Python Tutorial

发布 *2.7.18*

Guido van Rossum
and the Python development team

五月 20, 2020

Python Software Foundation
Email: docs@python.org

Contents

Python 是一种易于学习又功能强大的编程语言。它提供了高效的高层次的数据结构，还有简单有效的面向对象编程。Python 优雅的语法和动态类型，以及解释型语言的本质，使它成为在很多领域多数平台上写脚本和快速开发应用的理想语言。

多数平台上的 Python 解释器以及丰富的标准库的源码和可执行文件，都可以在 Python 官网 https://www.python.org/ 免费自由地下载并分享。这个网站上也提供一些链接，包括第三方 Python 模块、程序、工具等，以及额外的文档。

Python 解释器易于扩展，可以使用 C 或 C++（或者其他可以从 C 调用的语言）扩展新的功能和数据类型。Python 也可用作可定制化软件中的扩展程序语言。

这个教程非正式地介绍 Python 语言和系统的基本概念和功能。最好在阅读的时候有一个 Python 解释器做一些练习，不过所有的例子都是相互独立的，所以这个教程也可以离线阅读。

有关标准的对象和模块，参阅 library-index。reference-index 提供了更正式的语言定义。要写 C 或者 C++ 扩展，参考 extending-index 和 c-api-index。也有不少书籍深入讲解 Python。

这个教程并不试图完整包含每一个功能，甚至常用功能可能也没有全部涉及。这个教程只介绍 Python 中最值得注意的功能，也会让你体会到这个语言的风格特色。学习完这个教程，你将能够阅读和编写 Python 模块和程序，也可以开始学习更多的 Python 库模块，详见 library-index。

术语对照表 也很值得一读。

CHAPTER 1

课前甜点

如果你经常在电脑上工作，总会有些任务会想让它自动化。比如，对一大堆文本文件进行查找替换，对很多照片文件按照比较复杂的规则重命名并放入不同的文件夹。也可能你想写一个小型的数据库应用，一个特定的图形界面应用，或者一个简单的游戏。

如果你是专业的软件开发人员，你可能需要编写一些 C/C++/Java 库，但总觉得通常的编写、编译、测试、再次编译流程太慢了。可能给这样的库写一组测试，就是很麻烦的工作了。或许你写了个软件，可以支持插件扩展语言，但你不想为了自己这一个应用，专门设计和实现一种新语言了。

那么，Python 正好能满足你的需要。

对于这些任务，你也可以写 Unix shell 脚本或者 Windows 批处理完成，但是 shell 脚本最擅长移动文件和替换文本，并不适合 GUI 界面或者游戏开发。你可以写一个 C/C++/Java 程序，但是可能初稿都要很长的开发时间。Python 的使用则更加简单，可以在 Windows，Mac OS X，以及 Unix 操作系统上使用，而且可以帮你更快地完成工作。

Python 很容易使用，但它是一种真正的编程语言，提供了很多数据结构，也支持大型程序，远超 shell 脚本或批处理文件的功能。Python 还提供比 C 语言更多的错误检查，而且作为一种“超高级语言”，它有高级的内置数据类型，比如灵活的数组和字典。正因为这些更加通用的数据类型，Python 能够应付更多的问题，超过 Awk 甚至 Perl，而且很多东西在 Python 中至少和那些语言同样简单。

Python 允许你将程序划分为能在其他的 Python 程序中重复利用的模块。它内置了很多的标准模块，你可以在此基础上开发程序——也可以作为例子，开始学习 Python 编程。例如，一切内置模块提供诸如文件输入输出、系统调用、套接字、甚至图形界面接口工作包比如 Tk。

Python 是一种解释型语言，在程序开发阶段可以为你节省大量时间，因为不需要编译和链接。解释器可以交互式使用，这样就可以方便地尝试语言特性，写一些一次性的程序，或者在自底向上的程序开发中测试功能。它也是一个顺手的桌面计算器。

Python 程序的书写是紧凑而易读的。Python 代码通常比同样功能的 C，C++，Java 代码要短很多，有如下几个原因：

- 高级数据类型允许在一个表达式中表示复杂的操作；
- 代码块的划分是按照缩进而不是成对的花括号；
- 不需要预先定义变量或参数。

Python 是“可扩展的”：如果你知道怎么写 C 语言程序，就能很容易地给解释器添加新的内置函数或模块，不论是让关键的操作以最高速度运行，还是把 Python 程序链接到只提供预编译程序的库（比如硬件相关的图形库）。一旦你真正链接上了，就能在 Python 解释器中扩展或者控制 C 语言编写的应用了。

顺便提一下，这种语言的名字（python 一词直译为“蟒蛇”）得名自 BBC 节目“Monty Python 的飞行马戏团”，而与爬行动物没有关系。在文档中用 Monty Python 来开玩笑不只是被允许的，还是被推荐的！

现在你已经对 Python 跃跃欲试了，想要深入了解一些细节了。因为学习语言的最佳方式是使用它，本教程邀请你一边阅读，一边在 Python 解释器中玩耍。

在下一章节，会讲解使用解释器的方法。看起来相当枯燥，但是对于尝试后续的例子来说，是非常关键的。

教程的其他部分将通过示例介绍 Python 语言和系统中的不同功能，开始是比较简单的表达式、语句和数据类型，然后是函数和模块，最终接触一些高级概念，比如异常、用户定义的类。

CHAPTER 2

使用 Python 解释器

2.1 调用解释器

The Python interpreter is usually installed as `/usr/local/bin/python` on those machines where it is available; putting `/usr/local/bin` in your Unix shell’s search path makes it possible to start it by typing the command

```
python
```

to the shell. Since the choice of the directory where the interpreter lives is an installation option, other places are possible; check with your local Python guru or system administrator. (E.g., `/usr/local/python` is a popular alternative location.)

On Windows machines, the Python installation is usually placed in `C:\Python27`, though you can change this when you’re running the installer. To add this directory to your path, you can type the following command into the command prompt in a DOS box:

```
set path=%path%;C:\python27
```

在主提示符中输入文件结束字符（在 Unix 系统中是 `Control-D`，Windows 系统中是 `Control-Z`）就退出解释器并返回退出状态为 0。如果这样不管用，你还可以写这个命令退出：`quit()`。

The interpreter’s line-editing features usually aren’t very sophisticated. On Unix, whoever installed the interpreter may have enabled support for the GNU readline library, which adds more elaborate interactive editing and history features. Perhaps the quickest check to see whether command line editing is supported is typing `Control-P` to the first Python prompt you get. If it beeps, you have command line editing; see Appendix 交互式编辑和编辑历史 for an introduction to the keys. If nothing appears to happen, or if `^P` is echoed, command line editing isn’t available; you’ll only be able to use backspace to remove characters from the current line.

