Python3.2.3 官方文档

译者 序

首先感谢客户小马哥，因为有他的打击，才会有今天这份文档。至今仍清楚地记得笔者在小马哥的面试过程中那段场景：

小马哥： “还会其他编程语言吗”

笔者：“不会”

小马哥： “听过 python 吗”

笔者： “听过”

小马哥： “了解 python 吗”

笔者： “不了解”

小马哥：“也就是说只知道 python 这个名字，对吧”

当听到这句话的，笔者的小心脏立刻刺激得受不了了。心里感觉这话就像问一个数学教授知道勾股定理吗，然而回答竟是只听过。当时笔者心里犹如九把刀左一刀右一刀在划啊，一划一道渠，一划一道渠。 当晚回去果断淡定不了了。开机、下载、编写，于是一个崭新 python 版的 helloword 出现了

<python code> print ’Hello World’ <python code>

然后自信满满地 run： 结果定眼一看，红灿灿一行： SyntaxError: invalid syntax

接下半个小时 debug 呀，debug 啊，debug 呀，debug.... 度娘完了问 google,问完中文换英文.. 那个郁闷啊，趋向无穷大。笔者的多少脂肪就这样无名地被燃烧了。于是低下眉头，突然计上心头，果断后面加个括号。竟然成功了，后来一查 api，“我靠”，python3.2 更改方法了，并不向下兼容。突然心中有千万只草泥马浩浩荡荡奔腾而过啊。

于是盯着电脑吐槽：“python,你丫的装什么牛逼，人家台湾教授说中国内陆的河南南一半地方都吃不起茶叶蛋，但俺作为更内陆的甘肃人今天还都干了两个蛋蛋。你等着，看哥怎么收拾你”。于是一个翻译 python 官方文档的计划产生了，经过两个月在伸手不见五指的夜晚鏖战，终于把 python 这货的 3.2.3 的英文 api 文档给翻译出来啦，于大家分享。

最后，由于笔者英文水平有限，有不当之处，请尽力吐槽，不过有个要求，吐槽一定要让笔者知道。 当然扔鞋也是欢迎的啦，扔两只那是相当的感动，要是扔两只男式 42 码的帆布鞋那直接泪崩。

译者邵向兵 博客： http://blog.csdn.net/sxb0841901116/article/

吐槽博客： http://blog.csdn.net/sxb0841901116

目录

Python3.2.3 官方文档教程

第一章 Python 初步介绍

在接下来的实例中，用标记符（>>> 和 … ）来区别输入和输出。想要重现这些实例，你必须输入标记符后面的所有内容。那些不以标记符开头的语句是输出语句。注意在一个例子中在同一行出现第二个标记符意味着你必须输入一行空格。它用来结束多行输入命令。

在这本说明书中的好多例子都包括注释，甚至有一些在交互提示符中换行。在 python 语言中的注释以固定字符#开始，并一直延续到本行的结尾。注释可能出现在每行的开始或者接下来的空格和代码，但不包含在字符串内。在字符串内的#号仅仅就是#符号。由于注释是用来解释说明代码而不能被 python 编译器所编译，当输入例子的时候注释可以被忽略。例如：

```
#this is the first comment
SPAM = 1                 # and this is the second comment
                         #... and now a third
STERING = "# This is not a comment."
```

1.1Python 用作计算器

让我们学习一些简单的 python 命令。 开始翻译和等待主提示符 >>> （这不会费时）

1.1.1 数字

解释器充当一个简单的计算器：在里面你可以输入一个表达式，它将会写出结果。 表达式语法就是简单的操作符 + - * 和 /, 这和大多数语言中(如 Pascal 和 C)用法一样。插入语可以用来分组，例如：

```
>>> 2+2
4
>>> # This is a comment  这是注释
... 2+2
```

```
4
>>> 2+2 # and a comment on the same line as code  在同一行代码中注释
4
>>> (50-5*6)/4
5.0
>>> 8/5 # Fractions aren't lost when dividing integers  当整数相除时候，分数不会丢失
1.6
```

备注： 你可能没精确看到相同的结果。 由于机器不同可能会导致浮点数的结果不同。以后我们会看到更多关于控制浮点数输出的例子。也可以参照浮点数运算： 问题和局限，这篇文件对浮点数和它们表示之间的细微差别将有个全面的讨论。

为了整数相除获得整数结果，有个抛弃了所有小数点的操作符//：

```
>>> # Integer division returns the floor:
... 7//3
2
>>> 7//-3
-3
```

“=”符号用来把一个特定值赋值给一个变量，后来，在接下来交互符号前面不显示结果：

```
>>> width = 20
>>> height = 5*9
>>> width * height
 900
```

一个值可以同时赋给好多个变量。

```
>>> x = y = z = 0 # Zero x, y and z
>>> x
0
>>> y
0
>>> z
0
```

变量在它们使用之前必须被定义（或者赋值），否者会抛出错误。

```
>>> # try to access an undefined variable  尝试访问一个没有定义的变量
... n
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

这里完全支持浮点运算，包含多种类型操作数会吧整型操作数变为浮点数：例如

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
    3.5
```

同时也支持复数运算，虚数写得时候以 j 或者 J 为前缀。复数一般写成不为 0 的实数部分（实数 + 虚数 J） 或者可以用复数方法 complex（实数，虚数）。

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0, 1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

复数常常用来表示两个浮点数，实数和虚数部分。可以同 z.real 和 z.imag 从复数 z 中提取实数和虚数。

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

针对复数和整数之间的转换方法（float(), int()）对复数不适用，它不能正确地把一个复数转化为一个实数. 用 abs(z)方法可以获得它的大小，or 用 z.real 来获得实数部分。

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a) # sqrt(a.real**2 + a.imag**2)
5.0
```

在交互模式中，最后一个打印出来的表示会赋值给变量_. 这就意味着当你用 python 作为左桌面计算器的时候，它有时很容易继续计算。例如：

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

这个变量只能被用户只读，不要尝试去给你设值，那样你将会与用内置变量同样的名字变量，重新创建一个独立的新本地变量。

1.1.2 字符串

除了数字，python 还可以操作字符串，字符串可以用多种方式被展现。它们可以存在于单引号或者双引号中：

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

解释器输出字符串操作结果，如同和它们输入的方式一样。在引号里面的引号需要通过反斜杠进行转义来显示正确的数值。如果字符串中包含单引号而没有双引号，则字符串应该被双引号包围。否则用单引号包围。Print()方法可以对输入的字符输出更可读的结果。字符创常量可以用不同的方法分在好多行。 可以用反斜杠作为行最后一个字符，用来实现与下一行的逻辑连接。例如：

```
hello = "This is a rather long string containing\n\
several lines of text just as you would do in C.\n\
Note that whitespace at the beginning of the line is\
significant."
print(hello)
```

注意新行仍然需要被嵌入在字符串中，并且新行尾部需要一个反斜杠。例如打印结果如下：

```
This is a rather long string containing
several lines of text just as you would do in C.
Note that whitespace at the beginning of the line is significant.
```

或则，可用一对三个引号或者“‘把字符串包围， 当运行三个字符串的时候，行尾不是没有转义，而是包含着字符中了。 因此，如下利用一个反斜杠来避免无用的初始化空行。

```
print("""\
Usage: thingy [OPTIONS]
-h Display this usage message
-H hostname Hostname to connect to
""")
```

打印结果如下：

```
Usage: thingy [OPTIONS]
-h Display this usage message
-H hostname Hostname to connect to
```

如果我们想吧字符串当做输入字符，\n 字符不被转为新行，而是作为代码中的结束符。都当做数据包含在字符串中。例如：

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

打印结果：

```
This is a rather long string containing\n\
several lines of text much as you would do in C.
```

字符串还可以用+ 操作符进行相加和* 操作符进行重复。例如：

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

两个彼此相邻的字符串可以自动相连接。上面第一行可以被写成"Help""A": 这种方法主要用作两个字符串变量，而不是任意字符串表达式。

```
>>> 'str' 'ing' # <- This is ok
'string'
>>> 'str'.strip() + 'ing' # <- This is ok
```

```
'string'
>>> 'str'.strip() 'ing' # <- This is invalid
File "<stdin>", line 1, in ?
'str'.strip() 'ing'
^
SyntaxError: invalid syntax
```

像 C 一样，字符串可以被索引。字符串的第一个字符的下标是 0，这不需要任何的分割符。 一个字符就简单说就是长度为 1 的字符串。犹如在图表编程语言中，子字符串可以被切片符号表示。两个字符被冒号分割。 例如：

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

切片索引有自己的默认值，前面确实表示默认为0，后面确实表示默认为字符串的长度。

```
>>> word[:2] # The first two characters
'He'
>>> word[2:] # Everything except the first two characters
'lpA'
```

不像 C 语言的字符串，python 字符串不可以改变，如果在字符串中给特定的索引设值会导致错误：

```
>>> word[0] = 'x'
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: 'str' object does not support item assignment
>>> word[:1] = 'Splat'
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: 'str' object does not support slice assignment
```

但是通过结合字符串内容可以创建一个新的字符串是容易而有效的。

```
>>> 'x' + word[1:]
'xelpA'
>>> 'Splat' + word[4]
'SplatA'
```

一个有用的不变切分表达式就等于字符串。

```
>>> word[:2] + word[2:]
'HelpA'
>>> word[:3] + word[3:]
'HelpA'
```

不正确的切片索引可以被智能的处理。如果索引大于字符串长度，则用字符串长度代替。如果上索引值小于下索引则会返回一个空字符串。

```
>>> word[1:100]
'elpA'
>>> word[10:]
''
>>> word[2:1]
''
```

索引值可以为负数，表示从右边开始计数。例如

```
>>> word[-1] # The last character
'A'
>>> word[-2] # The last-but-one character
'p'
>>> word[-2:] # The last two characters
'pA'
>>> word[:-2] # Everything except the last two characters
'Hel'
```

但是要注意： -0 和 0 的作用是相同的。因此不会从右边开始计数。

```
>>> word[-0] # (since -0 equals 0)
'H'
```

超出负数的索引会被截断，但不要尝试用超出负数的索引做单个元素的索引：

```
>>> word[-100:]
'HelpA'
>>> word[-10] # error
Traceback (most recent call last):
File "<stdin>", line 1, in ?
IndexError: string index out of range
```

记住切片如何工作的一种方法就是把索引认为是两个字符之间的节点。这个字符串的左边下限为 0， 右边上限为 n.

```
+---+---+---+---+---+
| H | e | l | p | A |
+---+---+---+---+---+
0   1 2 3 4 5
-5  -4 -3 -2 -1
```

对于没有负数的索引，切片的长度与索引的长度不相同。如果两个都有界限，例如，长度就等于两者之差。

word[1:3] 的长度就是 2.

内置函数的方法 len()就会返回字符串的长度：

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

关于编码：

从 3.0 开始 python 支持所有的编码格式。

如果你想在字符串中包含特殊的字符，你可以利用 python Unicode-escape 编码进行实现。下面例子就是很好展示：

```
>>> 'Hello\u0020World !'
'Hello World !'
```

转义字符串\u0020 表示在给定的位置插入空格（空格的编码就是 0x0020）

除了这些标准的字符编码以外，python 还提供了一整套其他形式的编码来在已知的编码基础上创建特定编码字符串。

为了把一个字符串转化为特殊编码的字符，字符串对象提供了一种方法 encode()方法。编码的小写字母是引用参数：

```
>>> "Äpfel".encode('utf-8')
b'\xc3\x84pfel'
```

1.1.3 列表

Python 可以支持一些符合数据类型，常常和只一起分类。最典型的就是 list, 它可以写成在方括号内一组用逗号分开的数值。List 的数据项不一定都是相同的类型。

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
    ['spam', 'eggs', 100, 1234]
```

像字符串索引一样，列表索引从 0 开始，也可以切分和连接。

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']
```

所以的切操作符都返回一个包含要求元素的新列表，这意外着以下操作将返回列表 a 的一份浅

拷贝。

```
>>> a[:]
['spam', 'eggs', 100, 1234]
```

不像不可变的字符串，列表可以改变列表中的元素。

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

对片段设置也是允许的，这甚至能够改变 list 的大小和全部清除。

```
>>> # Replace some items:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Remove some:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Insert some:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> # Insert (a copy of) itself at the beginning
>>> a[:0] = a
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
>>> # Clear the list: replace all items with an empty list
>>> a[:] = []
>>> a
```

[]

内置方法 len()也支持列表

```
>>> a = ['a', 'b', 'c', 'd']
>>> len(a)
4
```

列表也可以充当另一个列表的元素

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
```

可以在列表结尾操作。

```
>>> p[1].append('xtra')
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

注意在上面例题中，p[1]和q常常都是指同样的对象，接下来我们学习对象语法。

Note that in the last example, p[1] and q really refer to the same object! We'll come back to object semantics later.

1.2 初步走进编程之门

备注： 在此换了 pdf 阅读器，以致从 pdf 原文件中的 python 关键字无法标色，故下文 python 源码统一用蓝色。

当然，我们可以运用 python 实现更加复杂的任务而不仅仅是两个数的相加，例如，我们一

个斐波拉契数列的子数列：

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
... print(b)
... a, b = b, a+b
...
1
1
2
3
5
8
```

这个例子介绍了一些新的特性。

首先，第一行包含一个双重赋值，变量 a 和 b 同时获得新的值 0 和 1， 在最后一行这种方法再次被使用，在此说明在右边的表达式在任何表达式赋值之前进行。右边的表达式依次会从左到右赋值。

其次，只要条件（b< 10）满足，则 while 循环语句就会执行下去。在 python 语句和 C 一样，任何非零整数的值是真的，零为假的。条件也可以是一个字符串或者一个列表值事实上都是序列。 非零长度的字符串是真的，空字符串是假的。 可以做个简单的测试，标准的字符比较操作符和 C 中的一样，< > == <= >= !=

