：

numbers[8:3:-1] #结果是[9, 8, 7, 6, 5]

步长不可以是0，这样不会向下执行

使用负数作为步长时，必须让开始点大于结束点。

**序列相加**

通过使用加号可以进行序列的连接操作：

实例：

[1, 2, 3] + [4, 5, 6] #结果是[1, 2, 3, 4, 5, 6]

'Hello. ' + 'world!' #结果是'Hello. world!'

但是列表和字符串是无法连接在一起的。尽管他们都是序列。两种相同类型的序列才能进行连接操作。

**乘法：**

用数字x乘以一个序列会生成新的序列，而在新的序列中，原来的序列将被重复x次。

实例：

'python' \* 5 #结果是'pythonpythonpythonpythonpython'

[42] \* 5 #结果是[42, 42, 42, 42, 42]

**None、空列表和初始化：**

空列表可以简单的通过两个中括号进行表示[]，里面什么也没有。

None，是一个Python内建值，表示什么也没有。如果想初始化一个长度为10的空列表，可以使用乘法。

实例：

sequence = [None] \* 7; #结果是[None, None, None, None, None, None, None]

**成员资格：**

in运算符：检查一个值是否在序列中，条件为真返回True，条件为假返回False。

实例:

permissions = 'rw'

'w' in permissions #结果为True

**长度、最大值和最小值：**

内建函数len、min和max。

len函数返回序列中包含元素的数量

字符串不能被修改。

**列表:**

列表是可变的：可以改变列表的内容。

**list函数：**

因为字符串不能像列表一样被修改，所以可以用list函数根据字符串创建列表。

实例：

list('Hello') #结果是['H', 'e', 'l', 'l', 'o']

list函数适用于所有类型的序列，而不只是字符串。

**基本的列表操作：**

列表可以使用所有适用于序列的标准操作，例如索引、分片、连接和乘法。

**改变列表：元素赋值**

实例：

x[1] = 2

**删除元素：**

使用del语句来实现：

实例：

names = ['Alice', 'Beth', 'Cecil', 'Dee-Dee', 'Earl']

del names[2]

names #结果是['Alice', 'Beth', 'Dee-Dee', 'Earl']

**分片赋值：**

实例：

name = list('Perl')

name[2:] = list('ar')

name #结果是['P', 'e', 'a', 'r']

在分片赋值时，可以使用与原序列不等长的序列将分片替换。

分片赋值语句可以在不替换任何现有元素的情况下插入新的元素。

实例：

numbers = [1, 5]

numbers[1:1] = [2, 3, 4]

numbers #结果是[1, 2, 3, 4, 5]

通过替换进行删除元素也是可以的：

实例：

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

numbers[1:4] = []

numbers #结果是[1, 5]

**列表方法：**

append方法用于在列表末尾追加新的对象。直接在原列表的基础上进行修改。

count方法统计某个元素在列表中出现的次数：

实例：

['to', 'be', 'or' , 'not' , 'to', 'be'].count('to') #结果是2

extend方法可以在列表的末尾一次性追加另一个序列中的多个值。

实例：

a = [1, 2, 3]

b = [4, 5, 6]

a.extend(b) #此时a = [1, 2, 3, 4, 5, 6]

extend和连接操作的区别：extend方法修改了被扩展的序列，而原始的连接操作则会返回一个全新的列表。这里所说的连接操作就是 +

也可以使用分片来达到相同效果：

实例：

a = [1, 2, 3]

b = [4, 5, 6]

a[len(a):] = b #此时a = [1, 2, 3, 4, 5, 6]

index方法用于从列表中找出某个值第一个匹配项的索引位置。

实例：

knights = ['We', 'are', 'the', 'knights', 'who', 'say', 'ni']

knights.index('who') #结果为4

knights.index('herring') #将会引发一个异常

insert方法用于将对象插入到列表中，第一个参数即表示插入的位置。

实例：

numbers = [1, 2, 3, 4, 5, 6, 7]

numbers.insert(3, 'four') #此时numbers = [1, 2, 3, 'four', 5, 6, 7]