解释器运行的时候有点像 Unix 命令行：在一个标准输入 tty 设备上调用，它能交互式地读取和执行命令；调用时提供文件名参数，或者有个文件重定向到标准输入的话，它就会读取和执行文件中的 脚本。

另一种启动解释器的方式是 `python -c command [arg] ...`，其中 *command* 要换成想执行的指令，就像命令行的 `-c` 选项。由于 Python 代码中经常会包含对终端来说比较特殊的字符，通常情况下都建议用英文单引号把 *command* 括起来。

有些 Python 模块也可以作为脚本使用。可以这样输入：`python -m module [arg] ...`，这会执行 *module* 的源文件，就跟你在命令行把路径写全了一样。

在运行脚本的时候，有时可能也会需要在运行后进入交互模式。这种时候在文件参数前，加上选项 `-i` 就可以了。

All command-line options are described in using-on-general.

2.1.1 传入参数

如果可能的话，解释器会读取命令行参数，转化为字符串列表存入 `sys` 模块中的 `argv` 变量中。执行命令 `import sys` 你可以导入这个模块并访问这个列表。这个列表最少也会有一个元素；如果没有给定输入参数，`sys.argv[0]` 就是个空字符串。如果脚本名是 `'-'`` (标准输入) 时，``sys.argv[0]` 就是 `'-'`。使用 `-c` 命令时，`sys.argv[0]` 就会是 `'-c'`。如果使用选项 `-m` *module*，`sys.argv[0]` 就是包含目录的模块全名。在 `-c` *command* 或 `-m` *module* 之后的选项不会被解释器处理，而会直接留在 `sys.argv` 中给命令或模块来处理。

2.1.2 交互模式

在终端（tty）输入并执行指令时，我们说解释器是运行在 *交互模式*（*interactive mode*）。在这种模式中，它会显示 *主提示符*（*primary prompt*），提示输入下一条指令，通常用三个大于号（`>>>`）表示；连续输入行的时候，它会显示 *次要提示符*，默认是三个点（`...`）。进入解释器时，它会先显示欢迎信息、版本信息、版权声明，然后就会出现提示符：

```
python
Python 2.7 (#1, Feb 28 2010, 00:02:06)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

多行指令需要在连续的多行中输入。比如，以 `if` 为例：

```
>>> the_world_is_flat = 1
>>> if the_world_is_flat:
...     print "Be careful not to fall off!"
...
Be careful not to fall off!
```

有关交互模式的更多内容，请见交互模式。

2.2 解释器的运行环境

2.2.1 源文件的字符编码

By default, Python source files are treated as encoded in ASCII. To declare an encoding other than the default one, a special comment line should be added as the *first* line of the file. The syntax is as follows:

```
# -*- coding: encoding -*-
```

其中 *encoding* 可以是 Python 支持的任意一种 `codecs`。

比如，要声明使用 Windows-1252 编码，你的源码文件要写成：

```
# -*- coding: cp1252 -*-
```

关于 第一行规则的一种例外情况是，源码以*UNIX* “*shebang*”行 开头。这种情况下，编码声明就要写在文件的第二行。例如：

```
#!/usr/bin/env python
# -*- coding: cp1252 -*-
```

CHAPTER 3

Python 的非正式介绍

在下面的例子中，通过提示符 (»> 与…) 的出现与否来区分输入和输出：如果你想复现这些例子，当提示符出现后，你必须在提示符后键入例子中的每一个词；不以提示符开头的那些行是解释器的输出。注意例子中某行中出现第二个提示符意味着你必须键入一个空白行；这是用来结束多行命令的。

这个手册中的许多例子都包含注释，甚至交互性命令中也有。Python 中的注释以井号 # 开头，并且一直延伸到该文本行结束为止。注释可以出现在一行的开头或者是空白和代码的后边，但是不能出现在字符串中间。字符串中的井号就是井号。因为注释是用来阐明代码的，不会被 Python 解释，所以在键入这些例子时，注释是可以被忽略的。

几个例子:

```
# this is the first comment
spam = 1  # and this is the second comment
          # ... and now a third!
text = "# This is not a comment because it's inside quotes."
```

3.1 Python 作为计算器使用

让我们尝试一些简单的 Python 命令。启动解释器，等待界面中的提示符，>>> （这应该花不了多少时间）。

3.1.1 数字

解释器就像一个简单的计算器一样：你可以在里面输入一个表达式然后它会写出答案。表达式的语法很直接：运算符 +、-、*、/ 的用法和其他大部分语言一样（比如 Pascal 或者 C 语言）；括号 (()) 用来分组。比如:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5.0*6) / 4
```

(下页继续)

(续上页)

```
5.0
>>> 8 / 5.0
1.6
```

整数（比如 2、4、20 ）有 int 类型，有小数部分的（比如 5.0、1.6 ）有 float 类型。在这个手册的后半部分我们会看到更多的数值类型。

The return type of a division (/) operation depends on its operands. If both operands are of type int, *floor division* is performed and an int is returned. If either operand is a float, classic division is performed and a float is returned. The // operator is also provided for doing floor division no matter what the operands are. The remainder can be calculated with the % operator:

```
>>> 17 / 3  # int / int -> int
5
>>> 17 / 3.0  # int / float -> float
5.666666666666667
>>> 17 // 3.0  # explicit floor division discards the fractional part
5.0
>>> 17 % 3  # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2  # result * divisor + remainder
17
```

在 Python 中，可以使用 ** 运算符来计算乘方[1]

```
>>> 5 ** 2  # 5 squared
25
>>> 2 ** 7  # 2 to the power of 7
128
```

等号 (=) 用于给一个变量赋值。然后在下一个交互提示符之前不会有结果显示出来:

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

如果一个变量未定义（未赋值），试图使用它时会向你提示错误:

```
>>> n  # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Python 中提供浮点数的完整支持；包含多种混合类型运算数的运算会把整数转换为浮点数:

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

在交互模式下，上一次打印出来的表达式被赋值给变量 _。这意味着当你把 Python 用作桌面计算器时，继续计算会相对简单，比如:

[1] 因为 ** 比 - 有更高的优先级, 所以 -3**2 会被解释成 -(3**2) ，因此结果是 -9. 为了避免这个并且得到结果 9, 你可以用这个式子 (-3)**2.

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

这个变量应该被使用者当作是只读类型。不要向它显式地赋值——你会创建一个和它名字相同独立的本地变量，它会使用魔法行为屏蔽内部变量。

除了 `int` 和 `float`，Python 也支持其他类型的数字，例如 `Decimal` 或者 `Fraction`。Python 也内置对 复数的支持，使用后缀 `j` 或者 `J` 就可以表示虚数部分（例如 `3+5j` ）。

3.1.2 字符串

除了数字，Python 也可以操作字符串。字符串有多种形式，可以使用单引号（`'...'`），双引号（`"..."`）都可以获得同样的结果[2]。反斜杠 `\` 可以用来转义:

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

In the interactive interpreter, the output string is enclosed in quotes and special characters are escaped with backslashes. While this might sometimes look different from the input (the enclosing quotes could change), the two strings are equivalent. The string is enclosed in double quotes if the string contains a single quote and no double quotes, otherwise it is enclosed in single quotes. The `print` statement produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
>>> print '"Isn\'t," they said.'
"Isn't," they said.
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print, \n is included in the output
'First line.\nSecond line.'
>>> print s  # with print, \n produces a new line
First line.
Second line.
```

如果你不希望前置了 `\` 的字符转义成特殊字符，可以使用 *原始字符串* 方式，在引号前添加 `r` 即可:

[2] 和其他语言不一样的是, 特殊字符比如说 `\n` 在单引号 (`'...'`) 和双引号 (`"..."`) 里有一样的意义. 这两种引号唯一的区别是，你不需要在单引号里转义双引号 `"` (但是你必须把单引号转义成 `\'`)，反之亦然.

```
>>> print 'C:\some\name'  # here \n means newline!
C:\some
ame
>>> print r'C:\some\name'  # note the r before the quote
C:\some\name
```

字符串字面值可以跨行连续输入。一种方式是用三重引号: `"""..."""` 或 `'''...'''`。字符串中的回车换行会自动包含到字符串中，如果不想包含，在行尾添加一个 \ 即可。如下例:

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

将产生如下输出（注意最开始的换行没有包括进来）:

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

字符串可以用 + 进行连接（粘到一起），也可以用 * 进行重复:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

相邻的两个或多个 *字符串字面值*（引号引起来的字符）将会自动连接到一起.

```
>>> 'Py' 'thon'
'Python'
```

把很长的字符串拆开分别输入的时候尤其有用:

```
>>> text = ('Put several strings within parentheses '
...         'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

只能对两个字面值这样操作，变量或表达式不行:

```
>>> prefix = 'Py'
>>> prefix 'thon'  # can't concatenate a variable and a string literal
  ...
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  ...
SyntaxError: invalid syntax
```

如果你想连接变量，或者连接变量和字面值，可以用 + 号:

```
>>> prefix + 'thon'
'Python'
```

字符串是可以被 *索引*（下标访问）的，第一个字符索引是 0。单个字符并没有特殊的类型，只是一个长度为一的字符串:

```
>>> word = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'
```

索引也可以用负数，这种会从右边开始数:

```
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
>>> word[-6]
'P'
```

注意 -0 和 0 是一样的，所以负数索引从 -1 开始。

In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, *slicing* allows you to obtain a substring:

```
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
```

注意切片的开始总是被包括在结果中，而结束不被包括。这使得 `s[:i] + s[i:]` 总是等于 `s`

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

切片的索引有默认值；省略开始索引时默认为 0，省略结束索引时默认为到字符串的结束:

```
>>> word[:2]   # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]   # characters from position 4 (included) to the end
'on'
>>> word[-2:]  # characters from the second-last (included) to the end
'on'
```

您也可以这么理解切片：将索引视作指向字符 之间，第一个字符的左侧标为 0，最后一个字符的右侧标为 *n* ，其中 *n* 是字符串长度。例如:

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

第一行数标注了字符串 0…6 的索引的位置，第二行标注了对应的负的索引。那么从 *i* 到 *j* 的切片就包括了标有 *i* 和 *j* 的位置之间的所有字符。

对于使用非负索引的切片，如果索引不越界，那么得到的切片长度就是起止索引之差。例如，`word[1:3]` 的长度为 2。

试图使用过大的索引会产生一个错误:

```
>>> word[42]  # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

但是，切片中的越界索引会被自动处理:

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Python 中的字符串不能被修改，它们是*immutable* 的。因此，向字符串的某个索引位置赋值会产生一个错误:

```
>>> word[0] = 'J'
  ...
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
  ...
TypeError: 'str' object does not support item assignment
```

如果需要一个不同的字符串，应当新建一个:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

内建函数 `len()` 返回一个字符串的长度:

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

参见:

typesseq Strings, and the Unicode strings described in the next section, are examples of *sequence types*, and support the common operations supported by such types.

string-methods Both strings and Unicode strings support a large number of methods for basic transformations and searching.

formatstrings 使用 `str.format()` 进行字符串格式化。

string-formatting The old formatting operations invoked when strings and Unicode strings are the left operand of the `%` operator are described in more detail here.

3.1.3 Unicode Strings

Starting with Python 2.0 a new data type for storing text data is available to the programmer: the Unicode object. It can be used to store and manipulate Unicode data (see http://www.unicode.org/) and integrates well with the existing string objects, providing auto-conversions where necessary.

Unicode has the advantage of providing one ordinal for every character in every script used in modern and ancient texts. Previously, there were only 256 possible ordinals for script characters. Texts were typically bound to a code page which mapped the ordinals to script characters. This lead to very much confusion especially with respect to internationalization (usually written as `i18n` —`'i'` + 18 characters + `'n'`) of software. Unicode solves these problems by defining one code page for all scripts.

Creating Unicode strings in Python is just as simple as creating normal strings:

```
>>> u'Hello World !'
u'Hello World !'
```

The small 'u' in front of the quote indicates that a Unicode string is supposed to be created. If you want to include special characters in the string, you can do so by using the Python *Unicode-Escape* encoding. The following example shows how:

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

The escape sequence \u0020 indicates to insert the Unicode character with the ordinal value 0x0020 (the space character) at the given position.

Other characters are interpreted by using their respective ordinal values directly as Unicode ordinals. If you have literal strings in the standard Latin-1 encoding that is used in many Western countries, you will find it convenient that the lower 256 characters of Unicode are the same as the 256 characters of Latin-1.

For experts, there is also a raw mode just like the one for normal strings. You have to prefix the opening quote with ‘ur’ to have Python use the *Raw-Unicode-Escape* encoding. It will only apply the above \uXXXX conversion if there is an uneven number of backslashes in front of the small ‘u’.

