编码风格

使用 4 空格缩进，而非 TAB。在小缩进（可以嵌套更深）和大缩进（更易读）之间，4 空格是一个很好的折中。TAB 引发了一些混乱，最好弃用。

折行以确保其不会超过 79 个字符。这有助于小显示器用户阅读，也可以让大显示器能并排显示几个代码文件。

使用空行分隔函数和类，以及函数中的大块代码。

可能的话，注释独占一行。

使用文档字符串。

把空格放到操作符两边，以及逗号后面，但是括号里侧不加空格：

```
a = f(1,2) + g(3,4)
```

统一函数和类命名。总是用 self 作为方法的第一个参数。

字符串

利用三引号（‘‘‘）或（”””）可以指示一个多行的字符串，在里面可以随便使用单引号和双引号。它也是文档字符串 DocStrings

```
>>> def doc():
        '''Hi
         Hi'''
        pass
>>> print doc.__doc__
Hi
         Hi
```

>>>

文档字符串第一行应该是关于对象用途的简介。如果文档字符串有多行，第二行应该空出来，与接下来的详细描述明确分离。

转义符：'what\'s up' 等价于"what's up"

\\表示反斜杠本身

行末单独的一个\表示在下一行继续，而不是新的一行：

'aaaaa\

dddd' 等价于'aaaaadddd'

原始字符串 r

如果我们生成一个“原始”字符串，\n 序列不会被转义，而且行尾的反斜杠，源码中的换行符，都成为字符串中的一部分数据。

```
>>> hello = r"This is a rather long string containing\n\
serveral lines of text much as you would do in C."
>>> print hello
This is a rather long string containing\n\
serveral lines of text much as you would do in C.
>>> hello = "This is a rather long string containing\n\
serveral lines of text much as you would do in C."
>>> print hello
This is a rather long string containing
serveral lines of text much as you would do in C
```

原始字符串的最后一个字符不能是“\”，如果想要让字符串以单“\”结尾，可

以这样：

```
>>> print r'ee''\\'
ee\
```

如果是 print r'ee\'则会返回错误；如果是 print r'ee\\'则会返回 ee\\

字符串可以由 + 操作符连接，可以由 * 操作符重复。

相邻的两个字符串文本自动连接在一起，它只用于两个字符串文本，不能用于字符串表达式。

```
>>> 'str      '.strip()
'str'
>>> 'str' 'jkl'
'strjkl'
>>> 'str       '.strip() 'jkl'
SyntaxError: invalid syntax
>>> 'str     '.strip() + 'jjj'
'strjjj'
```

字符串不可变，向字符串文本的某一个索引赋值会引发错误。不过，组合文本内容生成一个新文本简单而高效。

```
>>> word = 'thank'
>>> word[0]
't'
>>> word[0] = 'f'
Traceback (most recent call last):
```

```
  File "<pyshell#10>", line 1, in <module>
    word[0] = 'f'
TypeError: 'str' object does not support item assignment
>>> word[ :4] + 'g'
'thang'
```

切片操作有个有用的不变性：i[:s] + i[s:] 等于 i。

```
>>> word[:4] + word[4:]
'thank'
```

值被转换为字符串的两种机制：str、repr

str 函数：把值转换为合理形式的字符串

repr：创建一个字符串，以合法的 Python 表达式的形式来表示值

```
>>> print repr('hello,world!')
'hello,world!'
>>> print str('hello,world!')
hello,world!
>>> print repr('1000L')
'1000L'
>>> print str('1000L')
1000L
```

input 与 raw_input:

```
>>> raw_input('e:')
e:a
```

```
'a'
>>> input('e:')
e:a

Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    input('e:')
  File "<string>", line 1, in <module>
NameError: name 'a' is not defined
>>> input('e:')
e:'a'
'a'
>>> a = raw_input('e:')
e:123
>>> type(a)
<type 'str'>
>>> b = input('e:')
e:123.01
>>> type(b)
<type 'float'>
>>> raw_input('e:')
e:1 + 2
```

```
'1 + 2'
>>> input('e:')
e:1 + 2
3
```

由上面的例子可以知道，两个函数均能接收字符串，区别是 raw_input()直接读取控制台的输入（任何类型的输入它都可以接收），而 input()在接收字符串的时候需要加上引号，否则会引发错误。此外，在接收纯数字方面，input()具有自己的特性，并不是像 raw_input()一样把所有输入都当做字符串看待。除此之外，input()会接收合法的 python 表达式 1 + 2 返回 int 型的 3。

字符串大小写变换：

lower() 小写 upper() 大写

swapcase() 大小写互换 capitalize() 首字符大写

title()只有首字符大写，其余为小写。用于转换字符串为标题：

```
>>> 'that\'s all,folks'.title()
"That'S All,Folks"
```

string 模块的 capwords 函数：

```
>>> import string
>>> string.capwords('that\'s all,folks')
"That's All,folks"
```

字符串在输出时候的对齐：

ljust(width,[fillchar])：左对齐，输出 width 个字符，不足部分用 filchar 补足，默认的为空格。

rjust（width，[fillchar]）：右对齐。

center(width,[fillchar])：中间对齐。

zfill(width)：把字符串变成 width 长，并在右对齐，不足部分用 0 补足。

```
>>> string.capwords('what\'s all,forks.').ljust(20)
"What's All,forks.      "
>>> string.capwords('what\'s all,forks.').rjust(20)
"      What's All,forks."
>>> string.capwords('what\'s all,forks.').center(40)
"                  What's All,forks.                   "
>>> string.capwords('what\'s all,forks.').zfill(20)
"000What's All,forks."
```

字符串中的搜索和替换及其他常用方法：

find(subsrt,[start,[end]])：返回字符串中第一个出现 substr 的第一个字母的标号，如果没有 subser 则返回-1。start 和 end 的作用相当于在[start:end]中搜索。

```
>>> 'srrrtripoopp'.find('trip',-1,-4)
-1
>>> 'srrrtripoopp'.find('trip')
4
```

index(subsrt,[start,[end]])：与 find 一样，只是在字符串中没有 substr 时，会返回一个运行时错误。

rfind(subsrt,[start,[end]])：返回字符串中最后出现 substr 的第一个字母的标号，如果字符串中没有 substr，返回-1.

rindex(subsrt,[start,[end]])。

count(subsrt,[start,[end]])：计算 substr 在字符串中出现的次数。

replace(oldstr,newstr,[count])：把字符串中的 oldstr 替换为 newstr，count 为替换的次数。

strip([char])：把字符串中前后有 char 的字符全部去掉，默认去掉空格：

```
>>> '          aa    bb        '.strip()
'aa    bb'
>>> '*** aaa * bbb * !***'.strip('*')
' aaa * bbb * !'
```

当然还有 lstrip([char])、rstrip([char])：

```
>>> 'tssssssttt'.lstrip('t')
'sssssttt'
>>> 'tssssssttt'.rstrip('t')
'tsssss'
>>>
```

join 方法用来在队列中添加元素，但需要添加的队列元素都必须是字符串：

```
>>> seq = [1,2,3,4,5]
>>> sep = '+'
>>> sep.join(seq)   #连接数字列表

Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
```

```
    sep.join(seq)
TypeError: sequence item 0: expected string, int found
>>> seq = ['1','2','3','4','5']    #连接字符串列表
>>> sep.join(seq)
'1+2+3+4+5'
>>> dirs = '','usr','bin','eno'
>>> '/'.join(dirs)
'/usr/bin/eno'
>>> print 'C:' + '\'.join(dirs)    #注意\的转义问题
SyntaxError: EOL while scanning string literal
>>> print 'C:' + '\\'.join(dirs)
C:\usr\bin\eno
```

split 方法用来将字符串分割成序列：

```
>>> '/usr/bin/env'.split('/')
['', 'usr', 'bin', 'env']
```

如果不提供任何分隔符，程序会把所有空格作为分隔符：

```
>>> 'Using the default'.split()
['Using', 'the', 'default']
```

translate 方法也是替换字符串中的的某些部分，但只处理单个字符。它的优势在于可以同时进行多个替换，有些时候比 replace 效率高的多。使用 translate 转换之前，需要完成一张转换表，通过 string 模块里的 maketrans 函数完成。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的相同位置的字符替换：

```
>>> from string import maketrans
>>> table = maketrans('cs','kz')
>>> len(table)
256 #转换表是包含替换 ASCII 字符集中 256 个字符的替换字符的字符串
>>> table[97:123]
'abkdefghijklmnopqrztuvwxyz'
```

translate 的第二个参数是可选的，用来指定要删除的字符：

```
>>> 'kz cdds dffg'.translate(table,'f')
'kz kddz dg'
```

字符串格式化：

%操作符左侧放置一个需要格式化的字符串，这个字符串带有一个或多个嵌入的转换目标，都可以以%开始。如果需要在格式化字符串里面包括百分号，那么必须使用%%。

%右边放置一个或者多个对象，这些对象将会插入到左边想进行格式化字符串的一个或者多个转换目标位置上。如果要格式化实数（浮点数），可以使用 f 说明符类型，同时提供所需要的精度：一个句点再加上希望所保留的小数位数。因为格式化说明符总是一表示类型的字符结束，所以精度应该放在类型字符前面：%.3f。

基本的转化说明符：

1．%字符：标记转换说明符的开始

2．转换标志：-（减号）表示左对齐；+表示在转换值之前要加上正负号；“”（空白字符）表示正数之前保留空格；0 表示转换值若位数不够则用 0 补充

3．最小字段宽度：转化后的字符串至少应该具有该值指定的宽度，如果是*，则宽度会从值元组中读出

4．点（.）后跟精度值：如果转换的是实数，精度值就表示出现在小数点后的位数。如果转换的是字符串，那么该数字就是表示最大字段宽度。如果是*，那么精度就会从元组中读出

字符串格式化转换类型：

c：字符　　f，F：十进制浮点数

d，i：带符号的十进制整数　o：不带符号的八进制

n：不带符号的十进制　C：单字符（接受整数或者单字符字符串）

x：不带符号的十六进制（小写）　X：不带符号的十六进制（大写）

e：科学计数法表示的浮点数（小写）　E：科学计数法表示的浮点数（大写）

g：如果指数大于-4 或者小于精度值则和 e 相同，其他情况与 f 相同

G：如果指数大于-4 或者小于精度值则和 E 相同，其他情况与 F 相同

r：字符串（使用 repr 转换任意 Python 对象）

s：字符串（使用 str 转换任意 Python 对象）

一些简单的转换：

```
>>> 'Price of eggs:$%d' % 42
'Price of eggs:$42'
>>> 'Hexadecimal price of eggs:%x' % 42
'Hexadecimal price of eggs:2a'
>>> from math import pi
>>> 'Pi:%f...' % pi
```

```
'Pi:3.141593...'
>>> 'Very inexact estimate of pi:%i' % pi
'Very inexact estimate of pi:3'
>>> 'Using str:%s' % 42L
'Using str:42'
>>> 'Using repr:%r' % 42L
'Using repr:42L'
```

字段宽度和精度：

这两个参数都是整数（首先是字段宽度，然后是精度），通过点号（.）分隔：

```
>>> '%10f' % pi
'  3.141593'
>>> '%10.2f' % pi
'      3.14'
>>> '%.2f' % pi
'3.14'
>>> '%.5s' % 'Guido can Rossum'
'Guido'
```

可以使用*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：

```
>>> '%*.*s' % (10,5,'Guido ddds aaa')
'     Guido'
```

符号，对齐和 0 填充：

```
>>> '%010.2f' % pi
'0000003.14'
>>> '%-10.2f' % pi
'3.14
>>> print '%5d' % 10 + '\n' + '%5d' % -10   #用来对齐正负数
   10
  -10
>>> print ('%+5d' % 10) + '\n' + ('%+5d' % -10)
  +10
  -10
```

标示符第一个字母只能是字母或者下划线，其他部分可以有数字，标示符对字母大小写是敏感的。

比较字符串

使用 cmp()函数，当两个字符串一样的时候，返回 0。当 s 是 t 的一部分的时候 cmp(s,t)返回-1，相反 s 长于 t 的时候返回 1。也可以对指定的长度比较 cmp(s[m:n],t[z,p])。

模板字符串

string 模块提供另外一种格式化值的方法：模板字符串。substitute 这个模板方法会用传递进来的关键字参数 foo 替换字符串中的$foo：

```
>>> from string import Template
>>> s = Template('$x,glorious $x!')
>>> s.substitute(x='slurm')
```

'slurm,glorious slurm!'