接着， 循环的主体是要缩进的，随进符是分类语句的 python 方法。在相互交互提醒符中，必须输入一个 tab 或者空格为每一岁缩进行。事实上，你可以再文本编辑器上为 python 准备更加复杂的输入。所以常规的文本编辑器都有自己缩进属性，当符合语句交互输入时，必须由空行来标记结束。注意包含基本代码块的代码行必须缩进同样的字符长度。

最后，print 输出表达式被赋予的值，这与你想要写的表达式（上节在计算器例子中）不相同，所使用方式与处理重复表达式、浮点数量和字符串的一样。字符串打印时没有引号，会在字符串之间插入空格，因此你能更好的格式化这些字符串。如下：

```
>>> i = 256
```

```
*
256
>>> print(’The value of i is’, i)
The value of i is 65536
```

在输出之后可以用关键字结束符来避免新行，或者用不同字符串结束输出。

```
>>> a, b = 0, 1
>>> while b < 1000:
... print(b, end=’,’)
... a, b = b, a+b
...
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

第二章 更多控制流程语句

除了前面介绍的 while 语句外，python 也使用在其他语言中所熟知的一些流程控制语句。

2.1 if 语句

可能最被人周知控制语句就是 if 语句了，例如：

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
... x = 0
... print(’Negative changed to zero’)
... elif x == 0:
... print(’Zero’)
... elif x == 1:
... print(’Single’)
... else:
... print(’More’)
```

```
...
```

More

Elif 可不出现也可多次出现和 else 部分也是可以选择的。关键 elif 是 else if 的缩写形式，对于避免冗长的定义有很大帮助。If... Elif... Elif 语句在其他变成语言中可用 switch 或 case 语句来替代。

2.2 for 语句

在 python 中的 for 语句可能有以前你在 C 或者 pascal 中所用的有点小差别。她既不像 Pascal 语言中常常遍历数组的算法过程，也不像在 C 语言中给用户自己定义遍历步骤或模糊条件的能力，python 中的 for 语句可以按照元素在序列（列表或者字符串）中出现的顺序逐步地遍历它们。例如：

```
>>> # Measure some strings:
... a = [’cat’, ’window’, ’defenestrate’]
>>> for x in a:
... print(x, len(x))
...
cat 3
window 6
defenestrate 12
```

在循环语句中修改正在遍历的序列是不安全的（这种情况仅仅发生在可变类型中，如列表）。如果你需要修改你正在遍历的列表（例如，想复制选中项）你必须首先复制 list。然后利用分片符号很方便实现它。

```
>>> for x in a[:]: # make a slice copy of the entire list
... if len(x) > 6: a.insert(0, x)
...
>>> a
[’defenestrate’, ’cat’, ’window’, ’defenestrate’]
```

2.3 range()方法

如果你想迭代一个数字序列，内置方法 range（）可以很方便实现它。它可以生成连续的数字：

```
>>> for i in range(5):
... print(i)
...
0
1
2
3
4
```

方法中给出的结点不是产生序列中的一部分。Range(10)就是产生为长度为 10 序列中每个元素产生 10 个合法的索引值。该方法也可以让范围从其他数字开始，或者指定不同的增长数（甚至是负数，有时称这为阶梯）。

```
range(5, 10)
5 through 9
range(0, 10, 3)
0, 3, 6, 9
range(-10, -100, -30)
-10, -40, -70
```

为了迭代序列的目录，你可以结合 range()和 len（）结合起来，例如：

```
>>> a = [’Mary’, ’had’, ’a’, ’little’, ’lamb’]
>>> for i in range(len(a)):
... print(i, a[i])
...
0 Mary
1 had
2 a
3 little
```

```
4 lamb
```

但是，在许多情况下，运用 enumerate（）方法会更加方法，具体参照 Looping Techniques.

如果你想打印一个范围，奇怪的现象将会出现。

```
>>> print(range(10))
range(0, 10)
```

在很多情况下利用 range()返回对象表现像列表，但实际上它不是列表。尽管当你迭代期望序列时，它能够依次地返回元素，但是它没有真正生成列表，因此节省空间。

我们可以把对象成为是可迭代的，就是指它很适合作为预期得到有序元素事情的方法和结构的对象。 我们已经知道 for 语句就是这种迭代器。方法 list()也算一个，它可以产生有序列表。

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

稍后我们会看到更多能返回迭代对象和把迭代对象当做内容的方法。

2.4 break 和 continue 语句和在循环中的 else 子句

与在 C 语言中一样，Break 语句将会跳出离它最近的 for 或者 while 循环。

Continue 语句也是从 C 中借鉴而来，继续开始循环的下一步。

循环语句中有时含有 else 的子句。当通过遍历结束导致循环终止（for）或者当循环条件变为 false（如 while）,它将会执行。但是当循环被 break 语句所终止，它不会被执行。下面通过查询质数这个例子来学习：

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

是的，这是正确代码。请仔细看，else 语句是属于 for 循环的，而不是 if 语句。与 else 用在 if 语句中相比，else 用于循环中与用在 try 语句有更多相同之处。在 try 语句中当没有异常发生 else 语句将会执行，循环中当没有 break 发生 else 语句就会执行。对于更多的关于 try 语句和异常的信息可以查看处理异常：

2.5 Pass 语句

Pass 语句一般做些无关紧要的事情，当按照句法语句被要求时但是系统不需要任何操作，此时可以用 pass 语句。例如：

```
>>> while True:
... pass # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

还常常用在创建最小类中。

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

能用到 pass 的另一个地方就是当你运行新代码时它作为方法或者条件主体的拥有者，允许你在一个更加抽象的层次去不停地思考。Pass 将会被静静忽略：

```
>>> def initlog(*args):
... pass # Remember to implement this!
...
```

2.6 定义方法

我们可以创建一个实现任意范围内的斐波那契数列的输出功能方法。

```
>>> def fib(n): # write Fibonacci series up to n  写出在 n 以内的斐波那契数列
... """Print a Fibonacci series up to n."""  输出在 n 以内的斐波那契数列
... a, b = 0, 1
... while a < n:
... print(a, end=’ ’)
... a, b = b, a+b
... print()
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

关键字 def 是方法定义的标志。接下来紧跟方法名和被圆括号所包围的参数列表。方法的主体语句将在下一行开始并且必须缩进。

方法主体的首句可选择性地是一句字符，用来说明方法的主要功能。（更多的文档注释可以参照文档语句）。运用 docString 可以自动地生成网页或者打印文档。或者让用户更有效地浏览代码。在你写的代码中包含文档语句是个很好的行为，因此要养成这个习惯。

方法的实施标志着将会有一张新的符号表用来保存该方法的本地变量。更准确地说，在方法中所有的变量赋值都会保存到新的本地符号表中。然而，在变量符号表中首先看到的是引用。接着是围绕方法的本地符号表，接着是全局符号表，最后是内置名称的表。因此，尽管全局变量在一个方法中被引用，但它们不能被直接赋值（除非在全局语句中被命名）。

方法调用的真正参数时候，参数将会在被调用方法的本地符号表中被标记。因此，参数可以用调用值来传递。（这里的值其实是对象的引用，而不是对象值本身）。 当一个方法调用另一个方法，系统将会为新调用方法产生新的本地符号表。

在目前的符号表中方法定义采用方法名字。方法名字的值有个被解释器确认为用户自定义方法的类型。其他可以用在方法名称的值可以重新赋值给上面提到的方法名称。这可以认为是一个普遍的重命名机制。

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

你从其他语言转过来学习 python，你可能反对 fib 不是一个方法而是一个过程。因为它没有返回值。事实上，任何没有 return 语句的方法都有返回值，即使是相当无聊的方法。 这个返回值就是 null(这是内置名称)。如果返回值只有一个可写值，那么输出为空是被解释器所禁止的。如果你真想用 print()方法你就会看到：

```
>>> fib(0)
>>> print(fib(0))
None
```

写一个有返回斐波那契数列值得方法是很简单的，代替打印输出。

```
>>> def fib2(n): # return Fibonacci series up to n
... """Return a list containing the Fibonacci series up to n."""
... result = []
... a, b = 0, 1
... while a < n:
... result.append(a) # see below
... a, b = b, a+b
```

```
... return result
...
>>> f100 = fib2(100) # call it
>>> f100 # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

通常一样，这个例子涉及一些新的 python 特性。

- Return 语句表示从方法中返回一个值。没有 return 语句表示放回一个 None.没有执行到方法结尾也返回 None.
- Result.append(a)语句是调用列表对象 result 的一个方法。一个方法属于该对象就可以用对象.方法名称进行调用。不同类型可以定义不同的方法，不同类型的方法可以有相同的名称不会引起冲突。 在例子中显示的方法 append()被定义在 list 对象中。该方法意思是在 list 末尾增加一个新的元素。 在这个例子它等同于 result = result + [a], 但是比后者更有效率。

2.7 更多关于方法定义

Python 允许用可变数目的参数定义方法。以下有可以相互结合的三种方法。

2.7.1 默认参数值

最有用的形式就是给一个或多个变量制定默认值。这种方法可以创建一个允许调用时比定义时需要更少的参数。例如：

```
def ask_ok(prompt, retries=4, complaint=’Yes or no, please!’):
while True:
ok = input(prompt)
if ok in (’y’, ’ye’, ’yes’):
return True
if ok in (’n’, ’no’, ’nop’, ’nope’):
return False
retries = retries - 1
```

```
if retries < 0:
raise IOError(’refusenik user’)
print(complaint)
```

这个方法可以用以下几种方法调用：

- 给个唯一常量： ask_ok(’Do you really want to quit?’)
- 给一个变量： ask_ok(’OK to overwrite the file?’, 2)
- 设置所有的变量：

  ask_ok(’OK to overwrite the file?’, 2, ’Come on, only yes or no!’)

这个实例也介绍关键 in 的用法。其功能在于测试输入字符串是否还有特定的值。默认值也可以在方法定义时候就被限制范围。例如：

```
i = 5
def f(arg=i):
print(arg)
i = 6
f()
```

将会输出 5

重要提醒： 默认值仅被设置一次，这与以前默认值为可变对象（如列表、字典和多数类实例时）有很大的区别。例如， 接下来的方法累计被传入的参数变量到下次调用：

```
def f(a, L=[]):
L.append(a)
return L
print(f(1))
print(f(2))
print(f(3))
```

将会输出

```
[1]
[1, 2]
[1, 2, 3]
```

如果你不想默认值在两个调用时分享。你可以写如下方法代替上面。

```
def f(a, L=None):
```

```
if L is None:
L = []
L.append(a)
return L
```

2.7.2 关键字参数

可以通过形式<kwarg = value>关键字参数调用方法。例如， 接下来方法：

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

接受 1 个要求的参数（voltage）和三个可选择参数（state, action 和 type）。这个方法用以下任何一种方法调用。

```
parrot(1000) # 1 positional argument
parrot(voltage=1000) # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM') # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000) # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword
```

但以下方法调用时不合法：

```
parrot() # required argument missing 必须的参数缺失
parrot(voltage=5.0, 'dead') # non-keyword argument after a keyword argument 关键参数后无关键参数
parrot(110, voltage=220) # duplicate value for the same argument 同一参数重复赋值
parrot(actor='John Cleese') # unknown keyword argument 无名的关键参数
```

在方法调用中，关键字参数必须遵循位置参数。 所有的关键参数必须符合方法接受的参数其中之一。（例如， actor 不是 parrot 方法的合法参数），但是它们的次序不重要。这包含非选择的参数（例如 parrot(voltage = 1000)是合法的）。没有参数可以多次接受一个值。以下例

子由于这种限制而运行有问题。

```
>>> def function(a):
... pass
...
>>> function(0, a=0)
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument ’a’
```

当最后一个形参是**name 时，它可以接受包含除了形式参数之外的所有关键字的字典。这可以和形如“*name”的形式参数结合使用。*name 将会在下节讲到，它可以接受包含位置参数的数组。*name 必须在**name 之前出现。例如，我们可以如下方法：

```
def cheeseshop(kind,*arguments,**keywords):
print("-- Do you have any", kind, "?")
print("-- I’m sorry, we’re all out of", kind)
for arg in arguments:
print(arg)
print("-"*40)
keys = sorted(keywords.keys())
for kw in keys:
print(kw, ":", keywords[kw])
```

它可以按照如下方法调用

```
cheeseshop("Limburger", "It’s very runny, sir.",
"It’s really very, VERY runny, sir.",
shopkeeper="Michael Palin",
client="John Cleese",
sketch="Cheese Shop Sketch")
```

结果如下：

```
-- Do you have any Limburger ?
-- I’m sorry, we’re all out of Limburger
```

It’s very runny, sir.

It’s really very, VERY runny, sir.

----------client : John Cleese

shopkeeper : Michael Palin

sketch : Cheese Shop Sketch

2.7.3 可变参数列表

最后，相对不太频繁常用的方式是调用含有可变参数的方法。这些参数常被包含在元组中。在可变的参数中，零或更多正式参数都可能出现。

def write_multiple_items(file, separator,*args):

file.write(separator.join(args))

正常来说，这些可变参数常常放在正式参数列表的后面，因为它们会包揽所有传递给该方法的剩余输入参数。任何出现在*args 参数后低的正式参数会被认为是关键字参数，意味着它们只能当关键字使用而不是位置参数。

>>> def concat(*args, sep="/"):

... return sep.join(args)

...

>>> concat("earth", "mars", "venus")

’earth/mars/venus’

>>> concat("earth", "mars", "venus", sep=".")

’earth.mars.venus’

2.7.4 拆分参数列表

当参数已经存在列表或者元组中，但是需要分拆以供要求分离位置参数调用的方法，这是相反的情景出现了。例如：在内置方法 range()中期望获得独自的开始和停止参数。如果单独分开它们无法使用，就需要写一个方法用*操作符来调用实现分拆列表或者元组中的参数。

>>> list(range(3, 6)) # normal call with separate arguments

[3, 4, 5]

```
>>> args = [3, 6]
>>> list(range(*args)) # call with arguments unpacked from a list
[3, 4, 5]
```

2.7.5 同样的使用形式，字典可以用** 操作符实现关键字参数。