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

The raw mode is most useful when you have to enter lots of backslashes, as can be necessary in regular expressions.

Apart from these standard encodings, Python provides a whole set of other ways of creating Unicode strings on the basis of a known encoding.

The built-in function unicode() provides access to all registered Unicode codecs (COders and DECoders). Some of the more well known encodings which these codecs can convert are *Latin-1*, *ASCII*, *UTF-8*, and *UTF-16*. The latter two are variable-length encodings that store each Unicode character in one or more bytes. The default encoding is normally set to ASCII, which passes through characters in the range 0 to 127 and rejects any other characters with an error. When a Unicode string is printed, written to a file, or converted with str(), conversion takes place using this default encoding.

```
>>> u"abc"
u'abc'
>>> str(u"abc")
'abc'
>>> u"äöü"
u'\xe4\xf6\xfc'
>>> str(u"äöü")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-2: ordinal␣
↪not in range(128)
```

To convert a Unicode string into an 8-bit string using a specific encoding, Unicode objects provide an encode() method that takes one argument, the name of the encoding. Lowercase names for encodings are preferred.

```
>>> u"äöü".encode('utf-8')
'\xc3\xa4\xc3\xb6\xc3\xbc'
```

If you have data in a specific encoding and want to produce a corresponding Unicode string from it, you can use the unicode() function with the encoding name as the second argument.

```
>>> unicode('\xc3\xa4\xc3\xb6\xc3\xbc', 'utf-8')
u'\xe4\xf6\xfc'
```

3.1.4 列表

Python 中可以通过组合一些值得到多种 复合数据类型。其中最常用的 列表，可以通过方括号括起、逗号分隔的一组值（元素）得到。一个 列表可以包含不同类型的元素，但通常使用时各个元素类型相同:

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Like strings (and all other built-in *sequence* type), lists can be indexed and sliced:

```
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

所有的切片操作都返回一个新列表，这个新列表包含所需要的元素。就是说，如下的切片会返回列表的一个新的 (浅) 拷贝:

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

Lists also supports operations like concatenation:

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

与*immutable* 的字符串不同, 列表是一个*mutable* 类型，就是说，它自己的内容可以改变:

```
>>> cubes = [1, 8, 27, 65, 125]  # something's wrong here
>>> 4 ** 3  # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

你也可以在列表末尾通过 append() 方法来添加新元素（我们将在后面介绍有关方法的详情）：

```
>>> cubes.append(216)  # add the cube of 6
>>> cubes.append(7 ** 3)  # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

给切片赋值也是可以的，这样甚至可以改变列表大小，或者把列表整个清空:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
```

(下页继续)

(续上页)

```
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

内置函数 `len()` 也可以作用到列表上:

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

也可以嵌套列表 (创建包含其他列表的列表), 比如说:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3.2 走向编程的第一步

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the *Fibonacci* series as follows:

```
>>> # 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
```

这个例子引入了几个新的特性。

- 第一行含有一个 多重赋值: 变量 `a` 和 `b` 同时得到了新值 0 和 1. 最后一行又用了一次多重赋值, 这展示出了右手边的表达式，在任何赋值发生之前就被求值了。右手边的表达式是从左到右被求值的。
- The `while` loop executes as long as the condition (here: `b < 10`) remains true. In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything

with a non-zero length is true, empty sequences are false. The test used in the example is a simple comparison. The standard comparison operators are written the same as in C: < (less than), > (greater than), == (equal to), <= (less than or equal to), >= (greater than or equal to) and != (not equal to).

- 循环体是 缩进的：缩进是 Python 组织语句的方式。在交互式命令行里，你得给每个缩进的行敲下 Tab 键或者（多个）空格键。实际上用文本编辑器的话，你要准备更复杂的输入方式；所有像样的文本编辑器都有自动缩进的设置。交互式命令行里，当一个组合的语句输入时, 需要在最后敲一个空白行表示完成（因为语法分析器猜不出来你什么时候打的是最后一行）。注意，在同一块语句中的每一行，都要缩进相同的长度。
- The `print` statement writes the value of the expression(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple expressions and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely, like this:

```
>>> i = 256*256
>>> print 'The value of i is', i
The value of i is 65536
```

A trailing comma avoids the newline after the output:

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

Note that the interpreter inserts a newline before it prints the next prompt if the last line was not completed.

备注

CHAPTER 4

其他流程控制工具

除了刚刚介绍过的 `while` 语句，Python 中也会使用其他语言中常见的流程控制语句，只是稍有变化。

4.1 `if` Statements

可能最为人所熟知的编程语句就是 `if` 语句了。例如

```
>>> x = int(raw_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
```

There can be zero or more `elif` parts, and the `else` part is optional. The keyword ‘`elif`’ is short for ‘else if’, and is useful to avoid excessive indentation. An `if` …`elif` …`elif` …sequence is a substitute for the `switch` or `case` statements found in other languages.

4.2 `for` Statements

The `for` statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python’s `for` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

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

如果在循环内需要修改序列中的值（比如重复某些选中的元素），推荐你先拷贝一份副本。对序列进行循环不代表制作了一个副本进行操作。切片操作使这件事非常简单：

```
>>> for w in words[:]:  # Loop over a slice copy of the entire list.
...     if len(w) > 6:
...         words.insert(0, w)
...
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

4.3 `range()` 函数

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates lists containing arithmetic progressions:

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

The given end point is never part of the generated list; `range(10)` generates a list of 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ‘step’):

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

要以序列的索引来迭代，您可以将 `range()` 和 `len()` 组合如下:

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

(下页继续)

(续上页)

```
3 little
4 lamb
```

然而，在大多数这类情况下，使用 enumerate() 函数比较方便，请参见循环的技巧 。

4.4 break and continue Statements, and else Clauses on Loops

break 语句，和 C 中的类似，用于跳出最近的 for 或 while 循环.