如果替换字段是单词的一部分，那么参数名就必须使用括号括起来，从而准确指明结尾：

```
>>> s = Template('It\'s ${x}tastic!')
>>> s.substitute(x='slurm')
"It's slurmtastic!"
```

可以使用$$插入美元符号。

除了关键字参数外，还可以使用字典变量提供键/值对：

```
>>> s = Template('A $thing must never $action.')
>>> d = {}
>>> d['thing'] = 'gentleman'
>>> d['action'] = 'show his socks'
>>> s.substitute(d)
'A gentleman must never show his socks.'
```

数据结构

数据结构基本上就是用来存储一组相关数据的。Python 有一种名为容器的数据结构，容器基本上是包含其他对象的任意对象。序列（Python 有 6 种内建序列：列表、元组、字符串、Unicode 字符串、buffer 对象、xrange 对象）和映射（例如字典）是两类主要容器。序列中的每个元素都有自己的编号，而映射中的每个元素则有一个名字（也称为键）。既不是序列也不是映射的容器类型，集合（set）就是个例子。

列表

list 是处理一组有序项目的数据结构，列表是可变的数据类型（可以修改元素），支持原处修改对象操作。列表的元素不必是同一类型。但列表和字符串是无法连接在一起的，两种相同类型的序列才能进行连接操作。

索引与分片的赋值是原地进行的，是对列表的直接修改，而不是产生新的列表作为结果。

列表方法：

append(x)：把一个元素添加到列表的结尾

extend(L)：将 L 中的所有元素添加到对象列表中

insert(i, x)：在指定位置插入一个元素

remove(x)：删除列表中值为 x 的第一个元素，如果没有这样的元素，就会返回一个错误。

pop()从列表中删除指定位置的元素，并将其返回。a.pop()默认返回（删除）最后一个元素。

index(x)返回列表中第一个值为 x 的元素的索引，如果没有匹配的元素就会返回一个错误。

sort()对列表中的元素就地进行排序（直接修改了调用它的对象）。

reverse()就地倒排列表中的元素。

count(x)返回 x 在链表中出现的次数。

sort 方法详解：

```
>>> x = [4,6,2,1,7,9]
>>> x.sort()
>>> x
```

```
[1, 2, 4, 6, 7, 9]
```

为了得到一个排好序的副本，且保持原列表不变，下面的方法是错误的：

```
>>> y = x.sort()
>>> print y
None
```

因为 x.sort()返回的是 None，并不是一个排序好的列表，sort()方法只是进行了“原地”操作。正确的做法是：

```
>>> x = [4,6,2,1,7,9]
>>> y = x[:]
>>> y.sort()
>>> x
[4, 6, 2, 1, 7, 9]
>>> y
[1, 2, 4, 6, 7, 9]
```

只是简单的把 x 赋值给 y 是没用的，因为这样就让 x、y 指向了同一个列表了：

```
>>> y = x
>>> y.sort()
>>> y
[1, 2, 4, 6, 7, 9]
>>> x
[1, 2, 4, 6, 7, 9]
```

另一种获取已排序的列表副本的方法是，sorted 函数：

```
>>> x = [4,6,2,1,7,9]
>>> y = sorted(x)
>>> x
[4, 6, 2, 1, 7, 9]
>>> y
[1, 2, 4, 6, 7, 9]
```

这个函数可以用于任何序列，却总是返回一个列表：

```
>>> sorted('Python')
['P', 'h', 'n', 'o', 't', 'y']
>>> sorted('2351')
['1', '2', '3', '5']
```

高级排序

cmp、key、reverse 参数都可用于 sort()函数：

1. cmp 参数：

```
>>> n = [5,2,9,7]
>>> n.sort(cmp)
>>> n
[2, 5, 7, 9]
```

2. key 参数：

```
>>> x = ['aaaawt','ert','j']
>>> x.sort(key=len)
>>> x
```

['j', 'ert', 'aaaawt']

3. reverse 参数：

```
>>> x = [4,6,2,1,7,9]
>>> x.sort(reverse=True)
>>> x
[9, 7, 6, 4, 2, 1]
```

双端队列：把列表当队列使用（先进先出）：

双端队列（Double-ended queue 或称 deque）在需要按元素增加的顺序来移除元素时非常有用。collections 模块中的 deque 类型（它为在首尾两端快速插入和删除而设计）。双端队列通过可迭代对象（比如集合）创建，而且有些非常用用的方法，如下所示：

```
>>> from collections import deque
>>> q = deque(range(5))
>>> q.append(5)
>>> q.appendleft(6)
>>> q
deque([6, 0, 1, 2, 3, 4, 5])
>>> q.pop()
5
>>> q.popleft()
6
>>> q.rotate(3)    #右移 3 次
```

```
>>> q
deque([2, 3, 4, 0, 1])
>>> q.rotate(-1)    #左移一次
>>> q
deque([3, 4, 0, 1, 2])
```

双端队列好用的原因是它能有效地在开头（左侧）增加和弹出元素，这是在列表中无法实现的。除此之外，它还能够有效地旋转（rotate）元素（也就是将它们左移或右移，使头尾相连）。

对于列表来讲，有三个内置函数非常有用：fileter,map,reduce

filter(function, sequence)返回一个序列，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）。如果该序列是一个字符串或者元组，它返回值必定是同一类型，否侧，它总是 list。例如，以下程序可以计算部分素数：

```
>>> def f(x):
        return x % 2 !=0 and x %3 !=0
>>> filter(f,range(2,25))
[5, 7, 11, 13, 17, 19, 23]
```

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：

```
>>> def cube(x):return x * x * x
>>> map(cube,range(1,11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

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。例如：

```
>>> s = ['a','b','c','d']
>>> t = ['a','b','c','l']
>>> d = ['a','b','c','j']
>>> map(add,s,t,d)
['aaa', 'bbb', 'ccc', 'dlj']
>>> map(None,s,t,d)
[('a', 'a', 'a'), ('b', 'b', 'b'), ('c', 'c', 'c'), ('d', 'l', 'j')]
```

reduce(func, 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：

```
>>> def add(x,y):return x + y
>>> reduce(add,range(1,101))
5050
```

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。

列表推导式：

列表推导式提供了一个创建链表的简单途径，无需使用 map()，filter()以及 lambda。以定义方式得到列表通常要比使用构造函数创建这些列表更清晰。每一个列表推导式包括在一个 for 语句之后的表达式，零或多个 for 或 if 语句。返回值是由 for 或 if 子句之后的表达式得到的元素组成的列表。如果想要得到一个元组，必须要加上括号。

```
>>> freshfruit = ['
```

```
banana', '
loganberry ', 'passion fruit        ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
>>> [x, x**2 for x in vec]
# error - parens required for tuples
File "<stdin>", line 1, in ?
[x, x**2 for x in vec]
^
SyntaxError: invalid syntax
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
```

```
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

嵌套的列表推导式：

考虑以下 3X3 矩阵的例子，一个列表包含了三个列表，每个一行：

```
>>> mat = [
 [1, 2, 3],
 [4, 5, 6],
 [7, 8, 9],  ]
```

现在，如果你想交换行和列，可以用列表推导式：

```
>>> print [[row[i] for row in mat] for i in [0, 1, 2]]
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

为了不被嵌套的列表推导式搞晕，从右往左读。接下来有一个更容易读的版本：

```
>>> for i in [0,1,2]:
        for raw in mat:
            print raw[i],
        print
```

实用中，你可以利用内置函数完成复杂的流程语句。函数 zip() 在这个例子中可以

搞定大量的工作：

```
>>> zip(*mat)
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

删除语句：

有个方法可以从列表中按给定的索引而不是值来删除一个子项：del 语句。它不同于有返回值的 pop()方法。语句 del 还可以从列表中删除切片或清空整个列表。例如：

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

del 也可以删除整个变量：

```
>>> del a #此后再引用命名 a 会引发错误（直到另一个值赋给它为止）。
```

当两个变量同时引用一个列表的时候：

它们的确是同时引用了同一个列表。如果想避免出现这种情况，可以复制一个列表的副本。当在序列中做切片的时候，返回的切片总是一个副本。因此，如果你复制了整个列表的切片，将会达到一个副本：

```
>>> names = ['Mrs.Entity','Mrs.Thing']
>>> n = names
>>> n is names
True
>>> n == names
True
>>> n[0] = 'Mr.D'
>>> n
['Mr.D', 'Mrs.Thing']
>>> names
['Mr.D', 'Mrs.Thing']
>>> m = names[:]
>>> m is names
False
>>> m == names
True
>>> m[0] = 'Ms.P'
>>> m
['Ms.P', 'Mrs.Thing']
>>> names
['Mr.D', 'Mrs.Thing']
>>> n
```

['Mr.D', 'Mrs.Thing']

y = x[:]与 y = x 不同，前者是得到完全复制 x 后的另一个列表 y，后者是 x 与 y 指的同一个列表。

元组

元组通常用在使语句或者用户定义的函数能够安全的采用一组值的时候，即被使用的元祖的值不会改变。元组也是一个序列。一个元组由数个逗号分隔的值组成，例如：

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>>u = t, (1, 2, 3, 4, 5)  # 元组可以嵌套
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

tuple 函数：以一个序列作为参数并把它转换为元组。

```
>>> tuple([1,2,3])
(1, 2, 3)
>>> tuple('abs')
('a', 'b', 's')
>>> tuple((1,2,3))
(1, 2, 3)
```

元组在输出时总是有括号的，以便于正确表达嵌套结构。在输入时可以没有括号，不过经常括号都是必须的（如果元组是一个更大的表达式的一部分）。元组就像字符串，不可改变：不能给元组的一个独立的元素赋值（尽管你可以通过联接和切割来模拟）。还可以创建包含可变对象的元组，例如列表。

一个特殊的问题是构造包含零个或一个元素的元组：为了适应这种情况，语法上有一些额外的改变。一对空的括号可以创建空元组；要创建一个单元素元组可以在值后面跟一个逗号（在括号中放入一个单值不够明确），丑陋，但是有效。例如：

```
>>> empty = ()
>>> singleton = 'hello',
>>> singleton
('hello',)
```

语句 t = 12345, 54321, 'hello!'是元组封装（tuple packing）的一个例子：值 12345，54321 和'hello!'被封装进元组。其逆操作可能是这样：

```
x, y, z = t
```

这个调用等号右边可以是任何线性序列，称之为“序列拆封”非常恰当。“序列拆封”要求左侧的变量数目与序列的元素个数相同。要注意的是可变参数(multiple assignment）其实只是元组封装和序列拆封的一个结合。

序列

列表和元组都是序列。序列的两个主要特点是可以进行索引操作（从序列中抓取一个特定的项目）和切片操作（序列的一部分）以及加、乘、成员资格。len()、max()、min()也可以用于序列。

list[:]返回整个序列的拷贝；list[::-1]则得到一个反转的序列。

在序列中循环时，索引位置和对应值可以使用 enumerate() 函数同时得 到。

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
        print i, v
0 tic
1 tac
2 toe
```

同时循环两个或更多的序列，可以使用 zip() 整体打包。

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
        print 'What is your {0}?It is {1}.'.format(q, a)
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.
```

需要逆向循环序列的话，先正向定位序列，然后调用 reversed() 函数：

```
>>> for i in reversed(range(1,10,2)):
        print i,
9 7 5 3 1
```

要按排序后的顺序循环序列的话，使用 sorted() 函数，它不改动原序列，而是生成一个新的已排序的序列：

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
```

print f

apple

banana

orange

pear

字典

只能使用不可变的对象（例如字符串）来作为字典的键，但可以把可变或者不可变的对象作为字典的值。基本上来说，你应该使用简单的对象作为键。如：d = {key1:value1,key2:value2},键值对是没有顺序的。

序列是以连续的整数为索引，与此不同的是，字典以关键字为索引，关键字可以是任意不可变类型，通常用字符串或数值。如果元组中只包含字符串和数字，它可以做为关键字，如果它直接或间接的包含了可变对象，就不能当做关键字。不能用列表做关键字，因为链表可以用索引、切割或者 append()和 extend() 等方法改变。可以把字典看做无序的“键值”对（key:value pairs）集合，键必须是互不相同的。一对大括号创建一个空的字典：{} 。

字典的主要操作是依据键来存储和析取值。也可以用 del 来删除“键值”对（key:value）。如果你用一个已经存在的关键字存储值，以前为该关键字分配的值就会被遗忘。试图从一个不存在的键中取值会导致错误，但可以为一个不存在的键赋值。

字典的 keys()方法返回由所有关键字组成的列表，该列表的顺序不定（如果你需要它有序，只能调用关键字链表的 sort()方法）。可以用 in 关键字检查字典中是否存在某一关键字。这里有个字典用法的小例子：

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
```

创建字典：

如果可以事先拼出整个字典，这种方法就很简单：

```
>>> d = {}
```

如果需要第一次动态建立字典的一个字段，第二种比较合适：

```
>>> d['name'] = 'wisdom'
>>> d['age'] = 45
```

第三种关键字形式所需的代码是比较少，但必须都是字符节才行：

```
>>> dict(name='wisdom',age=45)
```

{'age': 45, 'name': 'wisdom'}

如果需要在程序运行时把键和值逐步建成序列：

>>> dict([('name','wisdom'),('age',45)])

{'age': 45, 'name': 'wisdom'}

字典方法举例：

clear 方法，清除字典中所有的项，这是个原地操作所以无返回值（或者说返回 None），例一：

```
>>> x = {}
>>> y = x
>>> x['key'] = 'value'
>>> y
{'key': 'value'}
>>> x = {}
>>> y
{'key': 'value'}
```

例二：

```
>>> x = {}
>>> y = x
>>> x['key'] = 'value'
>>> y
{'key': 'value'}
>>> x.clear()
```

```
>>> y
{}
```

在两个例子中，x 和 y 最初对应同一个字典，在一中，通过将 x 关联到一个新的空字典来“清除”它，这对于 y 没有任何影响，y 仍然关联到原先的字典。这可能是所需要的行为，但是如果真的想要清空原始字典中所有的元素，必须使用 clear 方法。正如例二中看到的，y 随后也被清空了。

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(shallow copy)，因为值本身就是相同的，而不是副本)：

```
>>> x = {'username':'admin','machines':['foo','bar','baz']}
>>> y = x.copy()
>>> y['username'] = 'mln'
>>> y['machines'].remove('bar')
>>> y
{'username': 'mln', 'machines': ['foo', 'baz']}
>>> x
{'username': 'admin', 'machines': ['foo', 'baz']}
```

可以看到，当在副本中替换值得时候，原始字典不受影响，但是，如果修改了某个值 (原地修改，而不是替换)，原始的字典也会改变，因为同样的值也存储在原字典中。避免这个问题的一种方法就是使用深复制 (deep copy)，复制其包含的所有的值。可以使用 copy 模块的 deepcopy 函数来完成操作。

```
>>> from copy import deepcopy
>>> d = {}
```

```
>>> d['names'] = ['Alfred','Bertrand']
>>> c = d.copy()
>>> dc = deepcopy(d)
>>> d['names'].append('Clirc')
>>> c
{'names': ['Alfred', 'Bertrand', 'Clirc']}
>>> d
{'names': ['Alfred', 'Bertrand', 'Clirc']}
>>> dc
{'names': ['Alfred', 'Bertrand']}
```

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：

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

也可以直接在所有字典的类型 dict 上面调用方法：

```
>>> dict.fromkeys(['name','age'])
{'age': None, 'name': None}
```

如果不想使用 None 作为默认参数，也可以自己提供默认值：

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

get 方法，使用 get 访问一个不存在的键时，没有任何异常，而得到了 None 值。

还可以自定义默认值，替换 None：

```
>>> d = {}
```

```
>>> print d.get('name')
None
>>> print d.get('name','a')
a
>>> print d.get('name','ann')
ann
>>> d   # get 方法并没有创建键-值
{}
```