```
>>> def parrot(voltage, state=’a stiff’, action=’voom’):
... print("-- This parrot wouldn’t", action, end=’ ’)
... print("if you put", voltage, "volts through it.", end=’ ’)
... print("E’s", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin’ demised", "action": "VOOM"}
>>> parrot(**d)
```

2.7.5 形式

由于普遍需要，在函数程序语言中会发现一个共同的特性，像 Python 中添加 Lisp。 像匿名关键字、小型匿名方法也被创建。下面这个方法返回两个参数的和。Lamdba a,b : a + b . 哪儿需要方法时，哪儿就可以使用形式。它们是根据句法限制在一个简单的表达式。在语义上，它们仅是针对正式方法定义的语法补充。像内嵌方法定义中，形式也可通过包含范围中引用变量。

```
>>> def make_incrementor(n):
... return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

2.7.6 文档字符串

在 python 中关于文档字符串的内容和格式有个规范。

第一行常常是一句对对象作用的剪短而又准确地总结。尽管简洁，但不能明确声明对象的名称和类型，因为通过其他方法(除了名字恰好是一个描述方法功能的动词)它们依然是可用的。

```
>>> def my_function():
... """Do nothing, but document it.
...
... No, really, it doesn’t do anything.
... """
... pass
...
>>> print(my_function.__doc__)
Do nothing, but document it.
No, really, it doesn’t do anything.
```

2.8 编码风格

现在你准备写出更长而复杂的 python 代码了， 也是该告诉你关于 python 的编码风格时候了。很多语言可以用不同的风格进行编码。有些风格比其他风格编写的代码更有可读性。 因此，让别人能有轻易地读懂你的代码是一直是个好想法，并且养成这种良好的编码风格将更大地帮助你。

对于 python，PEP 8 作为许多项目应该遵守的编码指导书而做的。 它提出了一种可读而悦目的编码风格。 每位 python 开发者应该读它。这里抽出一个重要的事项与你分享 ：

- 用四个空格代替 tab 键
- 每行不要超过 79 个字符。
- 用空行分离方法和类，大块代码中的方法。
- 必要的时候为每行添加注释。
- 用文档字符串。
- 在操作符两边用空格
- 用统一的风格命名自定义的方法和类
- 如果你的代码打算用在国际环境中，请不要用想象的字符编码。Python 默认的是

utf-8 ,在任何情况下可以用 Ascii .

- 同样的，即使有很少机会让说不同语言的人们读代码或者维护代码，但在定义中不要用非 ASCII 编码字符。

第三章 数据结构

这章将详细学习你以前已经知道的知识，同时也会添加一些新知识。

3.1 列表

列表数据类型有很多方法，以下是列表对象的所有方法：

list.append(x)

添加单个元素到列表末尾，等同于 a[len(a)] = [x]

list.extend(L)

通过添加指定列表中所有的元素来扩展列表，等同于 a[len(a):] = L.

list.insert(i, x)

把单个元素插入到制定的元素。第一个参数是在列表中的索引。因此 a.insert(0, x)意思是吧元素 X 插入到列表最前面，a.insert(len(a), x) 等同于 a.append(x).

list.remove(x)

从列表中删除第一个值为 x 的元素，如果没这个元素将会出错。

list.pop([i])

删除指定位置的元素，并且返回该元素，如果没有指定的索引，a.pop()就会删除列表中最后一个元素，括号中的参数是可以选择的。

list.index(x)

返回列表中第一个满足值为 X 元素的索引，如果没有该元素就会出错。

list.count(x)

计算在列表中值为 x 的元素出现的次数。

list.sort()

对列表中的元素进行排序。

list.reverse()

翻转列表中的元素。

以下是调用 list 中各种方法的例子：

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
29
Python Tutorial, Release 3.2.3
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

3.1.1 把列表当做栈来用

列表中的方法很容易实现把列表当做栈来用，在栈中元素师“后进先出”。给栈顶添加单个元素可以用方法 append(). 从栈顶检索一个元素用不带参数方法 pop()，例如：

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
```

```
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

3.1.2 把列表当做队列来用

也可以把列表当做队列来用，在队列中元素是“先进先出”。但是列表当做队列用时效率不是很高。尽管在列表末添加和删除元素是很快的，但是从列表开头插入和删除数据时很慢的。（因为所有的数据都要逐个交换）

为了实现队列，可以利用 collections.deque 方法，它可以从两端进行快速地添加和删除。例如：

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry") # Terry arrives
>>> queue.append("Graham") # Graham arrives
>>> queue.popleft() # The first to arrive now leaves   第一个到达的现在离开
’Eric’
>>> queue.popleft() # The second to arrive now leaves  第二个到达的现在离开
’John’
>>> queue # Remaining queue in order of arrival  按到达次序排列的剩余队列
deque([’Michael’, ’Terry’, ’Graham’])
```

3.1.3 递推式构造列表

递推式构造列表提供了一些简洁的方法来创建列表。通用程序可以创建新列表，在每个元素是一些对其他序列或迭代上每个元素操作后的结果，或者创建一个元素中满足特定条件的子序列。

例如： 假设我们想创建一个平方的列表，就像：

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

我们可以用如下语句获得同样的结果

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

这也等同于 squares = map(lambda x : x ** 2, range(10)), 但是它更加简洁和可读。

一个列表的综合应用常由包含 for 语句，接着零个或多个 for 或者 if 语句的方括号组成。结果将是一个满足表达式中 for 和 if 语句的新列表。 例如： 如下列表时结合两个不相等的列表。

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

等同于

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

注意： 在两个表达式中的 for 和 if 的次数应该相等。

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled  创建一个包含双倍数值的列表
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers  过滤列表中的负数
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements  为所有元素调用方法
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element  为每个元素调用方法
>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit'] 去掉每个元素中首尾空格
>>> # create a list of 2-tuples like (number, square)  创建一个包含二位数组的列表
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised  数据必须用括号括起来
>>> [x, x**2 for x in range(6)]
5.1. More on Lists 31
Python Tutorial, Release 3.2.3
File "<stdin>", line 1, in ?
[x, x**2 for x in range(6)]
^
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'  用两个 for 遍历出展开所有的列表元素
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

列表复合表达式可以获得更复杂的表达式和内嵌方法。

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

3.1.4 嵌套列表推导式

在列表推导式中初始表达式可以是任意表达式。包含其他的列表推导式。

思考一下用长度为 4 的 3 个列表实现的 3* 4 维的矩阵。

```
>>> matrix = [
... [1, 2, 3, 4],
... [5, 6, 7, 8],
... [9, 10, 11, 12],
... ]
```

用如下列表表达式就可以实现转换行列。

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

正如上节我们看到的，潜逃列表可以用 for 语句来实现。 因此上个例子也等同于：

```
>>> transposed = []
>>> for i in range(4):
... transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

也等同于：

```
>>> transposed = []
>>> for i in range(4):
... transposed_row = []
... for row in matrix:
... transposed_row.append(row[i])
... transposed.append(transposed_row)
```

```
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

在现实生活中，你可能更喜欢用内置方法来实现复杂的流程控制。Zip()方法就是对这个例子就有很好的应用。

```
>>> zip(*matrix)
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

3.2 删除语句

Del 删除语句能根据给定的索引而不是值从列表中快速的删除元素。它与能产生返回值的 pop()方法不同。Del 语句也可以从列表中删除数据段或者清空整个列表。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 语句也可以删除整个变量。

```
>>> del a
```

以后对 a 引用就会出错。（至少直到它被重新赋值）。

3.3 元组和序列

我们看到列表和字符串有许多相同的属性，例如索引和切片操作符。以下是两个关于序列数据类型的例子。由于 python 是一种不断演变的语言，其他的序列数据类型可以被添加。现在已经有一种标准的序列数据类型：元组

一个元组是由一些被逗号分隔的熟知组成的。例如：

```
>>> t = 12345, 54321, ’hello!’
>>> t[0]
12345
>>> t
(12345, 54321, ’hello!’)
>>> # 元组也可以嵌套
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, ’hello!’), (1, 2, 3, 4, 5))
>>> # 元组是不可变的
... t[0] = 88888
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: ’tuple’ object does not support item assignment 元组不支持元素赋值，
>>> # but they can contain mutable objects: 但是可包含可变的对象。
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

正如你看到的，当输出时元组常常被括号括起来，因此嵌套元组可以被正确解释。尽管常在输入时候用括号把它们包围，但是包或不包含都可以输入。 不可以对元组中的单个对象赋值，但是创建包含可变的对象的元组，比如列表。

尽管元组与列表有点相似，但是常常在不同情况和因不同的目的而使用。元组是不可变的，常常包含一个可以通过取出和索引访问元素的异构序列。列表是可变的，并且这些元素都是平等的，只能通过遍历逐次访问。

一个特殊的问题就是仅包含一个或空的元组的构建。为了符合这些规则，这语法有点特别之处。每个空元组可以被一对空括号所创建。含一个值的元组可以用一个值加上”，”进行创建。（用括号包围单个值没有意义）。尽管看起来很丑陋，但是很有效的。例如：

```
>>> empty = ()
>>> singleton = ’hello’, # <-- note trailing comma
>>> len(empty)
```

```
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

语句 t = 12345, 54321, 'hello!' 是一个包含不同值的元组例子:值是 12345, 54321 and 'hello!'都放在同一个元组中。

翻转元组的操作可以如下：

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

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其实只是元组封装和序列拆封的一个结合！

3.4 Set 集合

Python 还包含一个集合的数据类型。一个 Set 是一个不包含重复元素的无序集合。基本应用是成员资格测试和消除重复元素。 Set 对象也提供一些算术操作符，比如连接,交集，差值和堆成差。

花括号或 set()方法都可以用来创建集合。 注意如果你想要创建一个空集合，必须要用 set() 而不能用{}。 后者创建了一个空字典，空字典我们将在下一节讨论。

对于集合的使用在这里做一简单示范：

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket) # show that duplicates have been removed #显示除去重复元素
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket # fast membership testing #快速成员检测
True
>>> 'crabgrass' in basket
False
>>> # 展示通过 set 操作从两个词中获得唯一的字母

...
```

```
>>> a = set(’ abracadabra’ )
>>> b = set(’ alacazam’ )
>>> a # unique letters in a 在 a 中唯一字母
{’ a’ , ’ r’ , ’ b’ , ’ c’ , ’ d’ }
>>> a - b # letters in a but not in b 在 a 中但不在 b 中
{’ r’ , ’ d’ , ’ b’ }
>>> a | b # letters in either a or b 在 a 或 b 中
{’ a’ , ’ c’ , ’ r’ , ’ d’ , ’ b’ , ’ m’ , ’ z’ , ’ l’ }
>>> a & b # letters in both a and b 在 a 且在 b 中
{’ a’ , ’ c’ }
>>> a ^ b # letters in a or b but not both 在 a 或在 b 中，但不同时在两者中
{’ r’ , ’ d’ , ’ b’ , ’ m’ , ’ z’ , ’ l’ }
```

Like for lists, there is a set comprehension syntax:

像列表一样，集合也有“理解语法”。

```
>>> a = {x for x in ’ abracadabra’ if x not in ’ abc’ }
>>> a
{’ r’ , ’ d’ }
```

3.5 字典

定义在 python 中另一种数据类型是字典。其他语言中字典可能被定义为“组合记忆”或“组合数组”。 不像序列能用数字可以索引，在字典中是任何一种不变类型的关键字 keys 来查询。字符串和数字是常见的关键字。 如果元组中仅仅包含字符串，数组或者重数，那么元组也可以充当关键字。但是如果元组中直接或者间接包含可变的对象，那么元组就不能做关键字。

你不能把列表用作关键字，因为列表在索引赋值，切片赋值以及方法 append（）和 extend() 运用时可以改变本身值。

把字典认为是一对无序的 key：value 的对，并且关键字在一个字典中是唯一的是对字典非常正确的认识。可以用 del 方法来删除一对键值对。当你用一个已经存在的键去保存值时，

与其相关联的值将会丢弃。用没有存在的关键字去获得值时会出错。

用字典中的方法 list（d.keys（））就可以获得包含字典中所有的关键字的列表，列表中的关键字是无序的，如果你想要对关键字进行排序，可以用方法 sorted（d.keys()）.可以用 in 关键字来检查单个关键字是否在字典中。

如下是运用字典的一个简单例子：

```
>>> 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}
>>> list(tel.keys())
[’irv’, ’guido’, ’jack’]
>>> sorted(tel.keys())
[’guido’, ’irv’, ’jack’]
>>> ’guido’ in tel
True
>>> ’jack’ not in tel
False
```

在字典中可以用 dict（）构造方法直接直接从包含 key-vlaue 的序列中定义一个字典。

```
>>> dict([(’sape’, 4139), (’guido’, 4127), (’jack’, 4098)])
{’sape’: 4139, ’jack’: 4098, ’guido’: 4127}
```

此外，字典利用复合表达式从随机的 key 和 value 的表达式中来创建字典。

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

当关键字是简单的字符串时，可以用关键字参数直接指定对象，这样有时会更加的容易。

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

3.6 遍历技巧

当通过字典遍历数据时，用 items()方法就可以同时把关键字和相对应的值从字典中取出。

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

当用序列遍历数据时，用 enumerate()可以同时把位置索引和对应的值得到。

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

想要同时遍历两个或多个序列时，可以用方法 zip()把属性整合起来。

```
>>> 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.
```

想要倒叙遍历序列，首先正序指定遍历序列，然后调用方法 reversed().

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

想要有序的遍历列表，用方法 sorted()可以返回一个新的有序列表而不改变原先列表。