Loop statements may have an else clause; it is executed when the loop terminates through exhaustion of the list (with for) or when the condition becomes false (with while), but not when the loop is terminated by a break statement. This is exemplified by the following loop, which searches for prime numbers:

```
>>> 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，这是正确的代码。仔细看：else 子句属于 for 循环，**不属于** if 语句。）

When used with a loop, the else clause has more in common with the else clause of a try statement than it does that of if statements: a try statement’s else clause runs when no exception occurs, and a loop’s else clause runs when no break occurs. For more on the try statement and exceptions, see 处理异常.

continue 语句也是借鉴自 C 语言，表示继续循环中的下一次迭代:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print "Found an even number", num
...         continue
...     print "Found a number", num
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

4.5 pass Statements

pass 语句什么也不做。当语法上需要一个语句，但程序需要什么动作也不做时，可以使用它。例如:

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

这通常用于创建最小的类:

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

Another place pass can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The pass is silently ignored:

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

4.6 定义函数

我们可以创建一个输出任意范围内 Fibonacci 数列的函数:

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print a,
...         a, b = b, a+b
...
>>> # 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 引入一个函数 定义。它必须后跟函数名称和带括号的形式参数列表。构成函数体的语句从下一行开始，并且必须缩进。

函数体的第一个语句可以（可选的）是字符串文字；这个字符串文字是函数的文档字符串或 *docstring* 。（有关文档字符串的更多信息，请参阅文档字符串 部分）有些工具使用文档字符串自动生成在线或印刷文档，或者让用户以交互式的形式浏览代码；在你编写的代码中包含文档字符串是一种很好的做法，所以要养成习惯。

The *execution* of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables cannot be directly assigned a value within a function (unless named in a global statement), although they may be referenced.

在函数被调用时，实际参数（实参）会被引入被调用函数的本地符号表中；因此，实参是通过 按值调用 传递的（其中 值始终是对象 引用 而不是对象的值）。[1] 当一个函数调用另外一个函数时，将会为该调用创建一个新的本地符号表。

[1] 实际上，通过对象引用调用会是一个更好的表述，因为如果传递的是可变对象，则调用者将看到被调用者对其做出的任何更改（插入到列表中的元素）。

函数定义会把函数名引入当前的符号表中。函数名称的值具有解释器将其识别为用户定义函数的类型。这个值可以分配给另一个名称，该名称也可以作为一个函数使用。这用作一般的重命名机制:

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

Coming from other languages, you might object that `fib` is not a function but a procedure since it doesn’t return a value. In fact, even functions without a `return` statement do return a value, albeit a rather boring one. This value is called `None` (it’s a built-in name). Writing the value `None` is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using `print`:

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

写一个返回斐波那契数列的列表（而不是把它打印出来）的函数，非常简单:

```
>>> 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]
```

此示例中，像往常一样，演示了一些新的 Python 功能:

- The `return` statement returns with a value from a function. `return` without an expression argument returns `None`. Falling off the end of a function also returns `None`.
- `result.append(a)` 语句调用了列表对象 `result` 的 *方法*。方法是“属于”一个对象的函数，它被命名为 `obj.methodname` ，其中 `obj` 是某个对象（也可能是一个表达式），`methodname` 是由对象类型中定义的方法的名称。不同的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而不会引起歧义。（可以使用 类定义自己的对象类型和方法，请参阅类 ）示例中的方法 `append()` 是为列表对象定义的；它会在列表的最后添加一个新的元素。在这个示例中它相当于 `result = result + [a]` ，但更高效。

4.7 函数定义的更多形式

给函数定义有可变数目的参数也是可行的。这里有三种形式，可以组合使用。

4.7.1 参数默认值

最有用的形式是对一个或多个参数指定一个默认值。这样创建的函数，可以用比定义时允许的更少的参数调用，比如:

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = raw_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
```

这个函数可以通过几种方式调用:

- 只给出必需的参数: `ask_ok('Do you really want to quit?')`
- 给出一个可选的参数: `ask_ok('OK to overwrite the file?', 2)`
- 或者给出所有的参数: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

这个示例还介绍了 `in` 关键字。它可以测试一个序列是否包含某个值。

默认值是在 *定义过程* 中在函数定义处计算的，所以

```
i = 5

def f(arg=i):
    print arg

i = 6
f()
```

会打印 `5`。

重要警告: 默认值只会执行一次。这条规则在默认值为可变对象（列表、字典以及大多数类实例）时很重要。比如，下面的函数会存储在后续调用中传递给它的参数:

```
def f(a, L=[]):
    L.append(a)
    return L

print f(1)
print f(2)
print f(3)
```

这将打印出

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

如果你不想要在后续调用之间共享默认值，你可以这样写这个函数:

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

4.7.2 关键字参数

也可以使用形如 kwarg=value 的关键字参数 来调用函数。例如下面的函数:

```
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, "!"
```

接受一个必需的参数 (voltage) 和三个可选的参数 (state, action, 和 type)。这个函数可以通过下面的任何一种方式调用:

```
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
```

但下面的函数调用都是无效的:

```
parrot()                     # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword argument
parrot(110, voltage=220)     # duplicate value for the same argument
parrot(actor='John Cleese')  # unknown keyword argument
```

在函数调用中，关键字参数必须跟随在位置参数的后面。传递的所有关键字参数必须与函数接受的其中一个参数匹配 (比如 actor 不是函数 parrot 的有效参数)，它们的顺序并不重要。这也包括非可选参数，(比如 parrot(voltage=1000) 也是有效的)。不能对同一个参数多次赋值。下面是一个因为此限制而失败的例子:

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

When a final formal parameter of the form **name is present, it receives a dictionary (see typesmapping) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form *name (described in the next subsection) which receives a tuple containing the positional arguments beyond the formal parameter list. (*name must occur before **name.) For example, if we define a function like this:

```
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:
```

(下页继续)

(续上页)

```
        print arg
    print "-" * 40
    keys = sorted(keywords.keys())
    for kw in keys:
        print kw, ":", keywords[kw]
```

它可以像这样调用:

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

当然它会打印:

```
-- 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
```

Note that the list of keyword argument names is created by sorting the result of the keywords dictionary’s `keys()` method before printing its contents; if this is not done, the order in which the arguments are printed is undefined.

4.7.3 任意的参数列表

最后，最不常用的选项是可以使用任意数量的参数调用函数。这些参数会被包含在一个元组里（参见元组和序列)。在可变数量的参数之前，可能会出现零个或多个普通参数。:

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

4.7.4 解包参数列表

The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. For instance, the built-in `range()` function expects separate *start* and *stop* arguments. If they are not available separately, write the function call with the `*`-operator to unpack the arguments out of a list or tuple:

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

In the same fashion, dictionaries can deliver keyword arguments with the `**`-operator:

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

4.7.5 Lambda 表达式

可以用 lambda 关键字来创建一个小的匿名函数。这个函数返回两个参数的和：lambda a, b: a+b 。Lambda 函数可以在需要函数对象的任何地方使用。它们在语法上限于单个表达式。从语义上来说，它们只是正常函数定义的语法糖。与嵌套函数定义一样，lambda 函数可以引用所包含域的变量:

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

上面的例子使用一个 lambda 表达式来返回一个函数。另一个用法是传递一个小函数作为参数:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

4.7.6 文档字符串

There are emerging conventions about the content and formatting of documentation strings.