如果键存在，get 用起来就像普通的字典查询一样：

```
>>> d['name'] = 'Dick'
>>> d.get('name')
'Dick'
```

setdefault 方法，类似 get，能够获得与给定键相关联的值，除此之外，setdefault 还能在字典中不含有给定键的情况下设定相应的键-值，并相应的更新字典，默认值为 None：

```
>>> d = {}
>>> d.setdefault('name')
>>> d
{'name': None}   #默认值为 None
>>> d.setdefault('name','N')
>>> d
{'name': None}   #说明通过 setdefault 不能为已存在的键改变值
```

```
>>> d.setdefault('tt','N')
'N'    #为不存在的键设置值
>>> d
{'tt': 'N', 'name': None}
>>> d['name'] = 'Gill'   #为已存在的键修改值
>>> d
{'tt': 'N', 'name': 'Gill'}
```

has_key 方法，可以检查字典中是否含有给出的键，d.has_key(k)相当于 k in d：

```
>>> d = {}
>>> d.has_key('name')
False
>>> d['name'] = 'Eric'
>>> d.has_key('name')
True
```

keys 和 iterkeys 方法，keys 方法将字典中的键以列表形式返回，而 iterkeys 则返回针对键的迭代器。

```
>>> d.keys()
['url', 'spam', 'title']
>>> k = d.iterkeys()
>>> k
<dictionary-keyiterator object at 0x0222CE70>
```

items 和 iteritems 方法，items 方法将所有的字典项以列表方式返回，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：

```
>>> d = {'title':'Python','url':'http','spam':0}
>>> d.items()
[('url', 'http'), ('spam', 0), ('title', 'Python')]
```

iteritems 方法的作用大致相同，但是会返回一个迭代器对象而不是列表：

```
>>> it = d.iteritems()
>>> it
<dictionary-itemiterator object at 0x0222CDE0>
```

使用 for 循环，关键字和对应的值可以使用 iteritems() 方法同时解读出来：

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
        print k, v
gallahad the pure
robin the brave
```

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：

```
>>> d = {'title':'Python','url':'http','spam':0}
>>> d.pop('spam')
0
>>> d
{'url': 'http', 'title': 'Python'}
```

popitem 方法，类似于 list.pop，后者会弹出列表的最后一个元素。但不同的是，popitem 弹出随机的项，因为字典并没有“最后的元素”或者其他有关顺序的概念。

若想一个接一个的移除并处理项，这个方法就非常有效了（因为不用首先获取键的列表）：

```
>>> d = {'a':1,'b':2,'c':3}
>>> d.popitem()
('a', 1)
>>> d
{'c': 3, 'b': 2}
```

尽管 popitem 和列表的 pop 方法很类似，但字典中没有与 append 等价的方法。因为字典是无序的，类似于 append 的方法是没有任何意义的。

update 方法，利用一个字典项更新另外一个字典：提供的字典中的项会被添加到旧字典中，若有相同的键则会进行覆盖：

```
>>> d = {'a':1,'b':2,'c':3}
>>> x = {'d':4}
>>> d.update(x)
>>> d
{'a': 1, 'c': 3, 'b': 2, 'd': 4}
```

values 和 itervalues 方法，values 方法以列表的形式返回字典中的值（itervalues 返回值的迭代器），与返回键的列表不同的是，返回值的列表中可以包含重复的元素：

```
>>> d = {'a':1,'b':2,'c':3,'d':2}
>>> d.values()
[1, 3, 2, 2]
```

列表中存储关键字-值对元组的话，可以从中直接构造字典。键-值对来自某个特定模式时，可以用列表推导式简单的生成关键字-值列表：

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> dict([(x, x**2) for x in (2, 4, 6)])     # use a list comprehension
{2: 4, 4: 16, 6: 36}
```

使用简单字符串作为关键字的话，通常用关键字参数更简单：

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

字典的格式化字符串：

当以这种方式使用字典的时候，只要所有给出的键都能在字典中找到，就可以获得任意数量的转换说明符。这类字符串格式化在模板中非常有用（例如在 HTML）：

```
>>> template = '''<heml>
<head><title>%(title)s</title></head>
<body>
<h1>%(title)s</h1>
<p>%(text)s</p>
</body>'''
>>> data = {'title':'My Home Page','text':'Welcome!'}
>>> print template % data
<heml>
<head><title>My Home Page</title></head>
<body>
<h1>My Home Page</h1>
```

```
<p>Welcome!</p>

</body>
```

zip 与 map：

zip 将一个或者多个序列作为参数，然后分解组合返回列表：

```
>>> l1 = ['name','age','addres']

>>> l2 = [1,2,3,4]

>>> zip(l1,l2)

[('name', 1), ('age', 2), ('addres', 3)]

>>> dict(zip(l1,l2))

{'age': 2, 'name': 1, 'addres': 3}

>>> l3 = [10,20,30]

>>> zip(l1,l2,l3)

[('name', 1, 10), ('age', 2, 20), ('addres', 3, 30)]
```

map 类似于 zip，但是因为程度不够的情况下，较短的序列使用 None 来补全：

```
>>> s1 = 'strings'

>>> s2 = '1234567890'

>>> map(s1,s2)


Traceback (most recent call last):

  File "<pyshell#26>", line 1, in <module>

    map(s1,s2)

TypeError: 'str' object is not callable
```

```
>>> map(None,s1,s2)
[('s', '1'), ('t', '2'), ('r', '3'), ('i', '4'), ('n', '5'), ('g', '6'), ('s', '7'), (None, '8'), (None, '9'), (None, '0')]
>>> zip(s1,s2)
[('s', '1'), ('t', '2'), ('r', '3'), ('i', '4'), ('n', '5'), ('g', '6'), ('s', '7')]
```

集合

集合（set）是一个无序不重复元素的集。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。

可直接创建集合，（ ）里必须是列表：

```
>>> set([1,1,2,3,3])
set([1, 2, 3])
```

集合是由序列构建的。它们主要用于检查成员资格，因此副本是被忽略的：

```
>>> set([0,1,3,0,1,2,3])
set([0, 1, 2, 3])
```

和字典一样，几何元素的顺序是随意的，因此我们不应该以元素的顺序作为依据进行编程：

```
>>> set(['fee','fea','jj'])
set(['fee', 'jj', 'fea'])
```

集合是可变的，所以不能用作字典中的键。另外一个问题就是集合本身只能包含不可变（可散列）值，所以也就不能包含其他集合。

运算：

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
# unique letters in a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b
# letters in a but not in b
set(['r', 'd', 'b'])
>>> a | b
# letters in either a or b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b
# letters in both a and b
set(['a', 'c'])
>>> a ^ b
# letters in a or b but not both
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

判断子集：a 包含 b

```
>>> b.issubset(a)
True
>>> a.issuperset(b)
True
```

堆（heap）

它是优先队列的一种。使用要优先队列能够以任意顺序增加对象，并且能在任意时间（可能在增加对象的同时）找到（也可能是移除）最小的元素，也就是说它用于列表的 min 方法有效率的多。

Python 中没有独立的堆类型，只有一个包含一些堆操作函数的模块，这个模块叫做 heapq（q 是 queue 的缩写，即队列），包括 6 个函数：

heappush（heap，x） 将 x 入堆

heappop（heap） 将堆中最小元素弹出

heapify（heap） 将 heap 属性强制应用到任意一个列表

heapreplace（heap，x） 将堆中最小元素弹出，同时将 x 入堆

nlargest（n，iter） 返回 iter 中第 n 大的元素

nsmallest（n，iter） 返回 iter 中第 n 小的元素

heappush 函数用于堆加对的项。它只能用于通过各种堆函数建立的列表中。原因是元素的顺序很重要。

```
>>> from heapq import *
>>> from random import shuffle
>>> data = range(10)
>>> shuffle(data)    #将列表元素顺序打乱
>>> heap = []
>>> for n in data:
        heappush(heap,n)
>>> heap
```

```
[0, 1, 2, 4, 3, 6, 8, 7, 9, 5]
>>> heappush(heap,0.5)
>>> heap
[0, 0.5, 2, 4, 1, 6, 8, 7, 9, 5, 3]
```

元素的顺序并不像看起来那么随意。它们虽然不是严格排序的，但是也有规则的：位于 i 位置上的元素总比 2*i 以及 2*i+1 位置处的元素小。这是底层堆算法的基础，而这个特性称为堆属性（heap property）。

heappop 函数弹出最小的元素——一般来说都是在索引 0 处的元素。并且会确保剩余元素中最小的那个占据这个位置（保持堆属性）。一般来说，尽管弹出列表的第一个元素并不是很有效率，但是在这里不是问题，因为 heappop 在“幕后”会做一些精巧的移位操作：

```
>>> heappop(heap)
0
>>> heap
[0.5, 1, 2, 4, 3, 6, 8, 7, 9, 5]
>>> heappop(heap)
0.5
>>> heap
[1, 3, 2, 4, 5, 6, 8, 7, 9]
```

heapify 函数使用任意列表作为参数，并且通过尽可能少的移位操作，将其转换合法的堆（具有堆属性）。如果没有用 heappush 建立堆，那么在使用 heappush 和 heappop 前应该使用这个函数：

```
>>> data
[7, 6, 3, 5, 2, 0, 8, 4, 9, 1]
>>> heapify(data)
>>> data
[0, 1, 3, 4, 2, 7, 8, 5, 9, 6]
```

heapreplace 函数不像其它函数那么常用，它弹出堆的最小元素，并且将新元素推入。这样做比调用 heappop 之后再调用 heappush 更高效。

```
>>> heapreplace(data,0.8)
0
>>> data
[0.8, 1, 3, 4, 2, 7, 8, 5, 9, 6]
>>> heapreplace(data,10)
0.8
>>> data
[1, 2, 3, 4, 6, 7, 8, 5, 9, 10]
```

引用

当你创建一个对象并给它赋一个变量的时候，这个变量仅仅引用那个对象。而不是表示这个对象本身！也就是说，变量名指向你计算机中存储那个对象的内存，这被称作名称到对象的绑定。

条件、循环和其他语句

为模块和函数提供别名

```
>>> import math as foobar
>>> foobar.sqrt(4)
2.0
>>> from math import sqrt as ff
>>> ff(4)
2.0
```

赋值魔法

序列解包：

多个赋值操作可同时进行：

```
>>> x,y,z = 1,2,3
>>> print x,y,z
1 2 3
```

也可以交换多个变量：

```
>>> x,y = y,x
>>> print x,y,z
2 1 3
```

事实上，这里所做的事情叫做序列解包（sequence unpacking）或可迭代解包——将多个值的序列解开，然后放到变量的序列中。更形象一点的表示出来就是：

```
>>> values = 1,2,3
>>> values
(1, 2, 3)
>>> x,y,z = values
```

```
>>> x
1
```

当函数或者方法返回元组时，这个特性尤其有用。假设需要获取（和删除）字典中任意的键-值对，可以使用 popitem 方法，这个方法将键-值作为元组返回，那么这个元组就可以直接复制到两个变量中：

```
>>> d = {'name':'Robin','girlfriend':'Mraion'}
>>> key,value = d.popitem()
>>> key
'girlfriend'
>>> value
'Mraion'
```

链式赋值和增量赋值：

将同一个值赋给多个变量，不过这里只处理一个值：

```
x = y = somefunction()
```

等价于：

```
>>> y = somefunction()
>>> x = y
```

但不等价于：

```
>>> y = somefunction()
>>> x = somefunction()
```

增量赋值：

```
x += 1      x *= 2
```

条件和条件语句

这就是布尔变量的作用：

这些值在作为布尔表达式的时候，会被解释器看做假（false）：Fales，none，0，""，()，[]，{}，除了它们，其他任何值都被解释为真：

```
>>> True
True
>>> False
False
>>> True == 1
True
>>> False == 0
True
>>> True + False + 43
44
```

if,elif,else：

is：同一性运算符：

x is y 表示 x 和 y 是同一个对象

```
>>> x = y = [1,2,3]
>>> z = [1,2,3]
>>> x == y
True
>>> x is y
```

```
True
>>> x is z
False
>>> y == z
True
```

x 和 z 相等却不等同，因为 is 运算是判定同一性而不是相等性的。变量 x 和 y 都被绑定到同一个列表上，而变量 z 被绑定在另一个具有相同数值和顺序的列表上。它们的值可能相等，但是却不是同一个对象。

比较操作符 in 和 not in 审核值是否在一个区间之内。操作符 is 和 is not 比较两个对象是否相同；这只和诸如列表这样的可变对象有关。所有的比较操作符具有相同的优先级，低于所有的数值操作。

比较操作可以传递。例如 a < b == c 审核是否 a 小于 b 并且 b 等于 c。比较操作可以通过逻辑操作符 and 和 or 组合，比较的结果可以用 not 来取反义。这些操作符的优先级又低于比较操作符，在它们之中，not 具有最高的优先级，or 优先级最低，所以 A and not B or C 等于(A and (notB)) or C。当然，括号也可以用于比较表达式。

布尔运算：and，or：

```
if number <=10 and number >= 1
if ((cash > price) or castomer_has_good_credit) and not_out_of_s:
```

断言：

如果需要确保程序中某个条件一定为真才能让程序正常工作的话，assert 语句就有用了，它可以在程序中置入检查点。条件后可以添加字符串，用来解释断言：

```
>>> a = -1
```

```
>>> assert a >=0,'wrong'
```

循环

while 循环：

用来在任何条件为真的情况下重复执行一个代码块：

```
>>> name = ''
>>> while not name:
        name = raw_input('enter:')
enter:dd
>>> print 'Hello,%s' % name
Hello,dd
```

for 循环：

一般用于为一个集合（序列和其他可选迭代对象）的每个元素都执行一个代码块。

可以在循环中使用序列解包：

```
>>> d = {'x':1,'y':2,'z':3}
>>> for key,value in d.items():
        print key,'corresponds to ',value
y corresponds to  2
x corresponds to  1
z corresponds to  3
```

等价于：

```
>>> for key in d:
        print key,'corresponds to ',d[key]
```

y corresponds to 2

x corresponds to 1

z corresponds to 3

一些迭代工具：

并行迭代：

```
>>> names = ['anne','berh','gero']
>>> ages = [12,33,54]
>>> for i in range(len(names)):
        print names[i],'is',ages[i],'years old'


anne is 12 years old
berh is 33 years old
gero is 54 years old
```

内建的 zip 函数可以用来并行迭代，可以把两个序列“压缩”在一起，然后返回一个元组的列表：

```
>>> zip(names,ages)
[('anne', 12), ('berh', 33), ('gero', 54)]
```

现在我们可以在循环中解包元组：

```
>>> for name,age in zip(names,ages):
        print name,'is',age,'years old'


anne is 12 years old
```

```
berh is 33 years old
gero is 54 years old
```

zip 函数也可以作用于任意多的序列。关于它很重要的一点是 zip 可以应付不等长的序列：当最短的序列“用完”的时候就会停止：

```
>>> zip(range(5),xrange(0,100,2))
[(0, 0), (1, 2), (2, 4), (3, 6), (4, 8)]
```