```
>>> basket = [’ apple’ , ’ orange’ , ’ apple’ , ’ pear’ , ’ orange’ , ’ banana’ ]
>>> for f in sorted(set(basket)):
... print(f)
...
apple
banana
orange
Pear
```

3.7 深入条件控制

在 while 和 if 语句中用到的条件可以包含任何操作符而不仅仅是比较运算符。

比较运算操作符 in 和 not in 主要检查值是否在列表中出现(或不出现)。操作符 is 和 is not 用来比较两个对象是不是真正相同的对象。这仅仅用于可变对象如列表。所有的比较运算法拥有同样的优先级，但都比数字操作符的优先级低。

比较运算法可以连接使用，例如： a < b == c 用来测试 a 是否< b 并且 b == c。

可以用布尔操作符 and 和 or 来组合比较操作符。并且比较的结果（或者任何其他逻辑表达式）可以使用 not 操作符取反。 这些操作符（指 and, or 和 not）的优先级低于比较操作符。在它们之中，not 拥有最高的优先级，or 拥有最低的优先级。 因此 A and not B or C 等同于（A and (not B)）or C , 和其他一样，可以用括号表达想要的组合。

布尔操作符 and 和 or 是所谓短路的操作符。他们参数从做左到右依次判定，只要结果确定

就停止。例如如果 A 和 C 是真的但是 B 是假的，A and B and C 不能判定表达式 C，当短路操作符返回值作为普通值而不是布尔值使用时，则返回值是最后一个被判定的表达式。

可以允许把比较运算符或者其他布尔表达式的结果赋值给一个变量。例如：

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

注意在 python 中不像 C 语言，赋值操作不能出现在表达式中。C 程序员可能会对此抱怨，但是它避免了一个在 C 程序中经常遇到的典型的问题。想要在一个表达式中使用== 操作时确输入了 =。

第四章 模块

如果你退出 python 的解释器重新进入，则有关你已定义的方法和变量都会消失。因此，如果你想编写比较长的程序，你最好用文本编辑器来为解释器准备输入和用文件作为输入来运行程序。这就是众所周知的脚本。如果你的程序更强大而负责，你可能为了方便维护想把它分为几个文件。可能你想用你已写在文件中的便利方法而不需要把它拷贝到每个程序中。

为了支持这个功能，Python 提供一种方法把这些定义放置一个文件中，可以在脚本或者解释器的相互交互实例中运用他们。在 python 中这样的文件叫做 模块。在模块中的定义可以直接导入到其他模块或者 main 模块中。（你在顶层进入执行脚本或计算模式下的变量的集合）

一个模块是包含 python 定义和语句的文件。文件名称是由模块名加上后缀名.py 组成的。在模块中，模块的名字（可以作为一个字符串）是一个作为全局变量_name_的值的变量。例如，用你最喜欢的文本编辑器在当前目录下创建一个名叫 fibo.py 的文件。文件内容如下：

```
def fib(n): # write Fibonacci series up to n
a, b = 0, 1
while b < n:
print(b, end=’ ’)
a, b = b, a+b
print()
def fib2(n): # return Fibonacci series up to n
result = []
a, b = 0, 1
while b < n:
result.append(b)
a, b = b, a+b
return result
```

现在进入 python 解释器然后用如下命令导入这个模块。

```
>>> import fibo
```

这不允许进入当前符号表中直接定义在 fibo 的方法名。这只能进入名为 fibo 的模块中。用模块名字你可以访问方法。

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
’fibo’
39
```

如果你使用该方法，常常把它赋于一个本地名字

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

4.1 深入模块

一个模块可以包含可执行语句和方法定义。这些语句用来初始化模块。当模块第一次被导入到其他地方时候这些语句将会执行。

每个模块都有自己私有的符号表，定义在模块内的所有方法可以把它当做全局符号表来用。因此，模块的作者可以在模块中用全局变量而不同担心与用户全局变量的意外冲突。另一方面，如果你确切地知道自己在做什么，你可以使用引用模块函数的表示法访问模块的全局变量。Modname.itemnae.

模块中可以导入其他模块。习惯上把所有的 import 语句放在一个模块的开始位置，但这不是强制的。导入的模块名称将会在放在正在导入模块的符号表中。例如：

还有一种 import 语句的变体，可以从一个模块中将名字直接导入到当前模块的符号表中。

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这种方式不会把模块名字放在本地符号表中。（例如，fibo 是没有定义的）

还有一种导入所有定义模块的名称的变体

```
>>> from fibo import*
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这种方式会导入除以下划线_开始之外的开头所有名称。在许多情况下，python 开发者不会用这种工具。因为它会在解释器中引入一些未知的名称集合，有可能隐藏一些你已经定义的方法。

注意，一般而言，从模块或者包中 import * 这种做法是不赞成的，因为它常常影响代码的可读性。但是，你可以用它来保存在交互 session 中的输入

注意： 为了效率，在每一个解释会话中每个模块只能允许导入一次。 因此，如果你修改你的模块，你必须重启 python 解释器。Or 仅仅是一个你想交互测试的模块，你可以用方法 Imp.reload(),eg,import imp; imp.reload(moduleName);

4.1.1 像脚本一样执行 python

当用如下方法运行一个 python 脚本，

Python fibo.py <arguments>

在 python 中的代码将会被执行，就像你导入一样。但是方法”_name_”将会被设置为”_mian_”.

这就意外着可以再你的模块末尾将会添加如下方法：

```
if __name__ == "__main__":
import sys
fib(int(sys.argv[1]))
```

你可以在导入的模块中像脚本一样运行 python 代码，因为这些命令解析的代码仅当模块被当做主文件执行时会被运行。

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

如果仅仅是导入，代码不会执行。

```
>>> import fibo
>>>
```

这种方法常用来提供一个便利的用户接口给一个模块，或者为了测试的需要。

4.1.2 模块的搜索路径

当一个名叫 spam 的模块导入时，解释器首先会用这个名称在内置模块中寻找。如果没有发现 ，它会给定的 sys.path 路径下的目录中查找 spam.py 的文件。Sys.path 在如下位置被初始化：

- 包含输入脚本的目录或当前目录
- PYTHONPATH,(目录名称的列表，相当于 shell 变量的 path)
- 安装的默认路径

初始化后 Python 程序就会修改 sys.path.包含正在运行脚本的目录常放在查找路径的开始。在标准库路径的前面。这就意外着在这个路径下的脚本不应该与库文件具有相同的名称。否则当导入一个模块时候 python 将会把脚本当做模块加载，这通常会导致一个错误。

4.1.3 编译 python 文件

作为一个为使用大量标准模块的小程序启动时间加速的重要方式，如果在 spam.py 所在的目录存在一个名为 span.pyc 的文件，这被认为是模块 spamd 的字节码预编译的版本。

创建 spam.pyc 时文件 spam.py 的修改时间版本会被记录在 spam.pyc 文件中。如果这两者不一致，那么.pyc 文件就会忽略。

通常你无需要自行创建 spam.pyc 文件，每次 spam.py 成功编译后，都厚尝试将编译的版本写入 spam.pyc 文件中。如果尝试写入失败后，也不会引发什么错误。不论什么错误。Spam.pyc 文件的内容是平台无关的，所以一个 python 模块目录可以被不同体系架构的机器共享。

4.2 标准接口

Python 自带一些标准模块的库文件。这些库文件介绍在单独的文档（python 库文件介绍）中有所描述。一些模块在解释器中创建，它们提供了对非语言核心的但又为了效率又不得包含到里面部分的操作方法，或者是提供了对操作系统的底层的访问，例如系统调用。这些模块 ge 会根基底层平台进行不同的选择配置，比如：winreg 模块只能在 window 系统上提供， 另一个特殊模块更值得注意，sys，它内置在每个 python 解释器中，变量 sys.ps1 和 sys.ps2 定义了主提示符和此提示符使用的字符串。

```
>>> import sys
>>> sys.ps1
’ >>> ’
>>> sys.ps2
’ ... ’
>>> sys.ps1 = ’ C> ’
C> print(’ Yuck!’ )
Yuck!
C>
```

只有当解释器处于交互模式时候，这两个变量才能被定义。

变量 sys.path 是一系统决定在解释器搜索模块路径的字符串列表。它从默认的环境变量 pythonPATH 或者当 PYTHONPATH 为空时候从内置的默认路径进行初始化。你可以用标准

列表操作进行修改。

```
>>> import sys
>>> sys.path.append(’/ufs/guido/lib/python’)
```

4.3 dir()函数

内置函数 dir 用来寻找查找模块定义的名称。它返回一个排序后的字符串列表。

```
>>> import fibo, sys
>>> dir(fibo)
[’__name__’, ’fib’, ’fib2’]
>>> dir(sys)
[’__displayhook__’, ’__doc__’, ’__excepthook__’, ’__name__’, ’__stderr__’,
’__stdin__’, ’__stdout__’, ’_getframe’, ’api_version’, ’argv’,
’builtin_module_names’, ’byteorder’, ’callstats’, ’copyright’,
’displayhook’, ’exc_info’, ’excepthook’,
’exec_prefix’, ’executable’, ’exit’, ’getdefaultencoding’, ’getdlopenflags’,
’getrecursionlimit’, ’getrefcount’, ’hexversion’, ’maxint’, ’maxunicode’,
’meta_path’, ’modules’, ’path’, ’path_hooks’, ’path_importer_cache’,
’platform’, ’prefix’, ’ps1’, ’ps2’, ’setcheckinterval’, ’setdlopenflags’,
’setprofile’, ’setrecursionlimit’, ’settrace’, ’stderr’, ’stdin’, ’stdout’,
’version’, ’version_info’, ’warnoptions’]
```

没有参数 dir()会遍历当前你已经定义的模块名称。

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
[’__builtins__’, ’__doc__’, ’__file__’, ’__name__’, ’a’, ’fib’, ’fibo’, ’
sys’]
```

注意它会遍历所有类型的名称：那变量 模块 和函数等等

Dir()无法遍历出内置函数和变量的名称， 如果你想要那样的一个列表，在标准模块 buitins 中它们被定义。

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError', 'RuntimeWarning', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '__build_class__', '__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs', 'all', 'any','ascii', 'bin', 'bool', 'bytearray', 'bytes', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr','globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance','issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

4.4 包

包是一种通过“点模块名称”来创建模块命名空间的一种方法。例如，模块 A.B 表示在 A

的包下设计了一个子包 B。就像模块的应用保存不同模块的作者，这些不同的模块需要考虑相互的全局变量名称。但是点模块名称让多个模块包的作者无需担心彼此的模块名称（冲突），就像 Numpy 或 python 图像库。

假设你为了统一操作音频文件和音频数据，你可以定义一个多模块（包）的集合。在里面可能有不同音频文件的格式（通常用它们的扩展名来区别，例如.wav, .aiff, .au）因此你需要创建和维护很多的模块结合来实现不同文件格式之间的转换。也有可能你针对不同的文件格式有不同的操作（例如混音，添加回声，应用均衡器，创建人造立体声等等)。因此，为了进行这些操作你还需要另外编写一个永远也玩不成的模块。这里是你的包的一种可能的结构。

sound/ Top-level package 最顶层文件

__init__.py Initialize the sound package 初始化音频包

formats/ Subpackage for file format conversions 文件格式转换的子包

__init__.py

wavread.py

wavwrite.py

aiffread.py

aiffwrite.py

6.4. Packages 43

Python Tutorial, Release 3.2.3

auread.py

auwrite.py

...

effects/ Subpackage for sound effects 声音效果的子包

__init__.py

echo.py

surround.py

reverse.py

...

filters/ Subpackage for filters 过滤器的子包

__init__.py

equalizer.py

vocoder.py

karaoke.py

...

当导入包后，python 会通过 sys.path 的目录寻找包的子路径。

init.py 文件将被要求让 python 认为目录中包含包。这是为了避免一个含有烂俗名字的目录无意中隐藏了稍后在模块搜索路径中出现的有限模块，比如 string,最简单的情况下，只需要一个_init_.py 文件即可。当然它为包执行初始化代码或者设置_all_变量。

用户每次只能从包中特定的导入模块。例如：

import sound.effects.echo

这种方法加载子模块 sound.effects.echo. 必须用全名来引用。

sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)

导入子模块的另一种方式是：

from sound.effects import echo

这种方式也可以加载模块 echo,并且没有包的前缀也可以使用，因此它可以如下应用：

echo.echofilter(input, output, delay=0.7, atten=4)

另一种直接导入期望的方法和变量方法：

from sound.effects.echo import echofilter

同时这种方法加载了子模块 echo,但是这种方法可以直接调用函数 echofilter（）；

echofilter(input, output, delay=0.7, atten=4)

注意使用 from package impport item 这种方法时， the item 既可是一个子模块或者是在包中定义的其他名称，像函数，类，或者变量。Import 语句首先就会检测是否定义在包中。如果没有，它就会假设是个模块然后去加载。如果没有找到，就会报 ImportError 异常。

相反地，当用表达式 import item.subitem.subsubitem，除了最后一个子项其他都是包。最后一个可以使一个模块或者包，但不允许是在定义在模块中的类或者函数或者变量。

4.4.1 从包中导入*

当用户写 from sound.effects import * 会发生什么？ 理论上，他预期以某种方式友好地对待文件系统，寻找在包下的所有子模块，并且把它们都导入进来。它可能会花费更多的时间，

导入子模块可能不希望出现的负面影响，这种负面影响仅仅在子模块被明确导入时候才会出现的。

对包作者而来，唯一的解决办法就是提供了一个关于包的准确索引。 Import 语句会用如下的转换。如果一个包的_init_.py 代码定义了一个名为_all_的列表，当使用 form package import * 时它被当做应该被导入的模块名称的列表。 当发布包的一个新版本时，应该由 包作者负责这个列表时最新的。如果包作者在包中没有发现导入* 的用法。那么他们可以决定不去做这些。例如：

Sound/effects/_init_.py 文件有如下代码：

```
__all__ = ["echo", "surround", "reverse"]
```

这将意外着 from sound.effects. Import * 将会导入在 sound 包下三个名为显示的子模块。

如果_all_没有定义， 语句 form sound.effects import * 不会从包 sound.effects 导入所有子包到当前的命名空间。它仅仅确保包 sound.effects 已经导入。然后导入包中定义的任何名称。这会引入_init_.py 文件中定义的所有名称（及明确加载的模块）。它同样会引入通过之前导入语句明确加载额的包中的所有模块，思考如下代码：