同样可以用分片来实现：

实例：

numbers = [1, 2, 3, 4, 5, 6, 7]

numbers[3:3] = ['four'] #此时numbers = [1, 2, 3, 'four', 5, 6, 7]

pop方法会移除列表中的一个元素(默认是最后一个)，并且返回该元素的值。

实例：

x = [1, 2, 3]

x.pop() #结果为3，此时x = [1, 2]

x.pop(0) #结果为1，此时x = [2]

remove方法用于移除列表中某个值得第一个匹配项。

实例：

x = ['to', 'be', 'or', 'not', 'to', 'be']

x.remove('be') #此时x = ['to', 'or', 'not', 'to', 'be']

如果对不存在的元素进行remove，将会引发一个异常。

remove是一个没有发挥之的原位置改变方法，他修改裂变却没有返回值。

reverse方法将列表中的元素反向存放。

reverse方法改变列表但是不返回值。和remove类似。

sort方法用于在原位置对列表进行排序。

sort方法改变原来列表但是不返回值。

sorted方法返回排序后的列表，原列表不改变。

实例：

x = [4, 6, 2, 1, 7, 9]

y = sorted(x) #此时x = [4, 6, 2, 1, 7, 9]，y = [1, 2, 4, 6, 7, 9]

高级排序：

如果希望元素能按照特定的方式排序，可以通过compare(x,y)的形式自定义比较函数。定义好compare函数之后，就可以提供给sort方法作为参数了。内建函数cmp提供了鼻尖函数的默认实现方式。

实例：

cmp(42, 32) #结果是1

cmp(99, 100) #结果是-1

numbers = [5, 2, 7, 9]

numbers.sort(cmp) #numbers = [2, 5, 7, 9]

sort方法有另外两个可选参数：key和reverse。如果要使用他们，那么就要通过名字来指定(这叫做关键字参数)。参数key与cmp类似，必须提供一个在排序过程中使用的函数，该函数并不是直接用来确定对象的大小，而是为每个元素创建一个键，然后所有元素根据这个键来排序。

实例：根据元素的长度进行排序，可以使用len作为键函数

x = ['aardvark', 'abalone', 'acme', 'add', 'aerate']

x.sort(key=len) #x = ['add', 'acme', 'aerate', 'abalone', 'aardvark']

另一个关键参数reverse是简单的布尔值，用来指明是否要进行反向排序。

cmp、key、reverse参数都可以用于sorted函数。

**元组：不可变序列**

元组也是序列，但是不能修改。

创建元组很简单，如果用逗号分隔一些值，就自动创建了元组。

实例：

1, 2, 3 #自动创建元组(1, 2, 3)

一个元素的元组：42, #(42,)逗号不可以省略。(42)就不是一个元组。

空元组：()

**tuple函数**

tuple函数以一个序列作为参数并把它转换为元组。如果参数就是元组，那么该参数就会被原样返回。

实例：

tuple([1, 2, 3]) #返回值是(1, 2, 3)

tuple('abc') #返回值是('a', 'b', 'c')

tuple((1, 2, 3)) #参数原样返回。

基本的元组操作

元组与列表的区别：

1.元组可以在映射中当作键来使用，而列表则不可以。

2.元组作为很多内建函数和方法的返回值存在，也就是说你必须对元组进行处理。只要不尝试修改元组，那么处理元组在绝大多数情况下就是把他们当作列表来进行操作。

**字符串:**

**精简的字符串格式化：使用百分号%来实现**

在%的左侧放置一个字符串(格式化字符串)，右侧放置希望格式化的值。可以使用一个值，如一个字符串或者数字，也可以使用多个值的元组或者字典。

实例：

format = "Hello, %s, %s enough for ya?"

values = ('world', 'Hot')

print format % values #结果：Hello, world, Hot enough for ya?

如果要格式化字符串里包含百分号，那么必须使用%%，如果要格式化实数(浮点数)，可以使用f说明符类型，同时提供所需的精度：一个句点再加上希望保留的小数位数

**模板字符串：**

string模块提供另一种格式化值得方法：模板字符串。如下，substitute这个模板会用传进来的关键字参数foo替换字符串中的$foo

实例：

form string import Template

s = Template('$x', glorious $x!')

s.substitute(x = 'slurm') #结果为'slurm, glorious slurm!'