在上面的代码中，不推荐用 range 替换 xrange——尽管只需要前 5 个数字，但 range 会计算多有的数字，这要花费很长的时间。而是用 xrange 就没这个问题里了，它只计算前五个数字。

编号迭代

有些时候想要迭代序列中的对象，同时还要获取当前的对象的索引。例如，在一个字符串列表中替换所有包含“xxx”的子字符串：

```
for string in strings:
    if 'xxx' in string:
        index = strings.index(string)
        strings[index] = '[consored]'
```

没问题，但是在替换前要搜索给定的字符串似乎没必要。如果不替换的话，搜索还会返回错误的索引（前面出现的同一个词的索引）。一个比较好的版本如下：

```
>>> index = 0
>>> for string in strings:
        if 'xxx' in string:
            strings[index] = '[consored]'
```

```
            index += 1
```

方法有些奔，不过可以接受，另一种方法是使用内建的 enumerate 函数：

```
for index,string in enumerate(strings):
    if 'xxx' in string:
        strings[index] = '[consored]'
```

翻转和排序迭代

reversed 和 sorted，不是原地修改对象，而是返回翻转或排序后的版本：

```
>>> sorted('Hello,world!')
['!', ',', 'H', 'd', 'e', 'l', 'l', 'l', 'o', 'o', 'r', 'w']
>>> '+'.join(sorted('Hello,world!'))
'!+,+H+d+e+l+l+l+o+o+r+w'
```

跳出循环

break

break 语句用来终止循环语句的 ,哪怕循环条件没有成为 Fales 或序列没有被完全递归，也停止执行循环语句。

```
>>> for n in range(2,10):
        for i in range(2,n):
            if n % i == 0:
                print n,'equals',i,'*',n/i
                break
            else:
                print n,'is a prime number'
```

```
3 is a prime number
4 equals 2 * 2
5 is a prime number
5 is a prime number
5 is a prime number
6 equals 2 * 3
7 is a prime number
7 is a prime number
7 is a prime number
7 is a prime number
7 is a prime number
8 equals 2 * 4
9 is a prime number
9 equals 3 * 3
```

continue

continue 语句用来告诉 Python 跳出当前循环块中的剩余语句，然后继续进行下一轮循环：

```
while True:
    s = raw_input('Enter something:')
    if s == 'quit':
        break
```

```
        elif len(s) < 3:
            continue
        else:
            print 'www'
```

返回：

```
Enter something:3
Enter something:33
Enter something:ad
Enter something:adfg
www
Enter something:quit
```

raw_input 函数提供一个字符串。

while True/break 习语

如果需要当用户在提示如下输入单词时做一些事情，并且在用户不输入单词后结束循环：

```
>>> while True:
        w = raw_input('enter:')   #第一部分
        if not w:
            break
        print 'the word is ' + w     #第二部分


enter:e
```

```
the word is e
enter:w
the word is w
enter:
```

while True 的部分实现了一个永远不会自己停止的循环。但是在循环内部的 if 语句中加入条件是可以的，在条件满足时调用 break 语句，这样一来就可以在循环内部任何地方而不是只在开头（像普通的 while 循环一样）终止循环。if/break 语句自然地将循环分为两部分：第一部分负责初始化（在普通的 while 循环中，这部分需要重复）,第二部分则在循环条件为真的情况下使用第一部分内初始化好的数据。

循环中的 else 子句

当在循环内使用 break 语句时，通常是因为“找到”了某物或者因为某事“发生”了。在跳出时做了一些事情是很简单的（比如 print n），但是有些时候想要在没有跳出之前做些事情。那么怎么判断呢？可以使用布尔变量，在循环前将其设定为 False，跳出后设定 True，然后再使用 if 语句查看循环是否跳出了。

```
broke_out = False
for x in seq:
    do_something(x)
    if condition(x):
        broke_out = True
        break
    do_something_else(x)
if not broke_out:
```

```
        print 'I didn\'t break out'
```

更简单的方式是在循环中增加一个 else 子句——它仅在没有调用 break 时执行。让我们用这个方法重写刚才的例子：

```
>>> from math import sqrt
>>> for n in range(99,81,-5):
        root = sqrt(n)
        if root == int(root):
            print n
            break
        else:
            print 'Didn\'t find it'


Didn't find it
Didn't find it
Didn't find it
Didn't find it
```

列表推导式——轻量级循环

列表推导式（list comprehensive）是利用其它列表创建新列表，工作方式类似于 for 循环：

```
>>> [x*x for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

如果只想打印那些能被 3 整除的平方数：

```
>>> [x*x for x in range(10) if x%3==0]
[0, 9, 36, 81]
```

也可以增加更多 for 语句的部分：

```
>>> [(x,y) for x in range(3) for y in range(3)]
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

作为对比，下面的代码使用两个 for 语句创建了相同的列表：

```
>>> result = []
>>> for x in range(3):
        for y in range(3):
            result.append((x,y))
>>> result
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

也可以和 if 子句联合使用，像以前一样：

```
>>> girls = ['alice','bernice','clarice']
>>> boys = ['chris','arnold','bob']
>>> [b+'+'+g for b in boys for g in girls if b[0]==g[0]]
['chris+clarice', 'arnold+alice', 'bob+bernice']
```

这样就得到了那些名字首字母相同的男孩和女孩。男孩与女孩名字对的例子其实效率不高，因为它会检查每个可能的配对。

更优方案：

```
>>> girls = ['alice','bernice','clarice']
>>> boys = ['chris','arnold','bob']
```

```
>>> letterGirls = {}
>>> for girl in girls:
        letterGirls.setdefault(girl[0],[]).append(girl)
>>> print [b+'+'+g for b in boys for g in letterGirls[b[0]]]
['chris+clarice', 'arnold+alice', 'bob+bernice']
>>> letterGirls
{'a': ['alice'], 'c': ['clarice'], 'b': ['bernice']}
```

这个程序建造了一个叫做 letterGirls 的字典，其中每一项都把单字母作为键，以女孩名字组成的列表作为值。在字典建立后，列表推导式循环整个男孩集合，并且查找那些和当前男孩名字首字母相同的女孩集合。这样列表推导式就不用尝试所有的男孩女孩的组合，检查首字母是否匹配。

三人行

pass

pass 语句什么也不做。它用于那些语法上必须要有什么语句，但程序什么也不做的场合。它可以在代码中做占位符使用：

```
if name == 'a':
    print 'a'
elif name == 'b':
    pass
elif name == 'c':
    print 'c'
```

也通常用于创建最小结构的类：

```
>>> class MyEmptyClass:
        pass
```

del

一般来说，Python 会删除那些不再使用的对象（因为使用者不会再通过任何变量或者数据结构引用他们）：

```
>>> s = {'a':1,'b':3}
>>> r = s
>>> s
{'a': 1, 'b': 3}
>>> r
{'a': 1, 'b': 3}
>>> s = None
>>> r
{'a': 1, 'b': 3}
>>> r = None
```

首先，r 和 s 都被绑定到同一个字典上。所以当设置 s 为 None 的时候，字典通过 r 还是可用的。但是当我把 r 也设置为 None 的时候，字典就是“浮”在内存里面了，没有任何名字绑定到它上面。没有办法获取和使用它，所以 Python 解释器直接删除了那个字典（这种行为被称为垃圾收集）。注意，也可以使用 None 之外的其他值，字典同样会“消失不见”。另外一个方法就是使用 del 语句，它不仅会移除一个对象的引用，也会移除那个名字本身。

```
>>> x = 1
```

```
>>> del x
>>> x

Traceback (most recent call last):
  File "<pyshell#125>", line 1, in <module>
    x
NameError: name 'x' is not defined
```

函数

函数

函数允许你给一个语句块一个名称，然后你可以在你程序中的任何地方使用这个名称任意多次的运行这个语句块，这被称为调用函数。

```
def 函数名（变量名）:
    语句块
```

参数对于函数而言，只是给函数的输入，以便于我们可以传递不同的值给函数，然后得到相应的结果。

函数中的参数叫形参，而你提供给函数调用的值叫实参。

我们可以定义一个函数以生成任意上界的斐波那契数列：

```
>>> def fib(n):
        a,b = 0,1
        while b < n:
            print b,
```

```
        a,b = b,a + b

>>> fib(2000)

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

记录函数

如果在函数的开头写下“字符串”用以注释，它就会作为函数的一部分进行存储，这样为文档字符串：

```
def sqrare(x):
    'Calculates the square of the number x.'
    return x * x
```

文档字符串可以按如下方式访问：

```
>>> sqrare.__doc__
'Calculates the square of the number x.'
```

使用内建的 help 函数，就可以得到关于函数的信息，包括它的文档字符串的信息。

作为用户定义函数

函数名有一个为解释器认可的类型值。这个值可以赋给其它命名，使其能够作为一个函数来使用。这就像一个重命名机制：

```
>>> fib
<function fib at 0x01B17F70>
>>> f = fib
>>> f(400)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

以下示例演示了如何从函数中返回一个包含斐波那契数列的数值链表，而不是打印它：

```
>>> def fib2(n):
        result = []
        a,b = 0,1
    while b < n:
        result.append(b) #等同于 result += [b]
        a,b = b,a + b
    return result
>>> f1 = fib2(200)
>>> f1
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

return 语句从函数中返回一个值，不带表达式的 return 返回 None。过程结束后也会返回 None。没有 return 语句，或者虽有 return 语句但 return 后边没有跟任何值得函数不返回值：

```
>>> def test():
        print 'This is printed'
        return
        print 'This is not'

>>> x = test()
This is printed
```

这里的 return 语句只起到结束函数的作用。可以看到，第二个 print 语句被跳过了。print x 返回 None。所以，所有的函数的确返回了东西：当不需要它们返回值的时候，它们就会返回 None。

局部变量：

当你在函数定义内声明变量的时候，它们与函数外相同名称的其他变量没有任何关系，即变量名称对于函数来说是局部的，这称为变量的作用域，所有变量的作用域是它们被定义的块，从它们的名称被定义的那一点开始。

```
>>> def func(x):
        print 'x is',x
        x = 2
        print 'changed local x to',x
>>> x = 50
>>> func(x)
x is 50
changed local x to 2
>>> print 'x is still',x
x is still 50
```

第一次使用 x 的值得时候，Python 使用函数声明的形参的值（50）；接着，把 2 赋值给 x，但 x 是函数的局部变量，所以，改变 x 的值对函数外的 x 没有影响，而且最后一个语句也证明了外面的 x 确实没有被影响。

global 语句：

如果你想为定义在函数外的一个变量赋值，那么你就得告诉 Python 这个变量

不是局部的而是全局的。

使用 global 语句可以清楚地表明变量是在函数外定义的。

```
def func():
    global x
    print 'x is',x
    x = 2
    print 'x changed to',x
>>> x = 40
>>> func()
x is 40
x changed to 2
>>> 'x is still',x
('x is still', 2)
```

函数外定义的变量：全局变量；函数内部定义的变量：局部变量。可以同时制定多个全局变量：global x,y,z。

默认参数值：

对于一些函数，你可能希望它的参数是可选的，如果用户不想为这些参数提供值得话，这些参数就是用默认值。

```
>>> def say(message,time=1):
        print message*time
>>> say('Hi')
Hi
```

```
>>> say('Hi',5)
HiHiHiHiHi
```

只有在形参末尾的那些参数可以有默认参数值，def func (a=4 , b) 是无效的。

使用默认参数值创建的函数可以用较少的参数来调用。例如：

```
>>> def ask_ok(prompt,retries=4,complaint='Yes or no,please!'):
        while True:
            ok = raw_input(prompt)
            if ok in ('y','yes'):
                return True
            if ok in ('n','no'):
                return False
            retries = retries - 1
            if retries < 0:
                raise IOError('refusenik user')
            print complaint
```

这个函数可以通过几种不同的方式调用：

1. 只给出必要的参数：ask_ok('want to quit?')
2. 给出一个可选的参数：ask_ok('overwrite the file?',2)
3. 给出所有参数：ask_ok('sure?',1,'only y or n!')

```
>>> ask_ok('want to quit?')
want to quit?e
Yes or no,please!
```

```
want to quit?e

Yes or no,please!

want to quit?e

Yes or no,please!

want to quit?e

Yes or no,please!

want to quit?e


Traceback (most recent call last):

  File "<pyshell#41>", line 1, in <module>

    ask_ok('want to quit?')

  File "<pyshell#40>", line 10, in ask_ok

    raise IOError('refusenik user')

IOError: refusenik user

>>> ask_ok('overwrite the file?',2)

overwrite the file?e

Yes or no,please!

overwrite the file?e

Yes or no,please!

overwrite the file?n

False

>>> ask_ok('sure?',1,'only y or n!')
```

```
sure?e
only y or n!
sure?e

Traceback (most recent call last):
  File "<pyshell#43>", line 1, in <module>
    ask_ok('sure?',1,'only y or n!')
  File "<pyshell#40>", line 10, in ask_ok
    raise IOError('refusenik user')
IOError: refusenik user
```

关键字参数

使用参数名提供的参数叫做关键字参数。主要的作用在于可以明确每个参数的作用，就算弄乱了参数的顺序，对于程序功能也没有影响。关键字参数最厉害的地方在于可以在函数中给参数提供默认值。

函数可以通过关键字参数的形式来调用，形如 keyword=value。例如，以下的函数：

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

可以用以下任意方法调用：

parrot(1000)

parrot(action = 'VOOOOOM', voltage = 1000000)

parrot('a thousand', state = 'pushing up the daisies')

parrot('a million', 'bereft of life', 'jump')

不过以下几种调用是无效的：

parrot()

parrot(voltage=5.0, 'dead')

parrot(110, voltage=220) #形参不能在同一次调用中同时使用位置和关键字绑定值

parrot(actor='John Cleese')

通常，参数列表中的每一个关键字都必须来自于形式参数，每个参数都有对应的关键字。形式参数有没有默认值并不重要。实际参数不能一次赋多个值——形式参数不能在同一次调用中同时使用位置和关键字绑定值。

引入一个形如**name 的参数时，它接收一个字典，该字典包含了所有未出现在形式参数中的关键字参数。这里可能还会组合使用一个形如*name 的形式参数，它接收一个元组，包含了所有没有出现在形式参数列表中的参数值（*name 必须在**name 之前出现）。“*”用来收集其余的位置参数，参数前的星号将所有值放置在同一个元组中，可以说是将这些值收集起来，然后使用。如果不提供任何供收集的元素，*name 就是个空元组。使用“**”才能进行处理关键字参数的“收集”操作，只不过**name 代表的是字典。