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import*
```

在这个例子中，echo 和 surround 模块也会导入到当前命名空间下，因为当 from...import 执行时他们也定义在 sound.effects 包中。（这也可以当_all 定义时实现）

注意： 通常从包或者模块中导入* 的习惯是不被推荐的，因为他经常会让代码难以阅读，不过，在交互会话中使用它来减少输入是不会有问题的，并且有些模块被设计只能通过特性方法导入。

记住： from package import specific_submodule 永远那是不会错的。事实上，这是被推荐使用的方法，除非在不同包中导入的模块使用了相同的名字。

4.4.2 包内引用

当包设计成好几个子包时（就像在例子中的 sound 包），你可用绝对导入来指定到子模块或者相应的包。例如，如果模块 sound.filters.vocoder 需要用在 sound.effects 包中 echo 模块，它可用 from sound.effects import echo.

你也可以写用 from module import name 形式导入语句写相关导入。这些导入用.. 来描述涉及到相关导入的当前路径和父包。例如，从 surrond 模块导入：

from . import echo

from .. import formats

from ..filters import equalizer

注意：相对导入是以当前模块名称为基础的，因为主模块的名称总是_main_，所以以 python 程序中打算用作主模块的模块必须使用绝对导入。

4.4.3 跨目录的包

包支持一个特殊熟悉，_path_。这会在那个文件执行之前，初始化为一个包含包中_init_路径下的名称列表。这个变量可以修改用来影响对包中包含的子包和模块的搜索。

尽管这个特性并不常用，但它可以用来扩展一个包中的模块集合。

第五章 输入输出

一个程序的输出有好多种方式，数据既可以用人们读懂的形式打印出来或者写入到文件以便将来使用。这章就讨论这些问题。

5.1 格式化输出

目前我们已经接触了两种输出值方式： 表达式语句和 print()函数。（第三种就是使用 file 对象中的 write（）方法;标准的文件输出可参考 sys.stdout 库文件）

常常你会想控制输出格式化的数据而不是简单用空格分隔的字符。这里有两种方式用来格式化你的输出数据。第一种方式就是由你自己处理所有的字符串，用字符串中的切分或者链接操作你可以创建任何你想要是的字符。标准模块 string 包含一些将字符串填充到指定列宽度的有用操作，随后将会讨论这些。第二种方法就是使用 str.format()方法。

String 模块中包含一个 template 类，它提供另一种方法来将值转化为字符串形式。当然存在一个问题：你如何把值转化为字符串？ 幸运的是，python 已经提供了任何值转化为字符串的各种方法：把值传给方法 repr()或者 str();

Str()函数用来返回一个更方便人读的形式。而 repr()方法用来产生一个解释器方便的形式。（如果没有相等的语句就会产生 syntaxerror）。对于一个没有可以提供可供人方便读的特殊形式，str()返回值与 repr()返回值相同。对于更多的值例如数字或者类似于列表和字典的结构， 都用这两种方法可以产生同样的表现形式。 特别地，字符串有两个不同的表现实现。例如：

```
>>> s = ’ Hello, world.’
>>> str(s)
’ Hello, world.’
>>> repr(s)
"’ Hello, world.’ "
>>> str(1/7)
’ 0.14285714285714285’
>>> x = 10*3.25
>>> y = 200*200
>>> s = ’ The value of x is ’ + repr(x) + ’ , and y is ’ + repr(y) + ’ ...’
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
... hello = ’ hello, world\n’
>>> hellos = repr(hello)
>>> print(hellos)
’ hello, world\n’
>>> # The argument to repr() may be any Python object:
... repr((x, y, (’ spam’ , ’ eggs’ )))
"(32.5, 40000, (’ spam’ , ’ eggs’ ))"
```

这有两种方法来输出平方和立方的表格：

```
>>> for x in range(1, 11):
... print(repr(x).rjust(2), repr(x*x).rjust(3), end=’ ’)
... # Note use of ’ end’ on previous line
... print(repr(x*x*x).rjust(4))
...
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000

>>> for x in range(1, 11):
... print(’{0:2d} {1:3d} {2:4d}’.format(x, x*x, x*x*x))
...
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
```

10 100 1000

（注意在第一个例子中，用方法 print()在每列中添加了空格：它也可以会在参数之间添加空格）

在例子的方法描述了字符串类中的 str.rjust（）方法应用。它通过在字符串左侧填充指定宽度的空格以致右对齐。还有两个类似的方法:str.ljust()和 str.center()、这些方法不是用来写任何数据，它们会返回一个新的字符串。如果输入的字符串太长，它们不会截断它，而是原样返回。这也许是你的列表局混乱，但是比截断更好，那样会输出错误的值（如果你想截断，你总会可以使用切片操作，如 x.ljust(n)[:n]）

还有一个方法：zfile()，使用零在数字字符串左侧填充（到指定宽度）。它可以理解正负号。

```
>>> ’12’.zfill(5)
’00012’
>>> ’-3.14’.zfill(7)
’-003.14’
>>> ’3.14159265359’.zfill(5)
’3.14159265359’
```

Str.format()方法的基本用法就像下面一样：

```
>>> print(’We are the {} who say "{}!"’.format(’knights’, ’Ni’))
We are the knights who say "Ni!"
```

括号以及包含的字符（称为格式域）会被传入 str.format()的对象所替代。在括号中的数据指代传递给格式化方法对象的位置。

```
>>> print(’{0} and {1}’.format(’spam’, ’eggs’))
spam and eggs
>>> print(’{1} and {0}’.format(’spam’, ’eggs’))
eggs and spam
```

如果关键字参数在 str.format()方法中使用时，它们的值通过参数名指定。

```
>>> print(’This {food} is {adjective}.’.format(
... food=’spam’, adjective=’absolutely horrible’))
This spam is absolutely horrible.
```

位置和关键字参数可以随意结合使用。

```
>>> print(’The story of {0}, {1}, and {other}.’.format(’Bill’, ’Manfred’,
other=’Georg’))
The story of Bill, Manfred, and Georg.
```

’!a’ (apply ascii()), ’!s’ (apply str()) and ’!r’ (apply repr())这些方法可以用来在格式化前转化数值。

```
>>> import math
>>> print(’The value of PI is approximately {}.’.format(math.pi))
The value of PI is approximately 3.14159265359.
>>> print(’The value of PI is approximately {!r}.’.format(math.pi))
The value of PI is approximately 3.141592653589793.
```

字符名称后面可以跟一个可选的“:”符号和格式化分类符，这也是如何更好的控制格式化值得方法。下面的实例将 PI 小数点后截取为三位。

```
>>> import math
>>> print(’The value of PI is approximately {0:.3f}.’.format(math.pi))
The value of PI is approximately 3.142.
```

在“:”后传递一个整数将会设置这个字符宽度的最小字数。这个对美化表格很有用途。

```
>>> table = {’Sjoerd’: 4127, ’Jack’: 4098, ’Dcab’: 7678}
>>> for name, phone in table.items():
... print(’{0:10} ==> {1:10d}’.format(name, phone))
...
Jack ==> 4098
Dcab ==> 7678
Sjoerd ==> 4127
```

如果你有一个比较长的格式化字符，但是你不想把它分开， 使用名称代替位置来引用被各式化的变量将更好，这可以简单通过传递一个字典，并且使用方括号‘【】’访问所有的主键。

```
>>> table = {’Sjoerd’: 4127, ’Jack’: 4098, ’Dcab’: 8637678}
>>> print(’Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; ’
’Dcab: {0[Dcab]:d}’.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

通过用“**”把表作为关键字参数来实现这个功能，例如

```
>>> table = {’Sjoerd’: 4127, ’Jack’: 4098, ’Dcab’: 8637678}
>>> print(’Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}’.format(
**
table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

这在与新的内置函数 vars()结合时尤为有用，它返回一个包含所有本地变量的字典，字符串格式化方法 str.format()的完成介绍请参考 Format String Syntax

5.1.1 旧式字符串格式化

%操作符也可以用来格式化字符串，它像：cfunc:’sprintf-’格式化字符串风格一样解释左参数并作用于右参数，并且从该格式化操作中返回字符串结果，例如：

```
>>> import math
>>> print(’The value of PI is approximately %5.3f.’ % math.pi)
The value of PI is approximately 3.142.
```

由于 str.format（）是新增方法，python 的许多代码仍然用%操作符。但是，由于这个旧格式化风格最终会从语言中移除掉。Str.format 就会普遍应用。

更多的信息请查看旧式字符串格式化 部分。

5.2 文件读写

Open()方法返回一个文件对象，在大多数情况下传递两个对象: open（filename, mode）；

例如：

```
>>> f = open(’/tmp/workfile’, ’w’)
```

第一个参数是包含文件名称的字符串，第二个参数是包含描述文件使用方式的字符串。如果文件只读标记为“r”,只写标记为“w”（相同名字的已经存在文件将会被清除）,,“a”表示添加到文件结尾，数据就会自动的添加到文件的结尾。“r+”表示可读可写。Mode 参数是可选择的，如果没有此参数，默认表示可读的。

正常来说，文件会在 text 模式下打开，这就意味着你可以读写字符串到文件中，并且可以设置读写的编码格式（默认编码格式是 utf-8）.’b’表示用二进制的方式打开文件。这些数据将

会用字节对象形式进行读写。这种模式可以用在所有的非文本格式文件中。

在文本格式中，默认的将平台特殊的换行结束符（在 Unix 是\n，在 windos 是\r\n）换成\n 并在写入文件时转换回去。这是文件修改是无害的，但是在 JPEG 或者 EXE 文件中会破坏二进制数据。所以当读写这类文件时最好用二进制格式。

5.2.1 文件对象中方法

在这节所有的例子中假设一个名 f 的文件已创建。

为了读取文件的内容，调用 f.read(size). 这个方法就能读取一定大小的数据并且返回一个字符串或者字节对象。Size 是可选择的数字参数。当 size 缺少或者是负数，返回的就是整个文件内容。如果文件内容是你内存的两倍大，那就是你的问题了。不过，你应该尽可能大字节的读取文件内容，如果已经达到文件尾，f.read()将返回一个空字符串（“)、

```
>>> f.read()
’ This is the entire file.\n’
>>> f.read()
’’
```

F.readline()就是从文件中读取单行，字符结尾会带一个新的字符“\n”，并且仅当在文件最后一行并没有换行符时才会省略。如果 f.readline()会返回一个空字符串，则表示已经达到文件的末尾。当用“\n”表示一个空行时，将返回一个只含有一个换行符的字符串。

```
>>> f.readline()
’ This is the first line of the file.\n’
>>> f.readline()
’ Second line of the file\n’
>>> f.readline()
 ‘’
```

F.readlines()返回一个包含文件中多行数据的列表。如果传入一个可选择的 sizehint，它就会从文件中读取至少包含指定字节数的完整行并返回之。这个方法通常用来高效地从大文件中逐行读取数据，而不会把整个所有文件内容加载到内存中。此方法只返回完整的文件行。

```
>>> f.readlines()
```

[’This is the first line of the file.\n’, ’Second line of the file\n’]

另一种可供选择读取多行的方法就是循环文件对象。这是一种内存高效，快速和并且代码简洁的方式。

```
>>> for line in f:
... print(line, end=’’)
...
This is the first line of the file.
Second line of the file
```

后一种方法虽然代码简洁但是不能提供更加细节控制能力，由于两个方法管理不同的行缓存，千万不要搞混。

F.write(String) 把字节内容写入到文件中，返回写入到文件的字符数量。

```
>>> f.write(’This is a test\n’)
15
```

除了写字符串也可以写入其他内容，但是必须首先转换为字符串。

```
>>> value = (’the answer’, 42)
>>> s = str(value)
>>> f.write(s)
18
```

F.tell()返回一个指文件对象在文件中的当前位置的整数。表示从文件开头到当前位置的字节数。想要改变文件对象的位置，用方法 f.seek(offset, from_what). 位置是通过添加 offset 到参考点计算出来的。参考点通过 from_what 参数来选择。0 的 from_what 值意思就是从文件开头开始。值为 1 时代表从当前文件位置开始计算，值为 2 时表示从文件为开始计算。From_what 参数可以省略并且其默认值为 0，即使用文件开头作为默认参考点。

```
>>> f = open(’/tmp/workfile’, ’rb+’)
>>> f.write(b’0123456789abcdef’)
16
>>> f.seek(5) # Go to the 6th byte in the file
5
>>> f.read(1)
b’5’
```