如果是替换字段是单词的一部分，则参数名必须用括号括起来，从而明确指定结尾。

实例：

s = Template("It's ${x}tastic!")

s.substiture(x = 'slurm') #结果是"It's slurmtatic!"

可以使用$$插入美元符号。

**字符串格式化：完整版**

如果右操作数是元组的话，则其中的每一个元素都会被单独格式化，每个值都需要一个对应的转换说明符。

实例：

'%s plus %s equals %s' % (1, 1, 2) #结果是'1 plus 1 equals 2'

**字段宽度和精度**

字段宽度是转换后的值所保留的最小字符个数

精度：对于数字来说，是结果中应包含的小数的位数，对于字符串转换来说，是转换后的值所能包含的最大字符个数。

这两个参数都是正数，首先是字段宽度，然后是精度，通过点号(.)分隔。虽然两个都是可选参数，但是如果只给出精度，就必须包含点号。

可以使用\*(星号)作为字段宽度或者精度(或者两者都是用\*)，此时数值会从元组参数中读出。

实例：

'%.\*s' % (5, 'Guido van Rossum') #结果是'Guido'

'%.\*s -- %.\*s' % (5, 'Guido van Rossum', 5, 'Guido van Rossum') #结果是'Guido -- Guido'

**符号、对齐和0填充**

在字段宽度和精度之前还可以放置一个“标表”，该标表可以是零、加号、减号或空格。

零表示在宽度不够的情况下，将用0进行填充。

减号表示用来左对齐数值。

而空白(" ")意味着在正数前面加上空格。

实例：

print ('% 5d' % 10) + '\n' + ('% 5d' % -10)F

**字符串方法：**

字符串从string模块中继承了很多方法。

string模块中一些有用的字符串常量：

string.digits：包含数字0~9的字符串

string.letters：包含所有字母(大写和小写)的字符串

string.lowercase：包含所有小写字母的字符串

string.printable：包含所有可打印字符的字符串。结果是：

'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()\*+,-./:;<=>?@[\\]^\_`{|}~ \t\n\r\x0b\x0c'

string.punctuation：包含所有标点的字符串

string.uppercase：包含所有大写字母的字符串

字母字符串常量与地区有关。

find方法可以在一个较长的字符串中查找子字符串。它返回子串所在位置的最左端索引，如果没有找到，返回-1。

split方法将字符串分割成序列。如果不提供任何分隔符，程序会把所有空格作为分隔符。

实例：

'1+2+3+4'.split('+') #['1', '2', '3', '4']

join方法用来在队列中添加元素。需要添加的队列元素必须都是字符串。

实例：

seq = ['1', '2', '3']

seq.join('+') #seq = ['1+2+3']

seq = [1, 2, 3]

seq.join('+') #Error，因为要添加的队列元素必须都是字符串。

lower方法返回字符串的小写字母。

replace方法返回某些字符串的所有匹配项均被替换之后得到的字符串。

实例：

'This is a test'.repleace('is', 'eez') #'Theez eez a test'

translate方法可以替换字符串中的某些部分，但是只处理单个字符。参数不是简单的两个字符，具体查看帮助文档。

maketrans函数接受两个参数：两个等长的字符串，表示第一个字符串中的每个字符都用第二个字符串中相同位置的字符替换。转换后返回的是包含替换ASCII字符集中256个字符的替换字母的字符串。

strip方法会去除两侧(不包括内部)空格的字符串。也可以去除指定的字符，将他们作为参数即可。

实例：指定字符并不是当作字符串，而是单个字符，如下，并不是将\*!作为整体。

'\*\*\* SPAM \* for \* everyone!!! \*\*\*'.strip(' \*!') #'SPAM \* for \* everyone'

**字典：**

键可以是数字、字符串甚至是元组

创建字典：

实例：

phonebook = {'Alice' : '2341', 'Beth' : '9102', 'Cecil' : '3258'}

空字典：{}

dict函数通过其他映射(比如其他字典)或者(键，值)这样的序列对建立字典。并不是真正的函数。

实例：

items = [('name', 'Gumby'), ('age', 42)]

d = dict(items) #d = {'age' : 42, 'name' : 'Gumby'}

dict函数通过关键字创建字典:

实例：

d = dict(name = 'Gumby', age = 42) # d = {'age' : 42, 'name' : 'Gumby'}

**基本字典操作：**

字典的基本行为在很多方面与序列类似：

len(d)：返回d中项(键值对)的数量

d[k]：返回关联到键k上的值

d[k] = v：将值v关联到键k上

del d[k]：删除键为k的项

k in d：检查d中是否含有键为k的项

**字典的格式化字符串**

在每个转换说明符%后面加上用圆括号括起来的键，后面在跟上其他说明元素。

实例：

phonebook = {'Beth':'9102', 'Alice':'2341', 'Cecil':'3258'}

"Cecil's phone number is %(Cecil)s ." % phonebook #"Cecil's phone number is 3258"

此例中，%(Cecil)s中的s是表示打印的为字符串，不可省略。如下，当某个key对应的value为数值时：

实例：

phonebook = {'Beth':'9102', 'Alice':'2341', 'Cecil':'3258', 'Dnage':185186}

"Dange's phone number is %(Dange)d." % phonebook #"Dange's phone number is 185186"

这个地方不知道为什么不能把key设置为数字。

**字典方法：**

clear方法清除字典中所有项。这个是原地操作，无返回值。

实例：

x = {}

y = x

x['key'] = 'value' #y = {'key':'value'}

x = {} #y = {'key':'value'}

x.clear() #y = {}

如上例，如果要清空原始字典中的所有元素，则必须使用clear方法。

copy方法返回一个具有相同键值对的新字典。（这个方法实现的是浅复制，因为值本身就是相同的，而不是副本）

deepcopy方法复制包含所有的值。深复制。

浅复制和深复制的区别：

浅复制：只是复制父对象，不会复制对象内部的子对象。

深复制：复制父对象以及其子对象。

实例：

a = [1, 2, 3, ['a', 'b', 'c']]

b = copy.copy(a) #b = [1, 2, 3, ['a', 'b', 'c']]，但是b中子对象['a', 'b', 'c'] 应该是一个a[3]的引用，而没有被拷贝到b中。id(a)==id(b)为false，id(b[3])==id(a[3])为 true

c = copy.deepcopy(a) #c是a的完整拷贝。c = [1, 2, 3, ['a', 'b', 'c']]

b[0] = 8 #此时a、c都不变，b = [8, 2, 3, ['a', 'b', 'c']]

b[3].append('t') #此时a=[1, 2, 3, ['a', 'b', 'c', 't']]，b=[8, 2, 3, ['a', 'b', 'c', 't']]，c不变

fromkeys方法使用给定的键建立新的字典。每个键默认对应的值为None。

实例：

{}.fromkeys(['name', 'age']) #{'age':None, 'name':None}

可以直接使用dict类调用此方法：dict.fromkeys()

可以更改值得默认值：

实例：

dict.fromkeys(['name', 'age'], '(unknown)') #{'age':'(unknown)', 'name':'(unknown)'}