第一行应该是对象目的的简要概述。为简洁起见，它不应显式声明对象的名称或类型，因为这些可通过其他方式获得（除非名称恰好是描述函数操作的动词）。这一行应以大写字母开头，以句点结尾。

如果文档字符串中有更多行，则第二行应为空白，从而在视觉上将摘要与其余描述分开。后面几行应该是一个或多个段落，描述对象的调用约定，它的副作用等。

Python 解析器不会从 Python 中删除多行字符串文字的缩进，因此处理文档的工具必须在需要时删除缩进。这是使用以下约定完成的。文档字符串第一行 *之后* 的第一个非空行确定整个文档字符串的缩进量。（我们不能使用第一行，因为它通常与字符串的开头引号相邻，因此它的缩进在字符串文字中不明显。）然后从字符串的所有行的开头剥离与该缩进 "等效" 的空格。缩进更少的行不应该出现，但是如果它们出现，则应该剥离它们的所有前导空格。应在转化制表符为空格后测试空格的等效性（通常转化为 8 个空格）。

下面是一个多行文档字符串的例子:

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

(下页继续)

(续上页)

```
...     """
...     pass
...
>>> print my_function.__doc__
Do nothing, but document it.

    No, really, it doesn't do anything.
```

4.8 小插曲：编码风格

现在你将要写更长，更复杂的 Python 代码，是时候讨论一下 代码风格了。大多数语言都能以不同的风格被编写（或更准确地说，被格式化）；有些比其他的更具有可读性。能让其他人轻松阅读你的代码总是一个好主意，采用一种好的编码风格对此有很大帮助。

对于 Python，**PEP 8** 已经成为大多数项目所遵循的风格指南；它促进了一种非常易读且令人赏心悦目的编码风格。每个 Python 开发人员都应该在某个时候阅读它；以下是为你提取的最重要的几个要点：

- 使用 4 个空格缩进，不要使用制表符。

  4 个空格是一个在小缩进（允许更大的嵌套深度）和大缩进（更容易阅读）的一种很好的折中方案。制表符会引入混乱，最好不要使用它。

- 换行，使一行不超过 79 个字符。

  这有助于使用小型显示器的用户，并且可以在较大的显示器上并排放置多个代码文件。

- 使用空行分隔函数和类，以及函数内的较大的代码块。
- 如果可能，把注释放到单独的一行。
- 使用文档字符串。
- 在运算符前后和逗号后使用空格，但不能直接在括号内使用：`a = f(1, 2) + g(3, 4)`。
- Name your classes and functions consistently; the convention is to use `CamelCase` for classes and `lower_case_with_underscores` for functions and methods. Always use `self` as the name for the first method argument (see 初探类 for more on classes and methods).
- Don’t use fancy encodings if your code is meant to be used in international environments. Plain ASCII works best in any case.

备注

CHAPTER 5

数据结构

本章节将详细介绍一些您已经了解的内容，并添加了一些新内容。

5.1 列表的更多特性

列表数据类型还有很多的方法。这里是列表对象方法的清单：

`list.append(x)`
: Add an item to the end of the list; equivalent to `a[len(a):] = [x]`.

`list.extend(L)`
: Extend the list by appending all the items in the given list; equivalent to `a[len(a):] = L`.

`list.insert(i, x)`
: 在给定的位置插入一个元素。第一个参数是要插入的元素的索引，所以 `a.insert(0, x)` 插入列表头部，`a.insert(len(a), x)` 等同于 `a.append(x)` 。

`list.remove(x)`
: Remove the first item from the list whose value is *x*. It is an error if there is no such item.

`list.pop([i])`
: 删除列表中给定位置的元素并返回它。如果没有给定位置，`a.pop()` 将会删除并返回列表中的最后一个元素。（方法签名中 *i* 两边的方括号表示这个参数是可选的，而不是要你输入方括号。你会在 Python 参考库中经常看到这种表示方法)。

`list.index(x)`
: Return the index in the list of the first item whose value is *x*. It is an error if there is no such item.

`list.count(x)`
: 返回元素 *x* 在列表中出现的次数。

`list.sort(cmp=None, key=None, reverse=False)`
: 对列表中的元素进行排序（参数可用于自定义排序，解释请参见 `sorted()`）。

`list.reverse()`
: Reverse the elements of the list, in place.

多数列表方法示例：

```
>>> 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)
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]
>>> a.pop()
1234.5
>>> a
[-1, 1, 66.25, 333, 333]
```

You might have noticed that methods like `insert`, `remove` or `sort` that only modify the list have no return value printed –they return the default `None`. This is a design principle for all mutable data structures in Python.

5.1.1 列表作为栈使用

列表方法使得列表作为堆栈非常容易，最后一个插入，最先取出（"后进先出"）。要添加一个元素到堆栈的顶端，使用 `append()` 。要从堆栈顶部取出一个元素，使用 `pop()` ，不用指定索引。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]
```

5.1.2 列表作为队列使用

列表也可以用作队列，其中先添加的元素被最先取出 (“先进先出”)；然而列表用作这个目的相当低效。因为在列表的末尾添加和弹出元素非常快，但是在列表的开头插入或弹出元素却很慢 (因为所有的其他元素都必须移动一位)。

若要实现一个队列，可使用 `collections.deque`，它被设计成可以快速地从两端添加或弹出元素。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
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3 Functional Programming Tools

There are three built-in functions that are very useful when used with lists: `filter()`, `map()`, and `reduce()`.

`filter(function, sequence)` returns a sequence consisting of those items from the sequence for which `function(item)` is true. If *sequence* is a `str`, `unicode` or `tuple`, the result will be of the same type; otherwise, it is always a `list`. For example, to compute a sequence of numbers divisible by 3 or 5:

```
>>> def f(x): return x % 3 == 0 or x % 5 == 0
...
>>> filter(f, range(2, 25))
[3, 5, 6, 9, 10, 12, 15, 18, 20, 21, 24]
```

`map(function, sequence)` calls `function(item)` for each of the sequence’s items and returns a list of the return values. For example, to compute some cubes:

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

More than one sequence may be passed; the function must then have as many arguments as there are sequences and is called with the corresponding item from each sequence (or `None` if some sequence is shorter than another). For example:

```
>>> seq = range(8)
>>> def add(x, y): return x+y
...
>>> map(add, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

`reduce(function, sequence)` returns a single value constructed by calling the binary function *function* on the first two items of the sequence, then on the result and the next item, and so on. For example, to compute the sum of the numbers 1 through 10:

```
>>> def add(x,y): return x+y
...
```

(下页继续)

(续上页)

```
>>> reduce(add, range(1, 11))
55
```

If there' s only one item in the sequence, its value is returned; if the sequence is empty, an exception is raised.