```
>>> f.seek(-3, 2) # Go to the 3rd byte before the end
13
>>> f.read(1)
b’d’
```

在文本文件中（在字符串模式下用‘b’打开文件），仅仅婿从文件头开始计算相对位置（使用 seek(0,2)从文件尾计算时就会引发异常）

当你使用一个文件时，调用方法 f.close()来关闭它，然后释放任何被打开文件所占用的系统资源。在调用方法 f.close()之后，试图再次调用文件对象就会自动失败。

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

当处理文件对象时候，用 with 关键字是个很好的习惯。它有个益处： 在文件一系列操作结束后它就可以正确地关闭。从某种角度而言 异常也有值得表扬一面。它也比同等功能的 try-finally 代码块更简洁。

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

文件对象有许多其他方法，例如 isatty()和 truncate()，这些方法相对使用的不太频繁。详细请查看库文件指导文档。

5.2.2 pickle 模块

字符串很容易从文件中读写。数字则需要更多的处理，由于 read()方法只能返回字符串，因此它不得不传给像 int()函数，该方法能接受字符串“123”而返回数值 123. 但是，当你想更复杂的数据类型时候，例如列表，字典或者类实例，这些变得更加复杂。

为了不让用户不同编码和调试代码来保存复杂数据类型，python 提供了一个叫 pickle 的标准

接口。这是一个能处理任何 python 对象的神奇模块。（甚至一些格式的 python 代码）。并且把它转化为字符串。这个过程叫做 pickling（封装）。 从字符串形式来重建这个对象就成 unpickling（拆封）。在拆封和封装过程中，表现对象的字符串形式会保存在文件或者数据中，或者通过网络连接发送给远程机器。

如果你有个对象 x，和一个正在打开写的文件对象 f，最简单封装对象的方法仅仅用一行代码：

pickle.dump(x, f)

如果 f 是一个正在读的文件对象，可用如下代码重构这个对象：

x = pickle.load(f)

（当封装许多对象时候或者当你不想写封装的数据到文件，这里还有其他方法，具体请咨询在 python 库文件引用有关 pickle 的完整文档。）

Pickle 是一个来创建需要保存或者被其他程序或者同一个程序将来需要重新创建的 python 对象的标准方法。术语称为 persistent 对象（持久化对象）。Pickle 模块被如此广泛的使用。许多 python 扩展开发者都非常注意像矩阵这样的新数据类型是否可以被适当的封装和拆封。

第六章 类

相比其他编程语言，python 类机制用最少的语法和语义来添加类。它是 C++和 modula-3 类机制的结合结果。Python 类提供了面向对象的所有特性：类继承机制允许继承多个基类。一个子类可以重新基类的所有方法，一个方法可以用同样的方法名调用基类的方法。对象能包含任何数量和类型的数据。很模块一样，类带有 python 自身动态本性。它们在运行时创建，创建之后在将来可以修改。

在 C++术语中，正常来说类成员（包括数据成员）是公共的。所有的成员方法是虚拟的。就像在 Modula-3 一样，要从方法中引用对象成员是没有捷径的。方法功能就是用一个明确的首要参数用来显示对象。这个对象当方法被调用时会具体提供。在 smalltalk 语句中，类本身是对象，它们提供导入和重命名语法。不像 C++和 Modula-3,内置类型都可以作为基类让用户进行扩展。而是想在 C++,许多带有具体语法的内置操作符（算术运算符和下标）可以可以类实例。

关于类因为缺少普通的可以接受的术语，我暂时使用 smalltalk 和 C++中的术语（我更想使用 Modula-3）的术语，因为他的面向对象机制比 C++更接近 Python，但我想几乎没人听说过它。）

6.1 名称和对象相关术语

对象有自己的特性，多个名称（在多个作用中）可以绑定在同一个对象上。这在其他语言中称为别名。在对 python 中的第一印象中通常会被忽略。在处理不可变基本对象时（数字、字符串和元组）时可以放心忽略。但是，别名对于涉及到可变对象（如列表，字典和其他类型)的 python 源码语法时可能产生意想不到到的效果。这通常有利于代码的优化，因为别名在一些方面可以像指针使用。例如，你可以轻易的传递一个对象，因为通过继承可以传递指针。如果函数修改被作为参数传递过来的对象，调用者可以接受这一变化。这消除了传递两个不同参数机制的需要，就像在 Pascal.

6.2 Python 作用域和命名空间

在介绍类之前，首先我想告诉你一些关于 python 作用域的规则。类的定义非常巧妙地运用了命名空间，你需要知道范围和命名空间的工作原理以能全面了解接下来发生的。 顺便说一下，关于这节讲到的知识对于任何优秀的 python 程序员非常有用。

让我们开始以一些定义开始。

命名空间（namespace）是一个从名称到对象的映射。大多命名空间目前用 Python 字典实现的，但那通常不会被注意（除非为了性能），在将来它可以改变。命名空间的例子是：内置名称的 set（包含函数如 abs()和内置异常名称）；在模块中的全局变量名称；在函数调用时的局部名称。在一定程度上对象的属性赋值形成一个命名空间。掌握命名空间的重要事情是在不同的命名空间绝对没有关系。例如，两个不同的模块都可以不混淆的定义方法 maximize。模块的用户必须用模块名称为前缀。

随便说一下，我习惯上吧每一个跟在点号（.）后面的属性都称为属性（attribute）。例如在表达式 z.real。Real 是对象 z 的一个属性。严格意义上讲，在模块中引用的名称都是属性的引用：在表达式 modname.funcname,modname 是一个模块对象和 funcnam 是它的一个属性。 在这个例子，这恰好是在于模块属性和在模块定义中的全局变量名称之间的一个简单的映射：它们共享同样的命名空间。

属性是可读的或者是可写的。在后一种情况下，允许对属性赋值。如果模块属性是可写的，你可以这么写，modername.the_answer = 42. 可写属性也可以用 del 语句删除。例如，del modname.the_answer 将会从名叫 modname 模块中移除属性 the_answer。

命名空间可以在不同的时间里存在并且有不同的生命周期。当 python 解释器启动时，包含内置名称的命名空间就会创建。并且从不删除。当模块定义读入时，模块的全局命名空间就会创建。正常来说，模块命名空间一直存在直到解释器退出。通过解释器的顶层调用执行，从脚本文件中读取或者交互，都认为是_main_模块的一部分，因此他们也有自己的全局命名空间。（内置名称实际也存在于一个模块，称为 builtins.）

当函数调用时函数的局部命名空间就会创建，当函数返回值或者抛出在方法中没有处理的异常时，就会删除。当然，每个递归调用都有自己的局部命名空间。

作用域就是一个 python 程序可以直接访问命名空间的正文区域。这里“直接访问”的意思就是一个名称的非法引用试图在命名空间中寻找名称。

尽管作用域都是静态定义，但是它们动态使用。在执行过程中的任何时候，至少有给三个关联的命名空间可以直接访问的作用域：

- 首先被查的是包含局部变量的最内层作用域
- 任何关闭函数的作用域，它们以最近封装的作用域开始进行查询，包含的不是局部变量也不是非全局变量。
- 接着查询包含当前模块全局变量的作用域。
- 最后查询的就是最外面的作用域，它是包含内置方法的命名空间。

如果名称定义为全局的，那么所有的引用和赋值都可以直接给包含模块全局变量的中间作用域。为了重新绑定在最内层作用域外面发现的变量，nonlocal 语句可以使用。如果没有定义为非本地，这边变量只能读取。（读取这种变量的尝试就会在最内层作用域中产生一个本地局部变量，而外部那个相同标识符的变量不会改变）

通常，局部作用域引用当前函数的局部变量。函数外面，局部作用域引用引用和全局作用一样的命名空间：模块命名空间。类定义也会在局部作用域中引入另一个命名空间。

知道作用域可以在文本中定义是非常重要的。在模块中定义函数的全局作用域是那个模块的命名空间，不管函数从哪里或者用何种名称调用。另一方面，对名称的真正查询是在运行时候动态查询的。但是，语言的定义正在向编译时静态名称确定进化，因此不要依赖动态名称解决。（事实上，局部变量已经静态定义了）

Python 一个特别之处是--如果没有全局变量有效--名称的赋值常常进入最内层的范围。赋值不会拷贝数据--它们紧紧是把名称绑定在对象上。删除也是一样。Del 语句就会移除从局部作用域的命名空间去掉与 x 的绑定。事实上，介绍新名称的所有操作都用局部变量，特别是，import 语句和函数定义在局部局部作用域中绑定模块或者函数名称。

Global 语句可以用来描述活动在全局作用域中的特别变量并且应该绑定在那里。Nonlocal 语句描述活动在封装作用域中的特别变量并在那里绑定。

6.2.1 作用域和命名空间实例

下面的实例主要用来示范如何引用不同的作用域和命名空间，关键字 global 和 nonlocal 如何影响变量绑定。

```
def scope_test():
    def do_local():
        spam = "local spam"
    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"
    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

实例运行结果是：

After local assignment: test spam

After nonlocal assignment: nonlocal spam

After global assignment: nonlocal spam

In global scope: global spam

注意局部赋值（这是默认的）不能改变 scope_test 方法中的 spam 的绑定值。但是 nonlocal

赋值可以改变它的值。全局变量赋值只能改变与模块同一级的值。

你可以看到在全局变量赋值前没有给 spam 绑定值。

6.3 初识类

类引入一些新语法:三种新的对象类型和一些新的语义。

6.3.1 类定义语法

类定义的最简单形式如下：

class ClassName:

<statement-1>

.

.

.

<statement-N>

类定义和函数定义（def 语句）一样，必须先执行然后才生效。（你当然可以把类定义放在 if 语句分支中或者嵌入在函数中）

在实际情况中，在类定义中的语句常常是方法的定义，但是其他语句也是允许的并且有时是很有用的---以后我们会讨论到这点。在类中的定义方法通常有一个参数列表的特殊形式，用于方法的调用约定--再者这将来会解释。

当进入类定义时，新的命名空间将会创建且当用局部作用域来用-因此所有的有关局部变量的参数将会进入新的命名空间。特别地， 函数定义绑定新函数的名称。

当类定义完成时，（通过结尾），一个类对象就会产生。这个是由类定义所创建的包含命名空间内容的基本包装。我们将会在接下来的部分了解更多的关于类对象的知识。类对象在这里绑定在类定义文件开头给出的类名称。（就是实例中的 ClassName）

6.3.2 类对象

类对象支持两种类型操作：属性引用和实例化

在 python 中属性引用用标准的语法来操作所有的属性引用：obj.name.合法的属性名称就是当类对象创建时在类命名空间中的所有名称。因此，如果类定义如下：

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

则用 MyClass.i 和 MyClass.f 是合法的属性引用，各自返回一个整型和函数对象。类属性也可以进行赋值，因此你可以用通过赋值来来改变 MyClass.i 的值。_doc_是一个合法属性，返回一个属性类的文档字符串。；“一个简单类例子”

类实例就是用函数符号。假设类对象就是无参数的函数，它返回一个类的实例。 例如：（假设上面的类）

x = MyClass()

创建一个类实例和把这个对象赋值给局部变量 x.

实例化操作（调用类对象）将会创建一个空对象。许多类可以用设置特定的初始状态来创建对象。因此类可以定义一个名叫_init()的特殊方法，如下：

```
def __init__(self):
self.data = []
```

当类定义_init()方式时，类实例化就会自动调用_init_()方法为新创建的类实例。因此在这个实例中，一个新初始化的实例可以通过如下方式得到：

X = MyClass()

当然，_init_方法为了更灵活应用拥有参数。在那种情况下，给予类实例操作的实例将会传递给 init 方法。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)
```

6.3.3 实例对象

现在我们用实例对象做什么呢？实例对象唯一可用的操作就是属性引用。现在有两种合法的属性名称：数据属性和方法。

数据属性相当于 smallTalk 中的实例变量，C++中的数据成员。数据属性不需要申明。像局部连梁一样，当他们初次赋值的时候他们就存在了。例如，如果 x 是上面创建 MyClass 类的一个实例， 下面的代码块表示将会打印值 16.这个值没有任何错误。

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter*2
print(x.counter)
del x.counter
```

实例属性引用的另一中方法是方法。方法是属于对于对象的函数。（在 python 中，术语方法和类实例不是唯一的）：其他的对象类型也有方法。例如，list 对象有称为 append，insert,remove，sort 方法等等。但是，在接下来的讨论中，除非特别说明，我们用术语方法来用对象实例的方法。）

一个实例对象的合法方法名称取决于它的类。按照定义，一个类中所有函数对象定义了相对应的实例方法。因此，在我们例子中，x.f 是一个合法的方法引用，因为 MyClass.f 是一个方法，但是 x.i 不是，因为 MyClass.i 不是。但是 x.f 和 MyClass.f 不同，它时一个方法对象，而不是函数对象。

6.3.4 方法对象

通常来说，方法在绑定之后就会被调用。

X.f（）

在 MyClass 实例中，它将会返回字符串’hello world’.但是，你无需立刻调用方法：x.f 是方法对象，可以暂被保存然后后来再调用，例如：

```
xf = x.f
while True:
    print(xf())
```

将会不断打印 hello world 直到程序终止。

当方法调用时具体发生了什么？你可能已经注意到了,尽管函数定义 f()时可以有个参数，但 x.f（）调用时候没有传参数。那参数发生了什么？当要求参数的函数没有任何参数而调用时，python 一定会抛出异常。即使参数真正什么也没用。

事实上，你可以猜测答案： 方法有一个特性就是实例对象被当做第一个参数传递给了函数。在我们的例子中，x.f（）方法等同于 MyClass.f（x）.一般来说，调用一个包含 n 个参数列表的方法等同于相应的函数，这个函数包含一个在首次插入方法对象时创建的列表。

如果你仍然不明白方法如何工作，看看它的实现可能就会明白真相。 当一个实例属性引用一个不是数据属性时，它的类是可搜索的。如果这个名称表示一个合法函数对象属性，通过把在抽象类中发现的（指针）实例对象和函数对象封装一起进而创建了方法对象。当方法对象用一个列表参数调用时，一个新的参数列表将会从实例对象和参数列表中创建，并且函数对象用新的属性列表调用。

6.4 一些说明

数据属性可以重写同名的方法属性。这是为了避免在大型系统中产生问题的意外名称冲突。所以用一些减少冲突的常用方法是很有效果的。常用的方法包括：大写字母方法名称，用唯一的字符串来做为数据属性的名称（可以是个下划线_）或者用动词命名方法和用名字命名数据属性。

数据属性就像和对象的普通用户一样可以被方法引用。换句话说，类不能用来实现纯净的数据类型。事实上，在 python 中不能强制数据隐藏，一切基于约定。（另一方面，如 C 中写的，python 的实现可以做到完全隐藏实现细节并且在必要是可以控制对象的访问，这可以通过 C 语言扩展 Python）

客户应该谨慎使用数据属性，客户可能会混淆通过方法来维护的常量。而践踏他们的数据属性。注意只要能避免重复，客户可以自己添加数据属性给对象实例，而不影响方法的合法性--再次，命名约定可以避免很多麻烦。