例如内建的函数 range()需要独立的 start，stop 参数，你可以在调用函数时加

一个*操作符来自动把参数列表拆开（可变参数列表）：

```
>>> range(3, 6)
 [3, 4, 5]
>>> args = [3, 6]
>>> range(*args)
[3, 4, 5]
```

简单的小例子：

```
>>> def pp(x,y,z=3,*pospar,**keypar):
        print x,y,z
        print pospar
        print keypar
>>> pp(1,2,3,4,5,6,foo=1,bar=2)
1 2 3
(4, 5, 6)
{'foo': 1, 'bar': 2}
>>> pp(1,2)
1 2 3
()
{}
```

练习使用参数：

```
def story(**kwds):
    return 'Once upon a time ,there was a %(job)s
```

```
called %(name)s' % kwds


    def power(x,y,*others):
        if others:
            print 'Received redundant parameters:',others
        return pow(x,y)


    def interial(start,stop=None,step=1):
        'Imitates range() for step>0'
        if stop is None:
            start,stop = 0,start
        result = []
        i = start
        while i < stop:
            result.append(i)
            i += step
        return result
```

测试：

```
    >>> story(job='king',name='Ganby')
    'Once upon a time ,there was a king called Ganby'
    >>> story(name='Robin',job='brave knight')
    'Once upon a time ,there was a brave knight called Robin'
```

```
>>> params = {'job':'language','name':'Python'}
>>> story(**params)
'Once upon a time ,there was a language called Python'
>>> del params['job']
>>> story(job='stroke of genius',**params)
'Once upon a time ,there was a stroke of genius called Python'
>>> power(2,3)
8
>>> power(y=3,x=2)
8
>>> params = (5,) * 2
>>> params
(5, 5)
>>> power(*params)
3125
>>> power(3,3,'Hello,world!')
Received redundant parameters: ('Hello,world!',)
27
>>> power(*interial(3,7))
Received redundant parameters: (5, 6)
81
```

嵌套作用域：

一般用在一个函数中定义另一个函数：

```
>>> def func_external(any):
        def func_internal(number):
            return any * number
        return func_internal

>>> x = func_external(9)
>>> y = x(2)
>>> y
18
>>> z = func_external(2)(9)
>>> z
18
```

类似于 func_external 函数有存储子封闭作用域的行为叫做闭包（closure）。

关于函数的一些应用（递归方面）：

递归，简单来说就是引用（或者调用）自身的意思。有用的递归函数包含以下两个部分：

1.当函数直接返回值时有基本实例（最小可能性问题）；

2.递归实例，包括一个或者多个问题最小部分的递归调用。

阶乘：n*(n-1)*(n-2)*...*1

递归版本：

```
>>> def fact_re(n):
```

```
        if n == 1:
            return 1
        return n * fact_re(n-1)
>>> print fact_re(5)
120
```

原版本：

```
>>> def factorial(n):
        result = n
        for i in range(1,n):
            result *= i
        return result
>>> factorial(4)
24
```

求幂：

power(x,n)是 x 自乘 n-1 次的结果，所以 power(2,3)是 2 乘自身两次：2X2X2=8

递归版本：

```
>>> def power(x,n):
        if n == 0:
            return 1
        else:
            return x * power(x,n-1)
>>> power(2,5)
```

32

原版本：

```
>>> def power(x,n):
        result = 1
        for i in range(n):
            result *= x
        return result
>>> power(2,3)
8
```

二元查找：

```
>>> def search(l,number,lower=0,upper=None):
        if upper == None:
            upper = len(l) - 1
        if lower == upper:
            assert number == l[upper]
            return upper
        else:
            middle = (lower + upper) / 2
        if number > l[middle]:
            return search(l,number,middle+1,upper)
        else:
            return search(l,number,lower,middle)
```

```
>>> index = search([1,3,5,7,9,11,13],9)
>>> print index
4
```

n*(n-1)*n(n-2)*...*n*1 :

```
>>> def fact(n):
        res = 1
        for i in range(1,n):
            res *= n * i
        return res
>>> print fact(5)
15000
```

类

Python 在尽可能不增加新的语法和语义的情况下加入了类机制。类机制最重要的功能：类继承机制允许多继承，派生类可以覆盖（override）基类中的任何方法，方法中可以调用基类中的同名方法。对象可以包含任意数量的数据。

关于命名和对象的内容

对象：基本上可以看做数据（特性）以及由一系列可以存取、操作这些数据的方法所组成的集合。

对象的优点：

多态（Polymorephism）：意味着可以对不同类的对象使用同样的操作。

封装（Encapsulation）；对外部世界隐藏对象的工作细节。

继承（Inheritance）：以普通的类为基础建立专门的类对象。

多态来自于希腊语，意思是“有多种形式”。绑定到对象特性上面的函数称为方法。多态，使用对象而不用知道其内部细节，封装亦这样。

多态可以让用户对于不知道是什么类（或对象类型）的对象进行方法调用，而封装是可以不用关心对象是如何构建的而直接进行使用。

封装是对外部隐藏不必要的细节的原则，它与多态都是抽象的原则。

多态的多种形式是指，任何不知道对象到底是什么类型，但是又要对对象“做点什么”的时候，都会用到多态。例如：

标准库 random 中包含 choice 函数，可以从序列中随机选出元素，给变量赋值：

```
>>> from random import choice
>>> x = choice(['Hello,world!',[1,2,'e','e',4]])
```

运行后，变量 x 会包含字符串 Hello,world!，也有可能包含列表[1,2,'e','e',4]，不用关心到底是哪个类型。要关心的就是在变量 x 中字符已出现多少次，而不管 x 是字符串还是列表，都可以使用 count 函数：

```
>>> x.count('e')  #返回 1 或 2
```

本例重点是不需要检测类型，只需知道 x 有个叫做 count 的方法，带有一个字符作为参数，并且返回整数值就够了。整数也被称作对象（属于 int 类）。所有的对象都属于某一个类，称为类的实例（instance）。

内建的对象是基于类型的，自定的对象则是基于类的。可以创建类但是不能创建类型。

属于一个对象或类的变量被称为域，对象也可以使用属于类的函数来具有功能，

这样的函数被称为类的方法。域和方法可以合称为类的属性。

类也有方法，即仅仅为类定义的函数。仅仅在你有一个该类的对象的时候，你才可以使用这些功能。列表可以算是一个类。一个类也有域，它是仅仅为类定义的变量。方法（更专业一点可以称为绑定方法）将它们的第一个参数绑定到所属的实例上。对象是被特化的，多个名字（在多个作用域中）可以绑定同一个对象。这相当于其它语言中的别名。

作用域和命名空间

命名空间是从命名到对象的映射。顺便提一句，Python 中任何一个“.”之后的命名为属性 - - 例如，表达式 z.real 中的 real 是对象 z 的一个属性。严格来讲，从模块中引用命名是引用属性：表达式 modname.funcname 中，modname 是一个模块对象，funcname 是它的一个属性。因此，模块的属性和模块中的全局命名有直接的映射关系：它们共享同一命名空间！

作用域是 Python 程序中一个命名空间可以直接访问的文法区域。“直接访问”在这里的意思是查找命名时无需引用命名前缀。尽管作用域是静态定义，在使用时他们都是动态的。每次执行时，至少有三个命名空间可以直接访问的作用域嵌套在一起：包含局部命名的使用域在最里面，首先被搜索；其次搜索的是中层的作用域，这里包含了同级的函数；最后搜索最外面的作用域，它包含内置命名：

1. 首先搜索最内层的作用域，它包含局部命名
2. 任意函数包含的作用域，是内层嵌套作用域搜索起点，包含非局部，但是也非全局的命名。
3. 接下来的作用域包含当前模块的全局命名。
4. 最外层的作用域（最后搜索）是包含内置命名的命名空间。

重要的是作用域决定于源程序的意义：一个定义于某模块中的函数的全局作用域是该模块的命名空间，而不是该函数的别名被定义或调用的位置，了解这一点非常重要。另一方面，命名的实际搜索过程是动态的，在运行时确定的。

Python 的一个特别之处在于——如果没有使用 global 语法——其赋值操作总是在最里层的作用域。赋值不会复制数据——只是将命名绑定到对象。删除也是如此：del x 只是从局部作用域的命名空间中删除命名 x 。事实上，所有引入新命名的操作都作用于局部作用域。特别是 import 语句和函数定义将模块名或函数绑定于局部作用域。（可以使用 global 语句将变量引入到全局作用域。）

类定义语法

最简单的类定义形式如下

```
class ClassName:
    <statement-1>
.       。。。
    <statement-N>
```

进入类定义部分后，会创建出一个新的命名空间，作为局部作用域——因此，所有的赋值称为这个新命名空间的局部变量。特别是函数定义在此绑定了新的命名。

类定义完成时（正常退出），就创建了一个类对象。基本上它是对类定义创建的命名空间进行了一个包装；原始的局部作用域（类定义引入之前生效的那个）得到恢复，类对象在这里绑定到类定义头部的类名。

类对象

类对象支持两种操作：属性引用和实例化。

属性引用使用和 Python 中所有的属性引用一样的标准语法：obj.name。类对

象创建后，类命名空间中所有的命名都是有效属性名。所以如果类定义是这样：

```
class MyClass:
"""A simple example class"""
    i = 12345
    def f(self):
        return 'hello world'
```

那么 MyClass.i 和 MyClass.f 是有效的属性引用，分别返回一个整数和一个方法对象。也可以对类属性赋值，你可以通过给 MyClass.i 赋值来修改它。__doc__ 也是一个有效的属性，返回类的文档字符串："A simple example class" 。

类的实例化使用函数符号。只要将类对象看作是一个返回新的类实例的无参数函数即可。例如（创建一个新的类实例并将该对象赋给局部变量 x）：

```
x = MyClass()
```

__init__方法（构造方法）

__init__方法在类的一个对象被建立时，马上运行。这个方法可以用来对你的对象做一些你希望的初始化。使用__init__方法：

```
>>> class Person:
        def __init__(self,name):
            self.name = name
        def sayHi(self):
            print 'Hi',self.name
>>> p = Person('Tom')
>>> p.sayHi()
```

Hi Tom

这里，我们把__init__方法定义为与一个参数 name（以及普通的参数 self）。在这个__init__里，我们只是创建一个新的域，也称为 name。注意它们是两个不同的变量，尽管它们有相同的名字。点号使我们能够区分它们。

最重要的是，我们没有专门调用__init__方法，只是在创建一个类的新实例的时候，把参数包括在圆括号内跟在类名后面，从而传递给__init__方法。这是这种方法的重要之处。

现在，我们能够在我们的方法中使用 self.name 域。这在 sayHi 方法中得到了验证。

其实，在 self.name = name 中我们可以把 self.name 中的 name 看作是该类所具有的一个属性，而等号右边的 name 就是一个形参而已，通过 self.name = name 定以后，该类的其它函数（方法）可以直接调用这个属性。

```
>>> class computer:
        def __init__(self,name):
            self.name = name
        def start(self):
            print '%s is start' % self.name
>>> c = computer('b')
>>> c.start()
b is start
>>> print c.name
b
```

当然，出于弹性的需要，__init__ () 方法可以有参数。事实上，参数通过__init__ () 传递到类的实例化操作上。例如：

```
>>> class Complex:
        def __init__(self,realpart,imagpart):
            self.r = realpart
            self.i = imagpart
>>> x = Complex(3.0,4.5)
>>> x.r,x.i
(3.0, 4.5)
```

实例对象

实例对象唯一可用的操作就是属性引用。有两种有效的属性名。

和局部变量一样，数据属性不需要声明，第一次使用时它们就会生成。例如，如果 x 是前面创建的 MyClass 实例，下面这段代码会打印出 16 而在堆栈中留下多余的东西：

```
>>> x.counter = 1
>>> while x.counter < 10:
        x.counter = x.counter * 2
>>> print x.counter
16
>>> del x.counter
>>> x.counter
```

```
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    x.counter
AttributeError: Complex instance has no attribute 'counter'
```

另一种为实例对象所接受的引用属性是方法。方法是“属于”一个对象的函数。

实例对象的有效名称依赖于它的类。按照定义，类中所有（用户定义）的函数对象对应它的实例中的方法。所以在我们的例子中，x.f 是一个有效的方法引用，因为 MyClass.f 是一个函数。但 x.i 不是，因为 MyClass.i 不是函数。不过 x.f 和 MyClass.f 不同 - - 它是一个方法对象 ，不是一个函数对象。

方法对象

通常，方法通过右绑定调用：

```
x.f()
```

在 MyClass 示例中，这会返回字符串'hello world'。然而，也不是一定要直接调用方法。x.f 是一个方法对象，它可以存储起来以后调用。例如：

```
xf = x.f
while True:
    print xf()     #会不断的打印 ``hello world''
```

调用方法时发生了什么？你可能注意到调用 x.f()时没有引用前面标出的变量，尽管在 f()的函数定义中指明了一个参数。这个参数怎么了？事实上如果函数调用中缺少参数，Python 会抛出异常 - - 甚至这个参数实际上没什么用。

实际上，方法的特别之处在于实例对象作为函数的第一个参数传给了函数。在我们的例子中，调用 x.f()相当于 MyClass.f(x)。通常，以 n 个参数的列表去调用一

个方法就相当于将方法的对象插入到参数列表的最前面后，以这个列表去调用相应的函数。

一些说明

同名的数据属性会覆盖方法属性，为了避免可能的命名冲突－－这在大型程序中可能会导致难以发现的 bug－－最好以某种命名约定来避免冲突。可选的约定包括方法的首字母大写，数据属性名前缀小写（可能只是一个下划线），或者方法使用动词而数据属性使用名词。

数据属性可以由方法引用，也可以由普通用户（客户）调用。换句话说，类不能实现纯抽象数据类型。事实上 Python 中没有什么办法可以强制隐藏数据－－一切都基本约定的惯例。

self

类的方法与普通的函数只有一个特别的区别——它们必须有一个额外的第一个参数名称，但是在调用这个方法的时候你不必为这个参数赋值，Python 会提供这个值。这个特别的变量指对象本身，按照惯例，它的名称是 self。