get方法：宽松的访问字典项的方法，所谓宽松，是指试图访问字典中不存在的项时不会会出错。访问不存在项时，默认返回None，也可以自定义默认值。

实例：

d = {}

d.get('name', 'N/A') # N/A

has\_key方法可以检查字典中是否有给出的键。表达式d.has\_key(k)相当于表达式k in d。Python3.0中咩有这个函数。

items方法将所有字典项以列表的方式返回，这些列表项中的每一项都来自于（键，值）。但是项在返回时并没有特殊的顺序。



iteritems方法返回一个迭代器对象而不是列表。

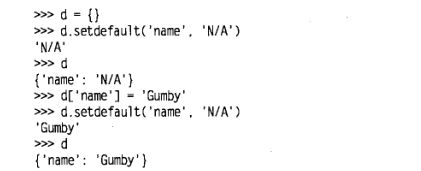
keys方法将字典中的键以列表形式返回。

iterkeys方法则返回针对键的迭代器。

pop方法用来获得对应于给定键的值，然后将这个键-值对从字典中移除。

popitem方法弹出字典中一个随机的项。字典中没有顺序，所有没有“最后的元素之说”。

setdefault方法能够获得与给定键相关联的值，并且可以在字典中不含有给定键的情况下设定相应的键值。



update方法可以利用一个字典项更新另一个字典，提供的字典中的项会被添加到旧的字典中，若有相同的键则会进行覆盖。

实例：

d= {

'title':'Python Web Site',

'url':'http://www.python.org',

'changed':'Mar 14 22:09:15 MET 2008'

}

x = {'title':'Python Language Website'}

d.update(x)

d

{'url':'http://www.python.ort', 'changed':'…..不变', 'title':'Python Language Website'}

values方法以列表的形式返回字典中的值。

itervalues方法返回值得迭代器。

**条件、循环和其他语句**

import导入时可以为模块或者函数起一个别名。但是此时只能有别名，而不能用实际的名字

实例：

import math as foobar

foobar.sqrt(4) #2.0

math.sqrt(4) #导致NameError：name "math" is not defined

**序列解包**

False None 0 '' () [] {} Python将会认为是假

bool函数可以转换其他值为bool值。

实例：

bool('test') #True

bool('') #False

**is同一性运算符**

比较的是同一性而不是是否相等。

实例:

x = y = [1,2]

z = [1,2]

x = z #True

y = z #True

x is y #True

x is z #False

布尔运算符：都是短路的运算符

and： a <= 10 and a >= 1 也可以使用 1 <= a <= 10

or

not

断言：

assert关键字，如果条件为真时，才让程序继续运行下去，否则引发Error

range函数产生某个范围的数字，不包含结尾。

实例：

range(0,4) #[0, 1, 2, 3]

range(3) #[0, 1, 2, 3]

zip函数可以把多个序列“压缩”在一起，然后返回一个元组的列表。当序列不等长时，最短的序列用完就会自动停止。

实例：

names = ['anne', 'beth', 'george', 'damon']

ages = [12, 45, 32, 102]

zip(names, ages) #[('anne', 12), ('beth', 45), ('george', 32), ('damon', 102)]

del方法删除对象：

del删除的只是对象的名称而不是内存中的值，用户没法删除，Python会自动回收。

实例：

x = y = 'a'

del x #此时x没有了定义，而y = 'a'

一个函数位于另一个里面，外层函数返回里层函数，也就是说函数本身被返回了--但并没有被调用。重要的是返回的函数还可以访问它的定义所在的作用域。

实例：

def multiplier(factor)

def mutiplyByFactor(number)

return number \* factor

return multiplyByFactor

每次调用外层函数，他内部的函数都被重新绑定，factor变量每次都有一个新的值。由于Python的嵌套作用域，来自（multiplier的）外部作用域的这个变量，稍后会被内层函数访问。

实例：

double = multiplier(2) #double为mutiplyByFactor(number)函数

double(5) #结果为 10

triple(3) #triple同为函数

rtiple(3) #结果为9

multiplier(5)(4) #结果为20

类似multipByFactor函数存储子封闭作用域的行为叫做闭包。

nonlocal关键字可以让用户对外部作用域(但并非全局作用域)的变量进行赋值。

标准库random中包含choice函数，可以从序列中随机选出元素。

lambda

实例：

a = [1, 2, 3]

b = [4, 5, 6]

t = map((lambda x,y:x\*y),zip(a,b)) #这种写法在py2系列是正确的，但是在3系列是错误的，因为zip返回的是元组的列表，本例中返回那就是[(1, 2), (3, 4), (5, 6)]，而解包时只把列表解包，元组不会再进行解包，即x = (1, 2)，y = (3, 4)…