从方法内部引用数据属性（或者其他方法）是没有便捷方式的。 我发现这可以增加方法的可读性，当浏览一个方法时，不会轻易混淆局部变量和实例变量。

常常，方法的第一参数称为 self。 这里除了约定在没有其他意思，名称 self 对 python 绝对没有特别的含义。但是，要注意，如果不遵守这种约定，你的代码可能对其他 python 程序员来说可读性很差。也可以理解，类浏览程序可能就是基于这种约定的写成的。

任何作为类属性的函数对象定义了一个该类实例的方法。在类定义中在书面上函数定义是封装的有时是没有必要的，也可以把一个函数对象赋值给在类中的局部变量。例如：

```
# Function defined outside the class
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 都是指向函数对象类 C 的所有属性，因此他们都是类 C 实例的所有方法--C 其实和 g 是等价的。 需要注意的是，这个习惯只会让让程序的读者迷惑。

方法可以通过用方法属性 self 可以调用其他方法。

```
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)
```

方法可以用和引用普通函数相同的方法使用全局名称。和方法相应的全局作用域是包含该方法定义的模块。（一个类从不能当做全局作用域来用）虽然在方法中使用全局变量很少有好的理由，但是全局作用域有很多合法的用处。首先，引入全局作用域的函数和模块可以当方法，如同定义在它里面的函数和类一样被使用。通常，包含方法的类在这个全局作用域中被定义。在接下来的一节，我们将会找到更加充分的理由来解释为什么方法可以引用它自己的类。

6.5 继承

当然，一门语言特性如果不支持继承那么名称类就失去了价值。子类继承父类的方法如下：

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

名称 BaseClassName 必须定义在一个包含派生类定义的作用域中。在基类名称的位置上，其他随意表达式都是允许的、例如，当基类定义在其他模块中，这也是可用的。

class DerivedClassName(modname.BaseClassName):

派生类的执行过程和父类执行过程是相同的。当类对象构造好后，基类就被记住了。这为了解决属性引用问题：如果一个请求的属性在类中没有发现，程序就会搜索它的基类。如果基类它本身也是从其他类派生而来，那么就系继续搜素。

对派生类的实例化没有特别之处。DeriverdClassName()创建了一个类的新实例。方法引用按照如下使用：对应的类属性可以被搜索的，如果需要就会继续在基类中查询。如果这指向函数对象，则方法引用是合法的。

派生类可以重写基类的方法。因为当调用同对象的其他方法时候，方法没有特殊的权限。调用定义在同一基类的该基类方法可能终止派生类方法的覆盖。（对于 C++程序员来说，在 python 中所有的方法都是有效虚拟的）

在派生类中一个重写方法事实上想扩展方法，而不是简单的代替同名的基类方法。最简单直接调用基类的方法就是用： BaseClassName.methodname(self, arguments)。这个对客户也非常有空。（注意，如果基类在全局作用域中和 BaseClassName 都可以访问， 这种方法才有效）

Python 有两种实现继承的内置函数：

方法一： 用 isinstance（）方法来检查实例类型：只有 obj._class_is 是整型或者其他从 int 继承的类，isinstance(obj, int)才会返回真。

方法二： 用 issubclass()方法来检查类继承：issubclass(bool, int)结果是真得，因为 bool 是 int 的一个子类。但是 issubclass(float, int)是假的因为 float 不是 int 的子类。

6.5.1 多重继承

Python 也支持多种继承形式。一个能继承多个基类的类定义如下：

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

大多数情况，最简单而言，你可以把从父类继承下来的属性查询看成是遵循深度优先，从左到右。

而不是在同一等级重复的同样类中执行两次。因此，如果一个属性没在派生类中找到，首先会在 base1 然后再 base1 的基类中，如果在那里都没发现，就会在 base2 中查找等等。

事实上，比刚才说的稍微有点复杂。方法执行顺序为了协同调用 super()而动态变化。这种方法在一些支持多重继承的语言中因调用接下来方法非常出名。并且要比单继承语言中的 super 调用更强大。

动态排序是必须的，因为多重继承中所有情况显示至少一个菱形关系。（从最底层的类开始，至少存在一个父类可以通过多条路径访问）。例如，所有的的类继承 object, 因此多种继承的每种情况提供至少一种路径到达 object. 为了保证至少有一种方法访问基类，动态算法用一种特别方法线性化了搜索顺序。这种方法就是保证在每个类按从左到右的顺序，每个父类只有一次，并且那是不变的。（即继承一个类不会一项它父类的优先级），总之，这些属性让使用多重继承设计可靠的和可扩展的类成为可能。更多信息请参考：

6.6 私有变量

除了对象内部其他都不能访问的“私有”变量在 python 中是不存在的。但是，大多数 python 代码都遵守一个规则：以下划线为前缀的名称被看成是 API 的非公共部分。它可以认为是一个细节实现并且改变时不需通知。

因为存在类私有成员的合法用例（即为了避免在子类定义的名称想冲突），所有存在对这种机制的限制支持。这种机制叫名称变换。_spam 形式的任何标识符（至少是两个开头下划线，最多一尾下划线）在书面上可以被_classname_spam 代替，这里 classname 是当前的类名。 只要这种变换出现在类定义中，那么这个不涉及到标识符的语法位置就可以处理。

名字变换对于让子类在不影响父类方法前提下重载方法很有重要意义。例如：

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

这种变换规则最初设计是为了避免冲突，如果强意要访问或者修改一个被认为私有变量，仍然是可以的。在一些特别的情况下，这种方法更加有用。例如调试器。

注意传给 exec()或者 eval()的代码不会将调用类作为当前类，这个很 global 全局变量的效果相似。它的作用限制于一起进行字节码编译的代码。同样的限制也适用于 getattr()，settattr()和 delattr()函数，以及直接引用_dict_时。

6.7 备注

有时有个像 Pasca 中“记录”和 C 中“数据体”的数据类型非常有用。集合一些数据项。一个空类定义可以清楚地显示：

```
class Employee:
    pass
john = Employee() # Create an empty employee record
# Fill the fields of the record
john.name = ’John Doe’
john.dept = ’computer lab’
john.salary = 1000
```

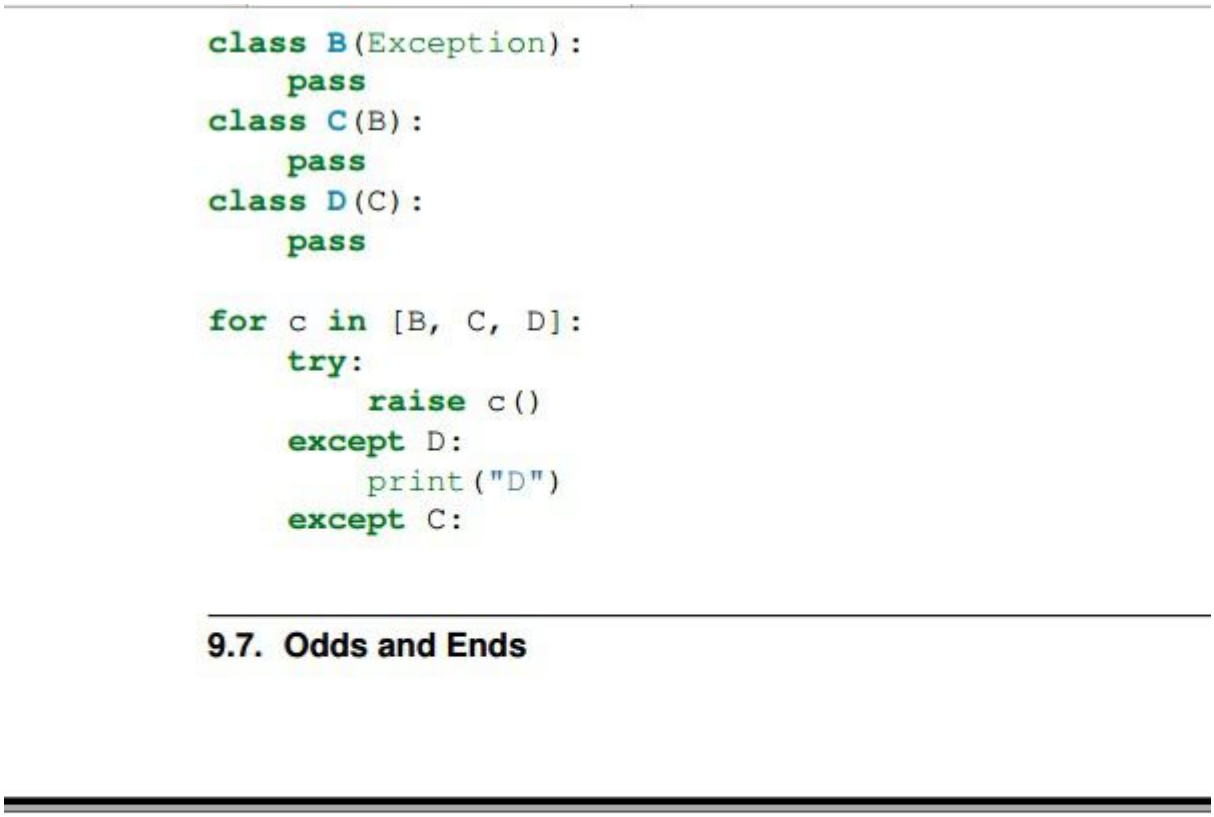
期望得到一个特殊抽象对局类型的 python 代码块可以传递给类。这个类可以模仿那种数据类型方法。例如，如果你有个从文件对象格式化数据的函数，你能定义一个包含方法 read 和 readline 的方法来获得数

据，然后把它作为一个参数传递给他。

实例方法对象也有许多属性：m.__self_是一个包含方法 m()的梳理对象，并且 m._func_是和该方法对应的函数对象。

6.8 异常也是类

在异常中的类和一个异常是可以兼容，这里的异常是指同一个类或者是一个基类（但是不能反过来说-遍历派生类的异常语句不能和基类相兼容）例如，接下来将按照顺序打印 BCD



```
class B(Exception):
    pass
class C(B):
    pass
class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
        print("D")
    except C:
```

9.7. Odds and Ends

Python Tutorial, Release 3.2.3

```
        print("C")
    except B:
        print("B")
```


注意如果异常语句进行翻转（except 在前），它将会打印出 BBB，-最先满足的异常语句就会触发。

当一个没有处理的异常的错误信息打印出来时候，异常类名就打印。接下来是分号和空格，最后是用内置方法转把实例换为字符串

6.9 迭代器

到目前为止，你可能已注意到许多容器对象都可以用 for 语句进行循环：

```
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)
```

这种访问风格清楚简洁方便。 迭代器的应用是 python 遍历统一。在这种场景背后，for 语句调用容器对象 iter()方法。函数返回一个迭代器对象。在迭代器对象里定义了每次只能访问一个元素的方法_next_()。当容器里面没有元素的时候，该方法就会抛出一个 stopInteration 异常，用来提醒 for 循环终止。你可以用内置的 next()调用_next_方法：以下这个例子显示它时如何工作的。

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    next(it)
StopIteration
```

想必你已经看到迭代器背后的实现机制，给你自己的类添加一个迭代器是很容易实现的。定义一个_iter_（）方法用来返回一个包含_next_（）方法的对象。如果类中定义了_next_()方法，那么_iter_()就会返回本身。

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    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 = self.index - 1
        return self.data[self.index]
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

6.10 生成器

生成器是一个创建迭代器的简单而有力的工具。它们书面写时就像规范的函数，但是用 yield 语句在任何时候都可以返回数据。每次在它上调用 next（）方法，生成器继续回到一起它离开的位置。（它记录所有数据值以及最后执行的语句）。下面自己就是展示生成器如何方便创建：

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]

>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

用生成器可以做任何于底层迭代器相关的事情，正如前面章节中描述的。让生成器更加紧凑的办法就是自动创建的_iter_()和_next_()方法。

生成器其他重要的特性就是在调用过程中局部变量和执行状态可以自动的保存。这让函数更加容易编写。也比通过调用实例变量像 self.index 和 self.data 更加清楚。

除了自动创建方法和保存程序状态之外，当生成器终止后，它们自动抛出 stopIteration 异常。总的来说，这些特性都让创建迭代器更加有效率简单。

6.11 生成器表达式

有些简单的生成器可以使用类似列表推导式的符号简单编码为表达式，但无序带有中括号。 这种表达式是专门为某种场景而设计的，在那里，生成器被一个封闭函数所使用。生成器表达式比完成生成器定义更加简单但是缺乏通用性。而且比等价的列表表达式更容易记住。

```
>>> sum(i*i for i in range(10))                 # sum of squares
285

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


>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # dot product
260

>>> from math import pi, sin
>>> sine_table = {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']
```

第七章 Python 标准库概览

7.1 操作系统接口

Os 模块提供主要许多与操作系统交互的函数。

```
>>> import os
>>> os.getcwd() # Return the current working directory
’C:\\Python31’
>>> os.chdir(’/server/accesslogs’) # Change current working directory
>>> os.system(’mkdir today’) # Run the command mkdir in the system shell
0
```

一定要用 import os 方式代替 from os import *. 这会使 os.open()方法覆盖内置的 open()函数。因为它们操作有很大的不同。

内置方法 dir()和 help()方法对交互的使用像 os 这种大模块非常有用。

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

对于日用文件和目录管理任务，shutill 模块提供一个更高级别的并方便使用的接口。

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
>>> shutil.move('/build/executables', 'installdir')
```

7.2 文件通配符

Glob 模块提供一个函数用来从目录通配符搜索中生产文件列表。

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

7.3 命令行参数

共同的工具脚本常常需要提供命令行参数。这些参数作为列表保存在 sys 模块中 argv 属性中。例如，接下来输出通过在命令行运行 python demo.py one two three 得到的结果。

```
>>> import sys
>>> print(sys.argv)
['demo.py', 'one', 'two', 'three']
```

Getopt 模块用 Unix 的习惯 getopt()函数来运行 sys.argv. 在 argparse 模块提供了许多更加作用强大和灵活的命令行操作。

7.4 错误输出重定向和程序终止

Sys 模块还包括许多属性如 stdin，stdout 和 stderr。后面的属性通常用来抛出警告或者错误信息，当 stdout

重定向时候也可以看到错误信息。

终止脚本的最直接方法就是用 sys.exit()方法。

```
>>> sys.stderr.write(’ Warning, log file not found starting a new one\n’ )
Warning, log file not found starting a new one
```

7.5 字符串模式匹配

re 模块为高级字符串成处理提供了正则表达式匹配。 对于复杂的匹配和处理，正则表达式能够提供简明优化的方法：

```
>>> import re
>>> re.findall(r’ \bf[a-z]*’ , ’ which foot or hand fell fastest’ )
[’ foot’ , ’ fell’ , ’ fastest’ ]
>>> re.sub(r’ (\b[a-z]+) \1’ , r’ \1’ , ’ cat in the the hat’ )
’ cat in the hat’
```

当仅仅需要一些简单的功能时候，优先使用 string 方法，因为它更容易读取和调试。

```
>>> ’ tea for too’ .replace(’ too’ , ’ two’ )
’ tea for two’
```

7.6 数学

数学模块为浮点数运算提供了对底层 C 函数库的访问支持。

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

Random 模块为生成随机选择提供了工具。

```
>>> import random
>>> random.choice([’ apple’ , ’ pear’ , ’ banana’ ])
’ apple’
>>> random.sample(range(100), 10) # sampling without replacement
```

```
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random() # random float
0.17970987693706186
>>> random.randrange(6) # random integer chosen from range(6)
4
```

SciPy project<https://sicpy.org> 项目中包含许多数据计算的模块。

7.7 互联网访问

Python 中有许多访问互联网和处理互联网协议的模块。其中最简单的两个就是从链接中获得数据的 urllib.request 和发送邮件的 smtplib.