假如你有一个类称为 MyClass 和这个类的一个实例 MyObject。当你调用这个对象的方法 MyObject.method(arg1,arg2)的时候，这会由 Python 自动转为 MyClass.method(MyObject,arg1,arg2)——这就是 self 的原理了。这也意味着如果你有一个不需要参数的方法，你还是得给这个方法定义一个 self 参数。

通常方法的第一个参数命名为 self。这仅仅是一个约定：对 Python 而言，self 绝对没有任何特殊含义。self 参数是对对象自身的引用，没有它的话，成员方法就没法访问他们要对其特性进行操作的对象本身了。所以，不属于任何类的函数（不在类中定义的函数）都可不加 self 参数。

类属性中的任何函数对象在类实例中都定义为方法。不是必须要将函数定义代码写进类定义中，也可以将一个函数对象赋给类中的一个变量。例如：

```
def f1(self, x, y):
    return min(x, x+y)
class C:
    f = f1
    def g(self):
        return 'hello world'
    h = g
```

现在 f, g 和 h 都是类 C 的属性，引用的都是函数对象，因此它们都是 C 实例的方法 - - h 严格等 g。要注意的是这种习惯通常只会迷惑程序的读者。

通过 self 参数的方法属性，方法可以调用其它的方法：

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

方法可以像引用普通的函数那样引用全局命名。与方法关联的全局作用域是包含类定义的模块（类本身永远不会做为全局作用域使用）。尽管很少有好的理由在方

法中使用全局数据，全局作用域确有很多合法的用途：其一是方法可以调用导入全局作用域的函数和方法，也可以调用定义在其中的类和函数。通常，包含此方法的类也会定义在这个全局作用域。

每个值都是一个对象，因此每个值都有一个类（也称为它的类型），它存储为object. __class__。

继承

当然，如果一种语言不支持继承，“类”就没有什么意义。派生类的定义如下所示：

```
class DerivedClassName(BaseClassName):
    <statement-1>
.       。。。
    <statement-N>
```

命名 BaseClassName（示例中的基类名）必须与派生类定义在一个作用域内。除了类，还可以用表达式，基类定义在另一个模块中时这一点非常有用。

派生类定义的执行过程和基类是一样的。构造派生类对象时，就记住了基类。这在解析属性引用的时候尤其有用：如果在类中找不到请求调用的属性，就搜索基类。如果基类是由别的类派生而来，这个规则会递归的应用上去。

派生类可能会覆盖其基类的方法。派生类中的覆盖方法可能是想要扩充而不是简单的替代基类中的重名方法。有一个简单的方法可以直接调用基类方法，只要调用：BaseClassName.methodname(self, arguments) 。有时这对于客户也很有用（要注意只有 BaseClassName 在同一全局作用域定义或导入时才能这样用）。

一个子类型在任何需要父类型的场合可以被替换成父类型，即子类型的对象可

以被视作是父类的实例，这种现象被称为多态现象。父类被称为基本类或超类。

```
>>> class SchoolMember:
        def __init__(self,name,age):
            self.name = name
            self.age = age
            print '(Initialized SchoolMember:%s)' % self.name
        def tell(self):
            print 'Age:%s Name:%s' % (self.age,self.name)
>>> class Teacher(SchoolMember):
        def __init__(self,name,age,salary):
            SchoolMember.__init__(self,name,age)
            self.salary = salary
            print '(Initialized Teacher:%s)' % self.name
        def tell(self):
            SchoolMember.tell(self)
            print 'Salary:%s' % self.salary
>>> class Student(SchoolMember):
        def __init__(self,name,age,marks):
            SchoolMember.__init__(self,name,age)
            self.marks = marks
            print '(Initialized Student:%s)' % self.name
        def tell(self):
```

```
                SchoolMember.tell(self)
                print 'Marks:%s' % self.marks
>>> t = Teacher('T',32,5000)
(Initialized SchoolMember:T)
(Initialized Teacher:T)
>>> s = Student('S',22,98)
(Initialized SchoolMember:S)
(Initialized Student:S)
>>> members = [t,s]
>>> for member in members:
        member.tell
<bound method Teacher.tell of <__main__.Teacher instance at 0x01CF5710>>
<bound method Student.tell of <__main__.Student instance at 0x01CF5670>>
>>> for member in members:
    member.tell()
Age:32 Name:T
Salary:5000
Age:22 Name:S
Marks:98
```

SchoolMember.__init__(self,name,age) 因为 Python 不会自动调用父类的

constructor（构造函数），所以得亲自专门调用它。还有另一个种方法是，在程序最开始输入__metaclass__ = type（__metaclass__ = type 表示下面建的类是新式类，而 super 函数只在新式类中起作用），然后在子类中输入 super(SchoolMember,self,name,age) .__init__()。

当然，也存在多继承。如果在继承元组中不止一个父类，就称为多重继承。Python 同样有限的支持多继承形式。多继承的类定义形如下例：

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    。。。
    <statement-N>
```

私有变量

只能从对像内部访问的“私有”实例变量，在 Python 中不存在。然而，也有一个变通的访问用于大多数 Python 代码 :以一个下划线开头的命名(例如 :__spam ）会被处理为 API 的非公开部分（无论它是一个函数、方法或数据成员 ）。它会被视为一个实现细节，无需公开。

需要注意的是编码规则设计为尽可能的避免冲突，被认作为私有的变量仍然有可能被访问或修改。在特定的场合它也是有用的，比如调试的时候。

要注意的是代码传入 exec ， eval() 或 execfile() 时不考虑所调用的类的类名，视其为当前类，这类似于 global 语句的效应， 已经按字节编译的部分也有同样的限制。这也同样作用于 getattr() ,setattr() 和 delattr ,像直接引用__dict__一样。

```
>>> class computer:
```

```
        __color = 'red'
        def __init__(self,name):
            self.computer_name = name
        def start(self,x):
            print 'The %s %s %s is start.' %
(self.__color,x,self.computer_name)
>>> c = computer('b')
>>> c.start('a')
The red a b is start.
```

进行如下操作，只会得到值 b 是因为__color 为类 computer 的私有属性：

```
>>> print c.computer_name
b
>>> print c.__color
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    print c.__color
AttributeError: computer instance has no attribute '__color'
```

析构方法：

指编译对象时 ,自动执行的方法(如下 :删除对象 c 的时候会输出 GAME OVER)。

```
>>> class computer:
        def __del__(self):
            print 'GAME OVER'
```

```
>>> c = computer()
>>> del c
GAME OVER
```

property 函数

在新式类中使用 property 函数而不是访问器方法。

```
>>> __metaclass__ = type
>>> class Rectangle:
        def __init__(self):
            self.width = 0
            self.height = 0
        def setSize(self,seze):
            self.width,self.height = size
        def getSize(self):
            return self.width,self.height
        size = property(getSize,setSize)
```

property 函数创建了一个属性，其中访问器函数被用作参数（先是取值，然后是赋值），这个属性命名为 size。这样一来就不再需要担心是怎么实现的了，可以用同样的方式处理 width，height 和 size。

```
>>> r = Rectangle()
>>> r.width = 10
>>> r.height = 5
>>> r.size
```

```
(10, 5)
>>> r.size
(10, 5)
>>> r.width
10
>>> r.size = 150,30
>>> r.size
(150, 30)
```

静态方法和类方法

静态方法和类成员方法分别在创建时分别装入 StateMethod 和 ClassMethod 类型的对象中。静态方法的定义没有 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调用。但 cls 参数是自动被绑定到类的。

```
>>> __metaclass__ = type
>>> class myclass:
        def smeth():
            print 'This is a static method'
        smth = staticmethod(smeth)
        def cmeth(cls):
            print 'This is a class method',cls
        cmeth = classmethod(cmeth)
```

装饰器：@

它能够对任何可调用的对象进行包装，既能够用于方法也能用于函数。

```
>>> __metclass__ = type
>>> class myclass:
        @staticmethod
        def smeth():
            pass
        @classmethod
        def cmeth():
            pass
```

定以后，可直接使用两种方法：myclss.smethn()和 myclass.cmeth()。

迭代器

迭代的意思就是重复做一些事情很多次。__ite__方法返回一个迭代器(iterator)，所谓的迭代器就是具有 next 方法（这个方法在调用时不需要任何参数）的对象。现在你可能注意到大多数容器对象都可以用 for 遍历：

```
for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
```

```
for line in open("myfile.txt"):
    print line
```

这种形式的访问清晰、简洁、方便。迭代器的用法在 Python 中普遍而且统一。在后台，for 语句在容器对象中调用 iter()。该函数返回一个定义了 next()方法的迭代器对象，它在容器中逐一访问元素。没有后续的元素时，next()抛出一个 StopIteration 异常通知 for 语句循环结束。以下是其工作原理的示例：

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x02104E90>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()
Traceback (most recent call last):
  File "<pyshell#54>", line 1, in <module>
    it.next()
StopIteration
```

了解了迭代器协议的后台机制，就可以很容易的给自己的类添加迭代器行为。

定义一个__iter__ ()方法，使其返回一个带有 next()方法的对象。如果这个类已经定义了 next()，那么__iter__ ()只需要返回 self：

```
>>> class Reverse:
        def __init__(self,data):
            self.data = data
            self.index = len(data)
        def __iter__(self):
            return self
        def next(self):
            if self.index == 0:
                raise StopIteration
            self.index -= 1
            return self.data[self.index]
>>> for char in Reverse('spam'):
        print char
m
a
p
s
```

使用时可能是计算一个值时获取一个值——而不是通过列表一次性获取所有值。如果有很多值，列表就会占用太多的内存。但还有其他的理由：使用迭代器更通用、更简单、更优雅。

一个实现了__iter__方法的对象时可迭代的，一个实现了 next 方法的对象则是迭代器。

```
>>> class Fibs:
        def __init__(self):
            self.a = 0
            self.b = 1
        def __iter__(self):
            return self
        def next(self):
            self.a,self.b = self.b,self.a + self.b
            return self.a
>>> fib = Fibs()
>>> for f in fib:
        if f > 1000:    #在斐波那契数列中比 1000 大的最小整数。
        print f
        break
1597
```

生成器：一种用普通函数语法定义的迭代器

生成器是创建迭代器的简单而强大的工具。它们写起来就像是正规的函数，需要返回数据的时候使用 yield 语句。每次 next()被调用时，生成器回复它脱离的位置（它记忆语句最后一次执行的位置和所有的数据值）。

如何创建生成器：

首先创建一个展开嵌套列表的函数：

```
nested = [[1,2],[3,4],[5]]
```

函数应该按顺序打印出列表中的数字，解决办法如下：

```
def flatten(nested):
    for sublist in nested:
        for element in sublist:
            yield element
```

任何包含 yield 语句的函数称为生成器。它不像 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 语句），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些开始执行。接下来可以通过在生成器上碟带来使用所有的值：

```
>>> for num in flatten(nested):
        print num
1
2
3
4
5
>>> list(flatten(nested))
[1, 2, 3, 4, 5]
```

以下示例同样演示了生成器可以很简单的创建出来：

```
>>> def reverse(data):
```

```
            for index in range(len(data)-1,-1,-1):
                yield data[index]
>>> for char in reverse('spam'):
            print char
m
a
p
s
```

前一节中描述了基于类的迭代器，它能作的每一件事生成器也能作到。因为自动创建了__iter__ ()和 next()方法，生成器显得如此简洁。

另一个关键的功能在于两次执行之间，局部变量和执行状态都自动的保存下来。这使函数很容易写，而且比使用 self.index 和 self.data 之类的方式更清晰。

除了创建和保存程序状态的自动方法，当发生器终结时，还会自动抛出 StopIteration 异常。综上所述，这些功能使得编写一个正规函数成为创建迭代器的最简单方法。

循环生成器：生成器表达式（生成器推导式）

有时简单的生成器可以用简洁的方式调用，就像不带中括号的链表推导式。这些表达式是为函数调用生成器而设计的。生成器表达式（生成器推导式）比完整的生成器定义更简洁，但是没有那么多变，而且通常比等价的链表推导式更容易记。

生成器表达式和列表推导式的工作方式类似，只不过返回的不是列表而是生成器（并且不会立刻进行循环）。所以返回的生成器允许像下面这样一步一步的进行计算：

```
>>> g = ((i+2)**2 for i in range(2,27))
>>> g.next()
16
```

和列表推导式不同的是就是普通圆括号的使用方式。生成器表达式可以在当前的圆括号内直接是用，例如在函数调用中，不用增加另外一对圆括号，换句话说，可以像下面这样编写代码：

```
>>> sum(i*i for i in range(10))
285
>>> xvec = [10,20,30]
>>> yvec = [7,5,3]
>>> sum(x*y for x,y in zip(xvec,yvec))
260
>>> sum(x*y for x in xvec for y in yvec)
900
>>> from math import pi, sin
>>> sine_table = dict((x, sin(x*pi/180)) for x in range(0, 91))
>>> unique_words = set(word for line in page for word in line.split())
>>> valedictorian = max((student.gpa, student.name) for student in
graduates)
>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1,-1,-1))
['f', 'l', 'o', 'g']
```

递归生成器（将嵌套列表返回成一个单一列表）

```
def flatten(nested):
    try:    #不要迭代类似字符串的对象
        try:
            nested + ''
        except TypeError:
            pass
        else:
            raise TypeError
        for sublist in nested:
            for element in flatten(sublist):
                yield element
    except TypeError:
        yield nested
>>> list(flatten([[[1],2],3,4,[5,[6,7]],8]))
[1, 2, 3, 4, 5, 6, 7, 8]
>>> list(flatten(['foo',['bar',['bza']]]))
['foo', 'bar', 'bza']
```

通用生成器

生成器是一个包含 yield 关键字的函数。当它被调用时，在函数体中的代码不会被执行，而会返回一个迭代器。每次请求一个值，就会执行生成器中的代码，直到遇到一个 yield 或者 return 语句。yield 语句意味着应该生成一个值。return 语句

意味着生成器要停止执行（不再生成任何东西，return 语句只有在一个生成器中使用才能进行无参数调用）。

换句话说，生成器由两部分组成：生成器的函数和生成器的迭代器。生成器的函数是用 def 语句定义的，包含 yield 的部分，生成器的迭代器是这个函数返回的部分。按一种不是很准确的说法，两个实体经常被当做一个，合起来叫做生成器。

```
def simple_generator():
    yield
>>> simple_generator
<function simple_generator at 0x016FEB70>
>>> simple_generator()
<generator object simple_generator at 0x01DC0148>
```