A third argument can be passed to indicate the starting value. In this case the starting value is returned for an empty sequence, and the function is first applied to the starting value and the first sequence item, then to the result and the next item, and so on. For example,

```
>>> def sum(seq):
...     def add(x,y): return x+y
...     return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```

Don' t use this example' s definition of `sum()`: since summing numbers is such a common need, a built-in function `sum(sequence)` is already provided, and works exactly like this.

5.1.4 列表推导式

列表推导式提供了一个更简单的创建列表的方法。常见的用法是把某种操作应用于序列或可迭代对象的每个元素上，然后使用其结果来创建列表，或者通过满足某些特定条件元素来创建子序列。

例如，假设我们想创建一个平方列表，像这样

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

We can obtain the same result with:

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

This is also equivalent to `squares = map(lambda x: x**2, range(10))`, but it' s more concise and readable.

A list comprehension consists of brackets containing an expression followed by a `for` clause, then zero or more `for` or `if` clauses. The result will be a new list resulting from evaluating the expression in the context of the `for` and `if` clauses which follow it. For example, this listcomp combines the elements of two lists if they are not equal:

```
>>> [(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)]
```

and it' s equivalent to:

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

(下页继续)

(续上页)

```
...                combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

注意在上面两个代码片段中，for 和 if 的顺序是相同的。

如果表达式是一个元组（例如上面的 (x, y)），那么就必须加上括号

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(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
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1, in <module>
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
>>> 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]
```

列表推导式可以使用复杂的表达式和嵌套函数

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

嵌套的列表推导式

列表推导式中的初始表达式可以是任何表达式，包括另一个列表推导式。

考虑下面这个 3x4 的矩阵，它由 3 个长度为 4 的列表组成

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

下面的列表推导式将交换其行和列

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

如上节所示，嵌套的列表推导式是基于跟随其后的 for 进行求值的，所以这个例子等价于:

```
>>> 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]]
```

反过来说，也等价于

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     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]]
```

实际应用中，你应该会更喜欢使用内置函数去组成复杂的流程语句。zip() 函数将会很好地处理这种情况

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

关于本行中星号的详细说明，参见解包参数列表。

5.2 The del statement

There is a way to remove an item from a list given its index instead of its value: the del statement. This differs from the pop() method which returns a value. The del statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

```
>>> 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 a 时会报错（直到另一个值被赋给它）。我们会在后面了解到 del 的其他用法。

5.3 元组和序列

我们看到列表和字符串有很多共同特性，例如索引和切片操作。他们是 序列数据类型（参见 typesseq）中的两种。随着 Python 语言的发展，其他的序列类型也会被加入其中。这里介绍另一种标准序列类型: 元组。

一个元组由几个被逗号隔开的值组成，例如

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

如你所见，元组在输出时总是被圆括号包围的，以便正确表示嵌套元组。输入时圆括号可有可无，不过经常会是必须的（如果这个元组是一个更大的表达式的一部分）。给元组中的一个单独的元素赋值是不允许的，当然你可以创建包含可变对象的元组，例如列表。

虽然元组可能看起来与列表很像，但它们通常是在不同的场景被使用，并且有着不同的用途。元组是*immutable* ，其序列通常包含不同种类的元素，并且通过解包（这一节下面会解释）或者索引来访问（如果是 `namedtuples` 的话甚至还可以通过属性访问）。列表是*mutable* ，并且列表中的元素一般是同种类型的，并且通过迭代访问。

一个特殊的问题是构造包含 0 个或 1 个元素的元组：为了适应这种情况，语法有一些额外的改变。空元组可以直接被一对空圆括号创建，含有一个元素的元组可以通过在这个元素后添加一个逗号来构建（圆括号里只有一个值的话不够明确）。丑陋，但是有效。例如

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

语句 `t = 12345, 54321, 'hello!'` 是 元组打包的一个例子：值 `12345`, `54321` 和 `'hello!'` 被打包进元组。其逆操作也是允许的

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

This is called, appropriately enough, *sequence unpacking* and works for any sequence on the right-hand side. Sequence unpacking requires the list of variables on the left to have the same number of elements as the length of the sequence. Note that multiple assignment is really just a combination of tuple packing and sequence unpacking.

5.4 集合

Python 也包含有 集合类型。集合是由不重复元素组成的无序的集。它的基本用法包括成员检测和消除重复元素。集合对象也支持像联合，交集，差集，对称差分等数学运算。

花括号或 `set()` 函数可以用来创建集合。注意：要创建一个空集合你只能用 `set()` 而不能用 `{}`，因为后者是创建一个空字典，这种数据结构我们会在下一节进行讨论。

以下是一些简单的示例

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruit = set(basket)               # create a set without duplicates
>>> fruit
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruit                 # fast membership testing
True
>>> 'crabgrass' in fruit
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> 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'])
```

类似于列表推导式，集合也支持推导式形式

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
set(['r', 'd'])
```

5.5 字典

另一个非常有用的 Python 内置数据类型是 字典 (参见 typesmapping)。字典在其他语言里可能会被叫做 联合内存或 联合数组。与以连续整数为索引的序列不同，字典是以 关键字为索引的，关键字可以是任意不可变类型，通常是字符串或数字。如果一个元组只包含字符串、数字或元组，那么这个元组也可以用作关键字。但如果元组直接或间接地包含了可变对象，那么它就不能用作关键字。列表不能用作关键字，因为列表可以通过索引、切片或 `append()` 和 `extend()` 之类的方法来改变。

It is best to think of a dictionary as an unordered set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}`. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

字典主要的操作是使用关键字存储和解析值。也可以用 `del` 来删除一个键值对。如果你使用了一个已经存在的关键字来存储值，那么之前与这个关键字关联的值就会被遗忘。用一个不存在的键来取值则会报错。

The `keys()` method of a dictionary object returns a list of all the keys used in the dictionary, in arbitrary order (if you want it sorted, just apply the `sorted()` function to it). To check whether a single key is in the dictionary, use the `in` keyword.

以下是使用字典的一些简单示例

```
>>> 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
```

`dict()` 构造函数可以直接从键值对序列里创建字典。

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

此外，字典推导式可以从任意的键值表达式中创建字典

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

当关键字是简单字符串时，有时直接通过关键字参数来指定键值对更方便

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

5.6 循环的技巧

当在序列中循环时，用 `enumerate()` 函数可以将索引位置和其对应的值同时取出

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

当同时在两个或更多序列中循环时，可以用 `zip()` 函数将其内元素一一匹配。

```
>>> 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.
```

(下页继续)

(续上页)

```
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

如果要逆向循环一个序列，可以先正向定位序列，然后调用 `reversed()` 函数

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

如果要按某个指定顺序循环一个序列，可以用 `sorted()` 函数，它可以在不改动原序列的基础上返回一个新的排好序的序列

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

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the `iteritems()` method.