```
>>> from urllib.request import urlopen
>>> for line in urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
...     line = line.decode('utf-8') # Decoding the binary data to text.
...     if 'EST' in line or 'EDT' in line: # look for Eastern Time
...         print(line)
<BR>Nov. 25, 09:43:32 PM EST
>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(注意第二个例子需要有一个在本地运行的 email 邮箱服务器)

7.8 时间和日期

Datatime 模块提供一些用简单或复杂方式处理时间和日期的类。当处理日期和时间数据时，

格式化输出和处理实现的重点就是高校的成员提取。这个模块同样支持时区处理。

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
’12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.’
>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

7.9 数据压缩

Python 还支持常用数据的打包和压缩。主要涉及到的模块式 zlib,gzip, bz2,zipfile and tarfile.

```
>>> import zlib
>>> s = b’witch which has which witches wrist watch’
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b’witch which has which witches wrist watch’
>>> zlib.crc32(s)
226805979
```

7.10 性能评测

一些 python 使用者对于同一问题的不同解决办法的性能很感兴趣。Python 提供了一种评测工具就可以马上回答这些问题。

例如， 当封装参数的时候可以用元组封装和拆封特性来代替传统的方法。Timeit 模块中可以迅速描述一个性能优势。

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

与 timeit 的细粒度相比，profile 和 pstate 模块提供了在大代码块中识别时间临界区的工具。

7.11 质量控制

开发高质量的软件的方法之一就是对每个功能写测试用例。在开发过程中频繁地运行这些用例。

Doctest 模块提供一个扫描模块和验证嵌套在程序文档字符中的测试。测试编制是简单的把一个典型的调用及它的结果剪切并粘贴到文档字符串里。这通过为用户提供一个实例改善了文档，并且它允许 doctext 模块确认代码和文档相符。

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod()    # automatically validate the embedded tests
```

Unittest 模块不像 doctest 模块那么容易使用。但是，它允许一个更加复杂的测试来维护分开文件。

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        self.assertRaises(ZeroDivisionError, average, [])
        self.assertRaises(TypeError, average, 20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

7.12 内置电池

Python 有“内置电池”的哲学，因为可以很好看到通过更大包扩展更加复杂和健壮的性能。例如：

- Xmlrpc.client 和 xmlrpc.server 模块可以实现远程程序调用。尽管模块有这样的名字，用户无需拥有 xml 的知识或处理 xml.
- Email 包是一个管理邮件消息的库。包含 MIME 和其他 rfc2822-base 消息文档。不像真正收发消息的 smtplib 和 poplib 模块，email 包有完整的工具，来编译和编码复杂的消息结构（包括附件）。或处理互联网编码和消息头协议。
- Xml.dom 和 xml.sax 包对编译大型数据交互格式提供了非常健全的支持。同样，csv 模块支持从一种通用的数据库格式中直接读写。总之，这些模块和包大大简化了 python 应用程序和其他工具之间的数据交换。

有若干模块可以实现国际化，包括 gettext，local 和 codecs 包。

第八章 标准库二

第二部分涵盖了许多更能满足专业开发人员需求的高级模块。这些模块在小脚本中很少出现。

8.1 输出格式化

Reprlib 模块为大型的或深度嵌套的容器缩写显示提供了 repr()函数的一个定制版本。

```
>>> import reprlib
```

```
>>> reprlib.repr(set('supercalifragilisticexpialidocious'))
"set(['a', 'c', 'd', 'e', 'f', 'g', ...])"
```

Pprint 模块提供了对输出内置函数和用户定义对象更加复杂的控制。这种方式是解释器能够读懂的。当结果多于一行时，“完美打印机”就会增加行中断和随进，一边更清晰的显示数据结构。

```
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
... 'yellow'], 'blue']]]
...
>>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
'white',
['green', 'red']],
[['magenta', 'yellow'],
'blue']]]
```

Textwrap 模块格式化文本段落来适应所给屏幕的宽度。

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
...
>>> print(textwrap.fill(doc, width=40))
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

Local 模块用来访问特殊数据格式的文化数据库。Local 的分组格式化函数属性为数字的分组分隔格式化提供了直接的方法。

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
```

```
’English_United States.1252’
>>> conv = locale.localeconv() # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
’1,234,567’
>>> locale.format_string("%s%.*f", (conv[’currency_symbol’],
... conv[’frac_digits’], x), grouping=True)
’$1,234,567.80’
```

8.2 模板

String 模块包含一个用途广泛的类，此类为最终用户的编辑提供了简单的语法支持。这让用户不修改应用程序的前提下实现他们应用程序的定制。

这种格式使用$加有效的 python 标识符（数字、字母和下划线）形式的占位符名称。通过在占位符两侧使用大括号便可以不用空格分隔在其后面跟随更多的字母和数字字符。使用$$来创建一个单独$转码字符。

```
>>> from string import Template
>>> t = Template(’${village}folk send $$10 to $cause.’)
>>> t.substitute(village=’Nottingham’, cause=’the ditch fund’)
’Nottinghamfolk send $10 to the ditch fund.’
```

当占位符在字典或者关键字参数中没有被提供时，方法 substitute（）就会抛出一个关键错误。对于邮件合并分隔的应用程序中，用户提供的数据可能不完整。这是用 safe_substitude（）方法可能更加适合--如果数据丢失，它不会改变占位符。

```
>>> t = Template(’Return the $item to $owner.’)
>>> d = dict(item=’unladen swallow’)
>>> t.substitute(d)
Traceback (most recent call last):
...
KeyError: ’owner’
>>> t.safe_substitute(d)
```

’Return the unladen swallow to $owner.’

模板派生类可以指定一个自定义分隔符。例如，图像浏览器的批量命名工具类可以选择使用百分号作为占位符，像当前日期，图片序列号或文件格式。

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
>>> fmt = input('Enter rename style (%d-date %n-seqnum %f-format):  ')
Enter rename style (%d-date %n-seqnum %f-format):  Ashley_%n%f

>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print('{0} --> {1}'.format(filename, newname))

img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

针对模板化的另一个应用程序就是从多种输出格式的细节中分离出程序逻辑。这就使得 xml 文件、简单文本报表以及 html 网页定制模板称为可能。

8.3 使用二进制数据记录布局

Struct 模块提供了 pack()和 unpack（）方法来处理可变长度的二进制格式。接下来的例子展示在一个没用 zipfile 模块的 zipfile 如何通过标题信息循环。压缩码“H”和“I”分别表示 2 和 4 字节无符号数字，“<”表明都是标准大小并且按照 little-endian 字节排序。

```
import struct

data = open('myfile.zip', 'rb').read()
start = 0
for i in range(3):                      # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)

    start += extra_size + comp_size     # skip to the next header
```

8.4 多线程

线程是一种针对分离不连续和依赖的任务的技术。用线程可以提高那些允许用户输入的程序响应，同时有其他程序在后台运行。一个相关的应用就是在运行 I/O 的同时另一个线程中执行运算。

下面的代码展示高优先级 threading 模块在后台执行任务时，但是主程序继续运行。

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile
    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')

background.join()    # Wait for the background task to finish
print('Main program waited until background was done.')
```

对于多线程应用最大的挑战就是协调那些共享数据或者资源的线程。到最终，线程模块提供大量同步原语，包括锁，时间，条件变量以及信号量

尽管这些工具功能很强大，但微小的设计错误就可能导致难以挽回的故障。因此，任务协调的首选方法就是把所有对资源访问集中到一个单线程中，然后用 queue 模块来那个线程来服务其他线程的请求。为内部线程通信和协调而用 Queue 对象的应用程序比较容易设计，更加可读，而且更加可靠。

8.5 日志

Logging 模块提供了一些功能全面和灵活的日志系统。最简单的形式就是把日志信息发送到一个文件或 sys.stderr;

import logging

logging.debug(’Debugging information’)

logging.info(’Informational message’)

logging.warning('Warning:config file %s not found', 'server.conf')

logging.error('Error occurred')

logging.critical('Critical error -- shutting down')

上面将会产生如下输出：

WARNING:root:Warning:config file server.conf not found

ERROR:root:Error occurred

CRITICAL:root:Critical error -- shutting down

默认的，提示信息和调试信息都会被捕获，并且把输出发送到标准错误。其他输出可选项包括通过邮件路由信息，数据报，套接字或到一个 http 服务器。新的过滤选择基于信息优先级不同的路由：Debug, info，warning,error 和 critical.

日志系统可以通过 paython 直接配置或者通过用户可编辑的配置文件进行加载，从而实现不修改应用程序而定制日志。

8.6 弱引用

Python 可以实现自动内存管理（对大多对象的引用计数并为消除循环引用做 garbage collection）.在最后一次对对象引用消除后，内存稍后就会释放。

这种方法在大多程序中运行良好，但是偶尔也需要在对象被其他东西使用时追踪对象，不幸的，仅仅为跟踪他们而创建的引用会使持久存在。Weakref 模块提供一些跟踪对象而不需要创建引用的工具。当对象不再需要时，它会自动从 weakref 表中自动移除，并且一个针对 wakref 对象的回滚事务就会触发。典型的应用的创建都是昂贵的，包括缓存对象。

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...             self.value = value
...     def __repr__(self):
...             return str(self.value)
...
>>> a = A(10)                   # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a            # does not create a reference
>>> d['primary']                # fetch the object if it is still alive
10
>>> del a                       # remove the one reference
>>> gc.collect()                # run garbage collection right away
```

```
0
>>> d['primary']                # entry was automatically removed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']               # entry was automatically removed
  File "C:/python31/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

8.7 列表工具

许多数据结构需求能通过内置列表类型满足，但是，有时处于不同性能取舍需要从中选择一种实现。

Array 模块能提供一个像列表的 array 对象，它仅仅能存储同类数据并且更加简洁。接下来例子展示了一个数字数组。存储是 2 个字节的无标识的二进制数据而不是在 python 对象中普通列表中的每个 16 字节的值。

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

Collections 模块通过方法 depue（）提供了一个类似列表对象，它从左边开始能更加快速添加和删除，但是在中间查询时很慢。这些对象很适合实现队列和广度优先查询。

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
>>> print("Handling", d.popleft())
Handling task1

unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

除了代替列表实现之外，标准库还提供了其他工具，比如处理排序列表的 bisect 模块。

```
>>> import bisect
>>> scores = [(100, ’ perl’), (200, ’ tcl’), (400, ’ lua’), (500, ’ python’)]
>>> bisect.insort(scores, (300, ’ ruby’))
>>> scores
[(100, ’ perl’), (200, ’ tcl’), (300, ’ ruby’), (400, ’ lua’), (500, ’ python’)]
```

Headpq 模块为基于正规列表的堆实现提供了函数。最小的值入口总是在位置 0 上。这对那些希望重复访问最小元素而不像做一次完成列表排序的应用过程序很有用。

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data) # rearrange the list into heap order
>>> heappush(data, -5) # add a new entry
>>> [heappop(data) for i in range(3)] # fetch the three smallest entries
[-5, 0, 1]
```

8.8 十进制浮点数计算

十进制模块提供了对十进制浮点数计算的 Decimal 数据类型。相比于内置的二进制 float 浮点实现，此类更加有助于以下情况：

- 需要精确十进制位数表示的财务系统或者其他用途。
- 控制精度
- 控制保留位数以来满足法律或者管理需求
- 重大十进制数的跟踪
- 那些用户想要控制数学计算结果的应用程序

例如，计算在 70 美分电话费中 5%的税收，在十进制和二进制浮点数不同可能导致不同额结果。如果要对最接近的分钟数进行舍入，这种差别就变得很重要。

```
>>> from decimal import*
>>> round(Decimal(’ 0.70’ )*Decimal(’ 1.05’ ), 2)
Decimal(’ 0.74’ )
>>> round(.70*1.05, 2)    0.73
```

Decimal 结果总会保留结尾 0，还会从带有两个小数位的被乘数自动推断为 4 个小数位。Decimal 让数学计算像手动处理一样，并且避免了当二进制浮点数无法精确的表示小数位时可能出现的结果。

高精度使 Decimal 类可以进行那些不适合二进制浮点数的模运算和等式测试。

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> sum([0.1]*10) == 1.0
False
```

Decimal 模块为算术运算提供了高精度的需要

>>> getcontext().prec = 36

>>> Decimal(1) / Decimal(7)

Decimal('0.142857142857142857142857142857142857')