生成器的函数返回的迭代器可以像其他的迭代器那样使用。

生成器方法

使用 send 方法（而不是 next 方法）只有在生成器挂起之后才有意义（也就是说 yield 函数第一次被执行之后）。如果真想对刚刚启动的生成器使用 send 方法，那么可以将 None 作为其参数进行调用。

```
>>> def repeater(value):
        while True:
            new = (yield value)
            if new is not None:value = new
>>> r = repeater(42)
>>> r.next()
```

```
42
>>> r.send('Hello,world!')
'Hello,world!'
```

模拟生成器：（步骤）

一、result = []

二、yield some_expression

三、result.append(some_expression)

四、return result

下面是 flatten 生成器用普通的函数重写的版本：

```
def flatten(nested):
    result = []
    try:
        try:
            nested + ''
        except TypeError:
            pass
        else:
            raise TypeError
        for sublist in nested:
            for element in flatten(sublist):
                result.append(element)
    except TypeError:
```

```
            result.append(nested)
        return result
>>> flatten([[[1],2],3,4,[5,[6,7]],8])
[1, 2, 3, 4, 5, 6, 7, 8]
```

错误和异常

我们可以使用 try...except...语句来处理异常。我们把可能引发错误的语句放在 try 块中，而把我们的错误处理语句放在 except 块中。

Python 中(至少)有两种错误 :语法错误和异常(syntax errors and exceptions)。

语法错误

语法错误，也称作解释错误，可能是学习 Python 的过程中最容易犯的：

```
>>> while True print 'Hello world'
File "<stdin>", line 1, in ?
while True print 'Hello world'
^
SyntaxError: invalid syntax
```

解析器会重复出错的行，并在行中最早发现的错误位置上显示一个小“箭头”。错误（至少是被检测到的）就发生在箭头指向的位置。示例中的错误表现在关键字 print 上，因为在它之前少了一个冒号。同时也会显示文件名和行号，这样你就可以知道错误来自哪个脚本，什么位置。

异常（Exceptions）

即使是在语法上完全正确的语句，尝试执行它的时候，也有可能会发生错误。

在程序运行中检测出的错误称之为异常，它通常不会导致致命的问题。大多数异常不会由程序处理，而是显示一个错误信息：

```
>>> 10 * (1/0)
Traceback (most recent call last):
File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

错误信息的最后一行指出发生了什么错误。异常也有不同的类型，异常类型做为错误信息的一部分显示出来：示例中的异常分别为零除错误、命名错误、类型错误。打印错误信息时，异常的类型作为异常的内置名显示。对于所有的内置异常都是如此，不过用户自定义异常就不一定了（尽管这是一个很有用的约定）。标准异常名是内置的标识（没有保留关键字）。

这一行后一部分是关于该异常类型的详细说明，这意味着它的内容依赖于异常类型。

错误信息的前半部分以堆栈的形式列出异常发生的位置。通常在堆栈中列出了

源代码行，然而，来自标准输入的源码不会显示出来。

bltin-exceptions 列出了内置异常和它们的含义。内建异常都在 exceptions 中，可通过 dir 函数列出模块内容：

```
>>> import exceptions
>>> dir(exceptions)
```

控制异常

可以编写程序来控制已知的异常。参见下例，此示例要求用户输入信息，一直到得到一个有效的整数为止，而且允许用户中断程序；需要注意的是用户生成的中断会抛出 KeyboardInterrupt 异常：

```
>>> while True:
        try:
            x = int(raw_input('Enter a number: '))
            break
        except ValueError:
            print 'Oops!That was no valid number.Try again...'


    Enter a number: k
    Oops!That was no valid number.Try again...
    Enter a number: 8
```

try 语句按如下方式工作：

首先，执行 try 子句（在 try 和 except 关键字之间的部分）。如果没有异常发生，except 子句在 try 语句执行完毕后就被忽略了。

如果在 try 子句执行过程中发生了异常，那么该子句其余的部分就会被忽略。 如果异常匹配于 except 关键字后面指定的异常类型，就执行对应的 except 子句。然后继续执行 try 语句之后的代码。

如果发生了一个异常，在 except 子句中没有与之匹配的分支，它就会传递到上一级 try 语句中。如果最终仍找不到对应的处理语句，它就成为一个未处理异常，终止程序运行，显示提示信息。

一个 try 语句可能包含多个 except 子句，分别指定处理不同的异常。至多只会有一个分支被执行。异常处理程序只会处理对应的 try 子句中发生的异常，在同一个 try 语句中，其他子句中发生的异常则不作处理。一个 except 子句可以在括号中列出多个异常的名字，例如：

```
... except (RuntimeError, TypeError, NameError):
...         pass
```

最后一个 except 子句可以省略异常名，把它当做一个通配项使用。一定要慎用这种方法，因为它很可能会屏蔽掉真正的程序错误，使人无法发现！它也可以用于打印一行错误信息，然后重新抛出异常（可以使调用者更好的处理异常）：

```
import sys
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
    except IOError as (errno, strerror):
    print "I/O error({0}): {1}".format(errno, strerror)
```

```
except ValueError:
print "Could not convert data to an integer."
except:
print "Unexpected error:", sys.exc_info()[0]
raise
```

try ... except 语句可以带有一个 else 子句，该子句只能出现在所有 except 子句之后。当 try 语句没有抛出异常时，需要执行一些代码，可以使用这个子句。例如：

```
for arg in sys.argv[1:]:
try:
f = open(arg, 'r')
except IOError:
print 'cannot open', arg
else:
print arg, 'has', len(f.readlines()), 'lines'
f.close()
```

使用 else 子句比在 try 子句中附加代码要好 ,因为这样可以避免 try ... except 意外的截获本来不属于它们保护的那些代码抛出的异常。

发生异常时，可能会有一个附属值，作为异常的参数存在。这个参数是否存在、是什么类型、依赖于异常的类型。

在异常名（列表）之后，也可以为 except 子句指定一个变量。这个变量绑定于一个异常实例 ,它存储在 instance.args 的参数中。为了方便起见 ,异常实例定义了__str__()，这样就可以直接访问过打印参数而不必引用.args。

这种做法不受鼓励。相反，更好的做法是给异常传递一个参数（如果要传递多个参数，可以传递一个元组），把它绑定到 message 属性。一旦异常发生，它会在抛出前绑定所有指定的属性：

```
>>> try:
...     raise Exception('spam', 'eggs')
...     except Exception as inst:
...         print type(inst)    # the exception instance
...         print inst.args     # arguments stored in .args
...         print inst    # __str__ allows args to printed directly
...         x, y = inst    # __getitem__ allows args to be unpacked directly
...         print 'x =', x
...         print 'y =', y
...
<type 'exceptions.Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

对于未处理的异常，如果它有一个参数，那么就会作为错误信息的最后一部分（明细）打印出来。

异常处理句柄不止可以处理直接发生在 try 子句中的异常，即使是其中（甚至是间接）调用的函数，发生了异常，也一样可以处理。例如：

```
>>> def this_fails():
        x = 1/0
>>> try:
        this_fails()
    except ZeroDivisionError as detail:
        print 'Handling run-time error:', detail
Handling run-time error: integer division or modulo by zero
```

抛出异常

程序员可以用 raise 语句强制指定的异常发生。例如：

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
File "<stdin>", line 1, in ?
NameError: HiThere
```

要抛出的异常由 raise 的唯一参数标识。它必需是一个异常实例或异常类（继承自 Exception 的类）。

如果你需要明确一个异常是否抛出，但不想处理它，raise 语句可以让你很简单的重新抛出该异常：

```
>>> try:
        raise NameError('HiThere')
    except NameError:
        print 'An exception flew by!'
        raise
```

```
An exception flew by!
Traceback (most recent call last):
File "<stdin>", line 2, in ?
NameError: HiThere
```

用户自定义异常

在程序中可以通过创建新的异常类型来命名自己的异常。异常类通常应该直接或间接的从 Exception 类派生，例如：

```
>>> class MyError(Exception):
        def __init__(self, value):
            self.value = value
        def __str__(self):
            return repr(self.value)
>>> try:
        raise MyError(2*2)
    except MyError as e:
        print 'My exception occurred, value:', e.value
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

在这个例子中，:class:Exception 默认的__init__ () 被覆盖。新的方式简单的创

建 value 属性。这就替换了原来创建 args 属性的方式。

异常类中可以定义任何其它类中可以定义的东西，但是通常为了保持简单，只在其中加入几个属性信息，以供异常处理句柄提取。如果一个新创建的模块中需要抛出几种不同的错误时，一个通常的作法是为该模块定义一个异常基类，然后针对不同的错误类型派生出对应的异常子类：

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass
class InputError(Error):
    """Exception raised for errors in the input.
    Attributes:
    expr -- input expression in which the error occurred
    msg -- explanation of the error
    """
    def __init__(self, expr, msg):
        self.expr = expr
        self.msg = msg
class TransitionError(Error):
    """Raised when an operation attempts a state transition that's
not
    allowed.
    Attributes:
```

```
        prev -- state at beginning of transition
        next -- attempted new state
        msg -- explanation of why the specific transition is not allowed
        """
        def __init__(self, prev, next, msg):
            self.prev = prev
            self.next = next
            self.msg = msg
```

与标准异常相似，大多数异常的命名都以“Error”结尾。

定义清理行为

try 语句还有另一个可选的子句，目的在于定义在任何情况下都一定要执行的功能。例如：

```
>>> try:
        raise KeyboardInterrupt
    finally:
        print 'goodbye,word'
goodbye,word

Traceback (most recent call last):
  File "<pyshell#43>", line 2, in <module>
    raise KeyboardInterrupt
KeyboardInterrupt
```

不管有没有发生异常，finally 子句在程序离开 try 后都一定会被执行。当 try 语句中发生了未被 except 捕获的异常（或者它发生在 except 或 else 子句中），在 finall 子句执行完后它会被重新抛出。try 语句经由 break，:keyword:continue 或 return 语句退出也一样会执行 finally 子句。以下是一个更复杂些的例子：

```
>>> def divide(x,y):
        try:
            result = x / y
        except ZeroDivisionError:
            print 'Division by zero!'
        else:
            print 'result is {}'.format(result)
        finally:
            print 'executing finally clause'
>>> divide(2,1)
result is 2
executing finally clause
>>> divide(2,0)
Division by zero!
executing finally clause
>>> divide('2','1')
executing finally clause
```

```
Traceback (most recent call last):
  File "<pyshell#57>", line 1, in <module>
    divide('2','1')
  File "<pyshell#54>", line 3, in divide
    result = x / y
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

如你所见，finally 子句在任何情况下都会执行。TypeError 在两个字符串相除的时候抛出，未被 except 子句捕获，因此在 finally 子句执行完毕后重新抛出。在真实场景的应用程序中，finally 子句用于释放外部资源（文件或网络连接之类的），无论它们的使用过程中是否出错。

预定义清理行为

有些对象定义了标准的清理行为，无论对象操作是否成功，不再需要该对象的时候就会起作用。以下示例尝试打开文件并把内容打印到屏幕上：

```
for line in open("myfile.txt"):
    print line
```

这段代码的问题在于在代码执行完后没有立即关闭打开的文件。这在简单的脚本里没什么，但是大型应用程序就会出问题。with 语句使得文件之类的对象可以确保总能及时准确地进行清理：

```
with open("myfile.txt") as f:
    for line in f:
        print line
```

语句执行后，文件 f 总会被关闭，即使是在处理文件中的数据时出错也一样。

其它对象是否提供了预定义的清理行为要查看它们的文档。

模块

模块

模块基本上就是一个包含了所有你定义的函数和变量的文件，为了在其它程序中重用模块，模块的文件名必须以.py 为扩展名。每一个 python 文件就是一个模块。

import ... 然后可以使用：模块名.函数名

尽量少使用 from...import...以避免名称的冲突。

如果你退出 Python 解释器重新进入，以前创建的一切定义（变量和函数）就全部丢失了。因此，如果你想写一些长久保存的程序，最好使用一个文本编辑器来编写程序，把保存好的文件输入解释器，我们称之为创建一个脚本。Python 提供了一个方法可以从文件中获取定义，在脚本或者解释器的一个交互式实例中使用。这样的文件被称为模块；模块中的定义可以导入到另一个模块或主模块中。

如果你想要直接调用函数，通常可以给它赋一个本地名称：

```
>>> fib = fibo.fib   #变量名=模块名.函数名
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

出于性能考虑，每个模块在每个解释器会话中只导入一遍。因此，如果你修改了你的模块，需要重启解释器——或者，如果你就是想交互式的测试这么一个模块，可以用 reload() 重新加载：

modulename = reload(modulename) 。

Python 中的模块存储在 site-packages 目录中，只要将模块放入其中，所有程序就都能将其导入了。

pprint

pprint 是个相当好的打印函数，能够提供更加智能的打印输出：

```
>>> import sys,pprint
>>> pprint.pprint(sys.path)
['C:/Users/lenovo/Desktop',
 'E:\\\xd1\xa7\xcf\xb0\\python\\python2.7\\Lib\\idlelib',
 'C:\\Windows\\system32\\python27.zip',
 。。。

'E:\\\xd1\xa7\xcf\xb0\\python\\python2.7\\lib\\site-packages\\wx-3.0-msw']
```

dir()函数

内置函数 dir()用于按模块名搜索模块定义，它返回一个字符串类型的存储列表：

```
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__',
'__package__', 。。。, 'winver']
```

无参数调用时，dir()函数返回当前定义的命名：

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'sys']
```

dir()不会列出内置函数和变量名。如果你想列出这些内容，它们在标准模块

builtin 中定义：

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError', 。。。, 'zip']
```

__name__

每个模块都有一个名称，在模块中可以通过语句来找出模块的名字（模 块的模块名（做为一个字符串）可以由全局变量__name__得到）。这在一个场合特别有用——当一个模块被第一次输入的核实后，这个模块的主块将被运行。加入我们只想在程序本身使用的时候运行主块，而在它被别的模块输入的时候不运行主块，我们就可以通过__name__属性完成。在文件中：

```
if __name__ == '__main__':
    print 'This program is being run by itself'
else:
    print 'I am being imported by another module'
```

当第一次在别的程序中输入 import 模块名的时候，会返回：