3系列中正确写法：

t = map((lambda x:x[0]\*x[1]),zip(a,b)) 仍旧存在问题！！！！！！！！！！！不懂。

issubclass方法查看一个类是否是另一个类的子类。issubclass(class1, class2)，测试class1是否为class2的子类。

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

getattr函数获取对象的特性。getattr(tc, 'talk', None) #检查tc.talk是否可以调用，也可用于（获取对象的特性）如果没定义（是指对象特性不存在时，即使特性对象内容为空，默认值也不会起作用，一定是未定义的），可以在函数第三个参数是上提供默认值。

hasattr函数设置对象的特性。setattr(tc, 'name', 'Mr.Gumby') #tc.name = Mr.Gumby

在try except else finally语句中：

当try语句块中，如果没有发生异常，则else语句块中的语句才执行。

实例：

try:

1/1

except TabError:

print "over"

else:

print 'no'

finally:

print 'gg'

property函数：

实例：

\_\_metaclass\_\_ = type

class Rectangle:

def \_\_init\_\_(self):

self.width = 0

self.height = 0

def setSize(self,size):

self.width, self.height = size

def getSize(self):

return self.width, self.height

size = property(getSzie, setSize)

property函数创建了一个属性，其中访问器函数（指getSize()、setSize()函数）被用作参数（先是取值，然后是赋值）。这个属性名命名为size。

调用时：

r = Rectangle()

r.width = 10

r.height = 5

r.size #(10, 5)

r.size = 150, 100

r.width #150

r.height #100

size特性仍然取决于getSize和setSize中的计算，但是他们看起来就像普通的属性一样。

property函数可以用0，1，2，3或者4个参数来调用。如果没有参数，产生的属性即不可读也不可写。如果只使用一个参数调用，（一个取值方法，不可是设值方法），产生的属性是只读。第三个参数（可选）是一个用于删除特性的方法（不需要参数）。第四个参数（可选）是一个文档字符串。property的4个参数分别叫做fget，fset，fdel和doc--。

静态方法和类成员方法：

静态方法和类成员方法在创建时分别被装入staticmethod和classmethod类型的对象中。

静态方法的定义没有使用self参数，且能够被类本身直接调用。

类方法在定义时需要名为cls的类似于self的参数，类成员方法可以直接用类的具体对象直接调用，但是cls参数是自动被绑定到类的。

实例：

\_\_metaclass\_\_ = type

class MyClass:  
 def smeth():

print ‘This is a static method‘

smeth = staticmethod(smeth)

def cmeth(cls):

print ‘This is a class method of’ . cls

cmeth = classmethod(cmeth)

新出的方法：

\_\_metaclass\_\_ = type

class MyClass:

@staticmethod

def smeth():

print ‘This is a static method‘

@classmethod

def cmeth(cls):

print ‘This is a class method of’ . cls

\_\_iter\_\_方法返回一个迭代器，所谓迭代器就是具有next方法（这个方法在调用时不需要任何参数）的对象。在调用next方法时，迭代器会返回它的下一个值。如果next方法被调用，但是迭代器没有值可返回，就会引发一个StopIteration异常。

在Python中，迭代器应该实现一个\_\_next\_\_方法，而不是next。而新的内建函数next可以用于访问这个方法。Python3.0中next(it)等同于之前版本的it.next()

实例：

class Fibs:

def \_\_init\_\_(self):

self.a = 0

self.b = 1

def next(self):

self.a, self.b = self.b, self.a + self.b

return self.a

def \_\_iter\_\_(self):

return self

fibs = Fibs()

for f in fibs:

if f > 1000:

print f

break

print ‘fibs.next() =’,fibs.next()

生成器是一种普通的函数语法定义的迭代器。

实例：

def flatten(nested):

for sublist in nested:

for element in sublist:

yield element

任何包含yield语句的函数被称为生成器。除了名字不同以外，他的行为和普通的函数也有很大的差别，这就在于他不是像return那样返回值，而是每次产生多个值，每次产生一个值（使用yield），函数就会被冻结：即函数停在那点等待被激活。函数被激活后就从停止的那点开始执行。

递归生成器：

实例：

def flatten(nested):

try:

for sublist in nested:

for element in flatten(sublist):

yield element

except TypeError:

yield nested

内建字符串str中的strip、lstrip、rstrip函数：

strip用于去除字符串首位的字符，同理，lstrip去除左边的字符串，rstrip。。

如果不传入参数，或者参数为None，则默认去除空格。当传入参数时（字符串），指去除指定位置的字符（而不是字符串）。

实例：

‘ aa ’.strip() # ‘aa’