```
I am being imported by another module
```

__name__作为模块的内置属性，简单点说，就是.py 文件的调用方式。如果__name__ == '__main__' 说明该模块被用户单独运行，我们可以进行相应恰当的操作。

sys 模块

sys.path 返回模块的搜索路径，初始化时使用 PYTHONPATH 环境变量的值。

sys.path.append(path)导入新的路径。但在退出后，自己添加的路径就自动消失了。

只有 sys.path 中路径里的文件夹能用 import 执行。

练习一：

在 pythonlianxi 下新建文件夹 functions，在 functions 中新建 commo.py：

```
def com():
    print 'hello'
```

同时另建一个空文件：__init__.py，使用 sys.path.append('E:\pythonlianxi')添加新的路径，然后 from functions.common import com，此时在 functions 文件夹中多了__init__.pyc 和 common.pyc 文件。此时再运行 com()会返回 hello。

```
>>> sys.path.append('E:\pythonlianxi')
>>> from functions.common import com
>>> com()
Hello
```

重新初始化 sys（Ctrl+F6），在__init__.py 文件里写入 from math import ceil，重新导入路径：sys.path.append('E:\pythonlianxi')，然后输入如下命令：

```
>>> import functions
>>> functions.ceil(4.3)
```

可以得到结果：5.0

练习二：

sys.exit(0)的作用是退出程序。

新建 sysexit.py 文件：

```
print 'a'
import sys
sys.exit(0)
print 'b'
```

运行该文件，发现只得到 a，所以 sys.exit(0)经常用于调试。

os 模块

os.name　指示你正在使用的平台 Windows 显示的是“nt”

os.getcwd()　返回当前 Python 脚本工作的目录路径

os.getenv(path)　获取当前系统的环境变量

os.listdir()　返回指定目录下的所有文件和目录名，“./”表示当前工作目录

os.remove()　用来删除一个文件：

```
os.remove(r'E:\pythonlianxi\functions\oo.txt')
```

os.removedirs()　删除一个空文件夹，可递归删除空文件夹：

```
os.removedirs('E:\\pythonlianxi\\mkdir\\1\\2')只能删除空文件夹 2。
os.removedirs(r'E:\pythonlianxi\mkdir\1\2')可以递归删除所有文件夹。
os.removedirs(r'E:\pythonlianxi\mkdir')不能删除该递归空文件夹。
```

os.rmdir()　只能删除单个空文件夹

os.system()　用来运行 shell 命令，例如： os.system('cmd')

os.linesep　给出当前平台使用的行终止符

os.path.split()　返回一个路径的目录名和文件名（一个元组）：

```
>>> os.path.split(r'E:\pythonlianxi\2\1.txt')
```

('E:\\pythonlianxi\\2', '1.txt')

os.path.isfile 和 os.path.isdir 分别检验给出的路径是一个文件还是一个目录，不存在或不是都返回 False

os.path.exists()　用来检验给出的路径是否存在

os.path.getsize()　返回文件大小

os.path.abspath()　获得当前文件的绝对路径

os.path.splitext()　分割文件，返回一个元组，包括文件名与扩展名

os.stat()　返回当前文件或者目录的状态

os.mkdir()　创建一个文件夹，默认权限 0777，文件夹存在则不能创建

os.makedirs()　可以递归创建目录

os.rename('1.txt','2.txt')　重命名文件，前提是先关闭 1.txt

os 模块包含普遍的操作系统的功能：

os.chdir(dirname)：改变工作目录到 dirname

os.path.normpath(path)：规范 path 字符串形式

os.path.basename(path)：返回文件名

os.path.dirname(path)：返回文件路径

应该用 import os 风格而非 from os import *。这样可以保证随操作系统不同而有所变化的 os.open()不会覆盖内置函数 open()。

在使用一些像 os 这样的大型模块时内置的 dir()和 help()函数非常有用：

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
```

```
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

练习一：递归删除文件夹

```
import os
def rmdir(dirs):
    if os.path.isdir(dirs):
        files = os.listdir(dirs)
        for f in files:
            f = dirs + '/' + f
            if os.path.isfile(f):
                try:
                    os.remove(f)
                except:
                    print '请检查文件' + f +'是否有删除权限'
            else:
                rmdir(f)
        os.rmdir(dirs)
    else:
        print 'dir is not extists'
dirs = raw_input('enter:')
rmdir(dirs)
```

练习二：递归创建文件夹

```
#创建 D:\dir//2014\01/21
import re
def mkdir(dirs):
    os.path.normpath(dirs)
    p = re.compile(r'\\')
    dirs = p.sub(r'/',dirs)
    print 'the dir is: ' + dirs
    lists = dirs.split('/')
    print 'the splitedlist is: '
    print (lists)
    d1 = ''
    if (lists[0].find(':') != -1):
        print 'the list[0] is: ' + lists[0]
        d1 = lists[0] + '/'
        del lists[0]
        for d in lists:
            if os.path.isdir(d1 + d):
                pass
            else:
                os.mkdir(d1 + d)
                d1 += d + '/'
```

```
dirs = raw_input('enter:')
mkdir(dirs)
```

删除非空目录，可使用 shutil 模块中的 rmtree 函数。

time 模块

time 模块提供各种时间操作的模块。一般有两种表示时间的方式：第一种是时间戳的方式（相对于 1970.1.1 00:00:00 以秒计算的偏移量），时间戳是唯一的；第二种以数组的形式表示(时间元组 :struct_time),共有九个元素 ,同一个时间戳的 struct_time 会因为时区的不同而不同（年、月、日、时、分、秒、周、儒历日、夏令时）。

周：当周一为 0 时，范围 0~6　　秒：范围 0~61　　分：0~59

儒历日：范围 1~366　　时：0~23　　日：1~31　　月：1~12

夏令时：0、1 或-1（夏令时使用-1 时，mktime 就会工作正常）

举例：

字符串日期：'Tue Nov 18 21:05:53 2014'

时间元组：(1999,1,1,1,1,1,1,-1)

时间戳：1416317837.0

0.time.time（ ）返回当前时间的时间戳：

```
>>> time.time()
1416318971.657
```

1.asctime（[tuple]）将时间元组转换为字符串，默认为当前时间：

```
>>> time.asctime()
'Tue Nov 18 21:05:53 2014'
```

```
>>> time.asctime((1999,1,1,1,1,1,1,1,-1))
'Tue Jan 01 01:01:01 1999'
```

2.clock（ ）在第一次调用的时候，返回的是程序运行的实际时间；第二次调用开始，返回的是第一次调用，到这次调用的时间间隔：

```
>>> time.clock()
2.851509903008742e-06
>>> time.clock()
9.749830192585275
```

3.sleep（ ）线程推迟指定的时间运行，经过测试，时间为秒：

```
import time
if __name__ == '__main__':
    time.sleep(1)
    print 'clock1:%s' % time.clock()
    time.sleep(1)
    print 'clock2:%s' % time.clock()
    time.sleep(1)
    print 'clock3:%s' % time.clock()
```

运行后返回：

```
clock1:2.28120792241e-06
clock2:1.01390453259
clock3:2.02783643967
```

4.ctime（ ）将一个时间戳转换成一个时间字符串，默认为当前时间：

>>> time.ctime()

'Tue Nov 18 21:16:00 2014'

5.gmtime（ ）获得全球统一时间：

>>> time.gmtime()

gmtime([seconds])(tm_year, tm_mon, tm_mday, tm_hour, tm_min, tm_sec, tm_wda1, tm_yday, tm_isdst)

将一个时间戳转换成一个 ATC 时间（0 时区）的时间元组，如果 seconds 参数未输入，则以当前时间为转换标准。

6.localtime（[seconds]）将一个时间戳转换成一个当前时区的时间元组，如果 seconds 参数未输入，则以当前时间为转换标准：

>>> time.localtime(1111111.0)

time.struct_time(tm_year=1970, tm_mon=1, tm_mday=14, tm_hour=4, tm_min=38, tm_sec=31, tm_wday=2, tm_yday=14, tm_isdst=0)

7.mktime（[tuple]）将一个时间元组转换为时间戳：

>>> time.mktime(time.localtime())

1416317837.0

>>> time.mktime((1999,1,1,1,1,1,1,1,-1))

915123661.0

8.strftime（ ）将指定的时间元组（默认当前时间），根据指定的格式话字符串输出，python 中时间日期格式化符号：strftime(format[,tuple])：

%y：两位数的年份表示（00-99） %Y：四位数的年份表示（0000-9999）

%m：月份（01-12） %d：月内中的一天（0-31）

%H：24 小时制小时数（0-23）　　%I：12 小时制小时数（01-12）

%M：分钟数（00-59）　　%S：秒（00-59）

%a：本地简化星期名称　　%A 本地完整星期名称

%b：本地简化的月份名称　　%B 本地完整的月份名称

%c：本地相应的日期表示和时间表示　　%j：年内的一天（001-366）

%p：本地 A.M.或 P.M.的等价符

%U：一年内的星期数（00-53），星期天为星期的开始

%w：星期（0-6），星期天为星期的开始

%W：一年中的星期数（00-53），星期一为星期的开始

%x：本地相应的日期表示　　%X：本地相应的时间表示

%Z：当前时区的名称　　%%：%本身

```
>>> time.strftime('%y.%m.%d %a %b %H:%M:%S')
'14.11.18 Tue Nov 22:30:34'
>>> time.strftime('%Y.%m.%d %A %B %I:%M:%S')
'2014.11.18 Tuesday November 10:30:52'
>>> time.strftime('%c')
'11/18/14 22:31:14'
>>> time.strftime('%x')
'11/18/14'
>>> time.strftime('%X')
'22:32:27'
>>> time.strftime('%a %b %d %X %Y')
```

'Tue Nov 18 22:35:08 2014'

>>> time.strftime('%Z')

'\xd6\xd0\xb9\xfa\xb1\xea\xd7\xbc\xca\xb1\xbc\xe4'

>>> #! gbk

>>> print(time.strftime('%Z'))

中国标准时间

9.strptime（string，format）将时间字符串根据指定的格式化符转换成时间元组。

文件处理

文件

你可以通过创建一个 file 类的对象来打开一个文件，分别使用 file 类的 read、readline 或 write 方法来恰当地读写文件。对文件的读写能力依赖于你在打开文件时指定的模式。最后，当你完成对文件的操作的是偶，你调用 close 方法来告诉 Python 我们完成了对文件的使用。

file = open("data.txt","r") open 函数的第一个参数是一个字符串，它含有需要打开的文件的名称。第二个参数用于指定一个模式。共有三种选择，“r”表示读取（默认就是“r”），“w”表示写入，“a”表示追加，“r+”表示以读写方式打开。（写入时将会抹去文件原有的内容，而追加则是在文件的末尾加上新的数据）

open 函数的执行结果并不是文件的内容。其实 open 函数所返回的是一个文件对象，该对象拥有访问文件内容所需的方法。

调用 read(size)方法时，如果没有提供参数 size，则它将读取文件中的所有数据，并以字符串的形式返回。如果提供一个正整数参数，则它只会读取那么多的字

符；当文件中没有其他数据可读取时，该方法会返回一个空字符串。

如果文件时文本，那么我们有可能希望一行一行的对其进行处理，我们可以使用 readline 方法，其功能为读取文件中的下一行文件。这里的“一行”指的是，直到下一个“行尾标记”之前的所有字符，并且包含该行尾标记。从文件中读取单独一行，字符串结尾会自动加上一个换行符（\n），只有当文件最后一行没有以换行符结尾时，这一操作才会被忽略。readline 方法在文件中没有可读取数据时也会返回一个空字符串，如果返回的是”\n”表示的是空行。

readline 方法最美妙的一点是，当文件对象用于 for 循环时，Python 会自动调用该方法。文件内容：

Mercury

Venus

Earth

Mars

打开该文件，然后输出每一行的长度：

```
>>> file = open("e:\data.txt","r")
>>> for line in file:
        print len(line)

8       #一共 7 个字符，返回长度是 8，是因为从文件中读取的每一行的末尾都有一个行结束符。可以用 string.strip 来去掉这个字符，它将去除字符串首尾两端的空白符（空格、制表符以及换行符等）并返回其结果。
6
```

6

4

用 string.strip：

```
>>> file = open('e:\data.txt','r')
>>> for line in file:
        print len(line.strip())


7
5
5
4
>>>file.close（ ）
```

每次使用 readline，都会返回新的内容：

```
>>> file = open('e:\data.txt','r')
>>> file.readline()
'mercury\n'
>>> file.readline()
'venus\n'
>>> file.readline()
'earth\n'
>>> file.readline()
'mars'
```

```
>>> file.readline()
''      #当没有内容时返回空字符串
>>> file.readline()
''
```

f.readlines()返回一个列表，其中包含了文件中所有的数据行。如果给定了sizehint 参数，就会读入多于一行的比特数，从中返回多行文本。这个功能通常用于高效读取大型行文件，避免了将整个文件读入内存。这种操作只返回完整的行：

```
>>> f.readlines()
['This is the first line of the file.\n', 'Second line of the file\n']
```

写入文件

```
>>> file = open('e:\data.txt','w')   #覆盖原内容
>>> file.write('tt')
>>> file.close()
>>> file = open('e:\data.txt')
>>> file.readline()
'tt'
>>> file.close()
>>> file = open('e:\data.txt','a')   #在原内容后面追加内容
>>> file.write('tt')
>>> file.close()
>>> file = open('e:\data.txt')
>>> file.readline()
```

'tttt'

f.write(string)将 string 的内容写入文件，返回 None：

```
>>> f.write('This is a test\n')
```

如果需要写入字符串以外的数据，就要先把这些数据转换为字符串：

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
```

f.tell()

返回一个整数，代表文件对象在文件中的指针位置，该数值计量了自文件开头到指针处的比特数。需要改变文件对象指针话话，使用 f.seek(offset,from what)。指针在该操作中从指定的引用位置移动 offset 比特，引用位置由 from what 参数指定。from what 值为 0 表示自文件起始处开始，1 表示自当前文件指针位置开始，2 表示自文件末尾开始。from what 可以忽略，其默认值为零，此时从文件头开始：

```
>>> f = open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)
# Go to the 6th byte in the file
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Go to the 3rd byte before the end
>>> f.read(1)
'd'
```

文件使用完后，调用 f.close()可以关闭文件，释放打开文件后占用的系统资源。调用 f.close()之后，再调用文件对象会自动引发错误。

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

用关键字 with 处理文件对象是个好习惯。它的先进之处在于文件用完后会自动关闭，就算发生异常也没关系。它是 try-finally 块的简写。

```
>>> with open('/tmp/workfile', 'r') as f:
        read_data = f.read()
>>> f.closed
True
```