‘mississipip’.strip(‘ipz’) #’mississ’ 由此可看出去掉时是按字符来查找的，而不是字符串。

‘mississipip’.strip(‘zpi’) #‘mississ’ 可以看出查找时每去除以个字符后，其他字符会再循环。

类变量与实例变量：

实例：

#!/usr/bin/python

#coding: utf-8

#测试类变量与实例变量

class test:

name = 'test'

val = 5

sum = 0

def \_\_init\_\_(self, val, sum):

self.val = val + 10

self.sum = sum + 100

def te(self):

print self.val

print self.sum

print self.name

T = test(1, 5)

T.te()

print test.sum

内建函数reduce：

内建函数reduce是一个二元函数，他用来将一个数据集合（元组，set，序列等）中的数据进行下列操作：用传给reduce中的函数func()（必须是一个二元操作函数）先对集合中的第1,2个数据进行操作，得到的结果再与第三个数据用func()函数运算，一次类推，最后得到一个结果。

实例：

m = []

for i in range(10):

m.append(set(range(i, i+5)))

reduce(set.union,m)

其中m = [set([0, 1, 2, 3, 4]), set([1, 2, 3, 4, 5]), set([2, 3, 4, 5, 6]), set([3, 4, 5,6, 7]), set([8, 4, 5, 6, 7]), set([8, 9, 5, 6, 7]), set([8, 9, 10, 6, 7]), set([8, 9, 10, 11, 7]), set([8, 9, 10, 11, 12]), set([9, 10, 11, 12, 13])]

得到结果为set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13])

set集合：

set集合2.4版本之后是Python内建类型，可直接使用。

set集合元素无序且不可重复。元素数量可以变，增加删除等，但是每个元素本身应该是不可变得。因此集合不可作为集合的元素。

实例：

a = set()

b = set()

a.add(b) #错误！！！！

使用frozenset类型的集合：frozenset是一个类。

a.add(forzenset(b)) #没毛病

模块heapq：包含一些堆操作的函数。

Python没有堆类型，由其他类型构成，堆中的元素顺序不是很随意。

堆属性：位于i位置上的元素总比i//2位置处的元素大（反过来说就是i位置处的元素总比2\*i以及2\*i+1位置处的元素小）。

heappop函数弹出堆中最小的元素。

heappush

heapify函数使用任意列表作为参数，并通过尽可能少的移位操作将其转换为合法的堆（也就是满足堆属性）

heapreplace函数弹出堆中的最小元素，并将新元素放入堆中。

nlatgest(n,iter)和nsmallest(n,iter)分别用来寻找可迭代对象Iterator中第n大或第n小元素。

双端队列（Double-ended queue，或称deque）：

双端队列存在于collections模块中。

time

time模块。

random模块中的shuffle函数原地修改传进的参数，返回None。

shelve模块

函数open：在调用它时（使用文件名作为参数，文件可不存在）它会返回一个shelf对象。可以用他来存储内容。需要把它当作普通的字典（但是键一定要作为字符串）来操作即可。

re模块：

包含一些有用的操作正则表达式的函数。

complile函数将正则表达式（以字符串书写的）转换为模式对象，可以实现更有效率的匹配。如果在调用search或者match函数的时候使用字符串表示的正则表达式，他们也会在内部将字符串转换为正则表达式对象。转换后的的模式对象本身也有匹配/查找的函数，就像方法一样，所以re.search(pat,string)（pat是用字符串表示的正则表达式）等价于pat.search(string)（pat是用compile创建的模式对象）。经过compile转换的正则表达式对象也能用于普通的re函数。

函数re.search一旦找打子字符串，函数就会返回MatchObject（值为True），否则返回None（值为False）。

实例：

import re

t = re.compile(“test”)

if t.search(“fdafj;ltestffsd”):

print ‘Found it!’

re.match函数会在给定的字符串的开头匹配正则表达式。

re.split函数会根据模式的匹配项来分割字符串。允许使用任意长度的逗号和空格序列来分割字符串。如果模式包含小括号，那么括起来的字符组合就会散布在分割后的字符串之间。例如：re.split(‘o(o)’,foobar)会生成[‘f’, ‘o’, ‘bar’]

实例：

some\_text = ‘alpha, beta,,,,gamma delta’

re.split(‘[, ]+’, some\_text, maxsplit=2) #maxsplit参数表示字符串最多分割成几部分(m+1)。

[‘alpha’, ‘beta’, ‘gamma delta’]

函数re.findall以列表形式返回给定模式的所有项。

实例：子字符串中查找所有的单词

pat = ‘[a-zA-Z]+’

text = ‘”Hm… Err – are you sure?” he said, sounding insecure.’

re.findall(pat, text)

['H', 'm', 'E', 'r', 'r', 'C', 'a', 'r', 'e', 'y', 'o', 'u', 's', 'u', 'r', 'e','h', 'e', 's', 'a', 'i', 'd', 's', 'o', 'u', 'n', 'd', 'i', 'n', 'g', 'i', 'n','s', 'e', 'c', 'u', 'r', 'e']

re.sub函数：使用给定的替换内容将匹配模式的子字符串（最左端并且非重叠的子字符串）替换掉。

实例：

pat = {name}

text = “Dear {name}…”

re.sub(pat, ‘Mr. Gumby’, text) #结果’Dear Mr. Gumby …’

re.escape函数：对字符串中所有可能被截石位正则运算符的字符进行转义。

实例：

re.escape('http://www.baidu.com')

'http\\:\\/\\/www\\.baidu\\.com'

让正则表达式变得更加易读的方式是在re函数中使用VERBOSE标志。它允许在模式中添加空白（空白字符，tab，换行等等），re则会忽略他们。除非将其放在字符类或者用反斜线转义。

实例：

re.compile(r’’’

\\* #....

( #.....

[^\\*]+ #。。

‘’‘,re.VERBOSE)

**文件操作：**

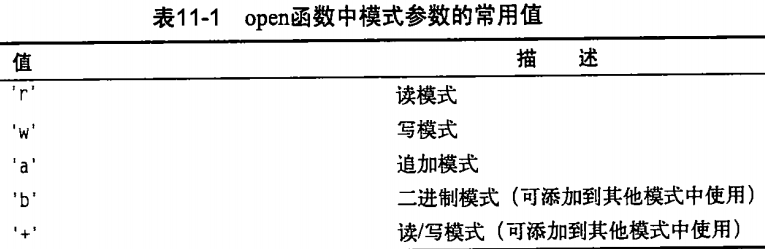
open函数用来打开文件：返回一个文件对象。

open(name[,mode[, buffering]]) 模式mode和缓冲(buffering)参数都是可选的。

缓冲参数是0或者False时，无缓冲，直接对硬盘进行操作。如果是1或者True，进行缓冲，只有使用flush或者close时才更新硬盘上的数据。大于1的数字代表缓冲区的大小。-1或者是任何负数，代表使用默认的缓冲区大小。

默认的模式是只读模式。

’+‘参数可以用到其他任何模式中，指明度和写都是允许的。



读写：

write和read方法：  
write以追加的方式写入文件。

seek(offset[,whence])：把当前位置（进行读写的位置）移动到由offest定义的位置，whence默认是0，表示偏移量offest是从文件开头开始计算的，whence设置为1时表示偏移量相对于当前位置进行移动，whence设置为2时表示相对于文件尾进行移动。offest可以设置为负数，表示根据当前位置向前移动。

tell方法可以返回当前文件的读写位置。

文件对象可迭代：

实例：

f = open(filename)

for x in f:

print x

为了可以让文件自动关闭，可以使用with语句：

with open(‘somefile’) as somefile:

do\_something(somefile)

with语句可以打开文件并将其赋值到变脸上，之后就可以将数据写入语句体中的文件（或许执行其他操作），文件在语句结束后会被自动关闭。在Python2.5以及之前版本，需要导入模块才能使用：from \_\_feature\_\_ import with\_statement

os.linesep决定了换行符是什么。