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

有时可能会想在循环时修改列表内容，一般来说改为创建一个新列表是比较简单且安全的

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

5.7 深入条件控制

while 和 if 条件句中可以使用任意操作，而不仅仅是比较操作。

比较操作符 in 和 not in 校验一个值是否在（或不在）一个序列里。操作符 is 和 is not 比较两个对象是不是同一个对象，这只对像列表这样的可变对象比较重要。所有的比较操作符都有相同的优先级，且这个优先级比数值运算符低。

比较操作可以传递。例如 a < b == c 会校验是否 a 小于 b 并且 b 等于 c。

比较操作可以通过布尔运算符 and 和 or 来组合，并且比较操作（或其他任何布尔运算）的结果都可以用 not 来取反。这些操作符的优先级低于比较操作符；在它们之中，not 优先级最高，or 优先级最低，因此 A and not B or C 等价于 (A and (not B)) or C。和之前一样，你也可以在这种式子里使用圆括号。

布尔运算符 and 和 or 也被称为 短路运算符：它们的参数从左至右解析，一旦可以确定结果解析就会停止。例如，如果 A 和 C 为真而 B 为假，那么 A and B and C 不会解析 C。当用作普通值而非布尔值时，短路操作符的返回值通常是最后一个变量。

也可以把比较操作或者逻辑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 Python 与 C 不同，赋值操作不能发生在表达式内部。C 程序员可能会对此抱怨，但它避免了一类 C 程序中常见的错误：想在表达式中写 == 时却写成了 =。

5.8 比较序列和其他类型

Sequence objects may be compared to other objects with the same sequence type. The comparison uses *lexicographical* ordering: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. If two items to be compared are themselves sequences of the same type, the lexicographical comparison is carried out recursively. If all items of two sequences compare equal, the sequences are considered equal. If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one. Lexicographical ordering for strings uses the ASCII ordering for individual characters. Some examples of comparisons between sequences of the same type:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Note that comparing objects of different types is legal. The outcome is deterministic but arbitrary: the types are ordered by their name. Thus, a list is always smaller than a string, a string is always smaller than a tuple, etc.[1] Mixed numeric types are compared according to their numeric value, so 0 equals 0.0, etc.

[1] The rules for comparing objects of different types should not be relied upon; they may change in a future version of the language.

备注

CHAPTER 6

模块

如果你从 Python 解释器退出并再次进入，之前的定义（函数和变量）都会丢失。因此，如果你想编写一个稍长些的程序，最好使用文本编辑器为解释器准备输入并将该文件作为输入运行。这被称作编写 *脚本*。随着程序变得越来越长，你或许会想把它拆分成几个文件，以方便维护。你亦或想在不同的程序中使用一个便捷的函数，而不必把这个函数复制到每一个程序中去。

为支持这些，Python 有一种方法可以把定义放在一个文件里，并在脚本或解释器的交互式实例中使用它们。这样的文件被称作 *模块*；模块中的定义可以 *导入*到其它模块或者 *主*模块（你在顶级和计算器模式下执行的脚本中可以访问的变量集合）。

模块是一个包含 Python 定义和语句的文件。文件名就是模块名后跟文件后缀 `.py` 。在一个模块内部，模块名（作为一个字符串）可以通过全局变量 `__name__` 的值获得。例如，使用你最喜爱的文本编辑器在当前目录下创建一个名为 `fibo.py` 的文件，文件中含有以下内容:

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

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，并用以下命令导入该模块:

```
>>> import fibo
```

在当前的符号表中，这并不会直接进入到定义在 `fibo` 函数内的名称；它只是进入到模块名 `fibo` 中。你可以用模块名访问这些函数:

```
>>> 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'
```

如果你想经常使用某个函数，你可以把它赋值给一个局部变量:

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

6.1 更多有关模块的信息

模块可以包含可执行的语句以及函数定义。这些语句用于初始化模块。它们仅在模块 第一次在 import 语句中被导入时才执行。[1] (当文件被当作脚本运行时，它们也会执行。)

每个模块都有它自己的私有符号表，该表用作模块中定义的所有函数的全局符号表。因此，模块的作者可以在模块内使用全局变量，而不必担心与用户的全局变量发生意外冲突。另一方面，如果你知道自己在做什么，则可以用跟访问模块内的函数的同样标记方法，去访问一个模块的全局变量，`modname.itemname`。

模块可以导入其它模块。习惯上但不要求把所有 `import` 语句放在模块（或脚本）的开头。被导入的模块名存放在调入模块的全局符号表中。

`import` 语句有一个变体，它可以把名字从一个被调模块内直接导入到现模块的符号表里。例如:

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

这并不会把被调模块名引入到局部变量表里（因此在这个例子里，`fibo` 是未被定义的）。

还有一个变体甚至可以导入模块内定义的所有名称:

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

This imports all names except those beginning with an underscore (_).

注意通常情况下从一个模块或者包内调入 `*` 的做法是不太被接受的，因为这通常会导致代码的可读性很差。不过，在交互式编译器中为了节省打字可以这么用。

If the module name is followed by `as`, then the name following `as` is bound directly to the imported module.

```
>>> import fibo as fib
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这会和 `import fibo` 方式一样有效地调入模块，唯一的区别是它以 `fib` 的名称存在的。

这种方式也可以在用到 `from` 的时候使用，并会有类似的效果:

[1] 实际上，函数定义也是 “被执行” 的 “语句”；模块级函数定义的执行在模块的全局符号表中输入该函数名。

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

注解： For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter –or, if it’ s just one module you want to test interactively, use `reload()`, e.g. `reload(modulename)`.

6.1.1 以脚本的方式执行模块

当你用下面方式运行一个 Python 模块:

```
python fibo.py <arguments>
```

模块里的代码会被执行，就好像你导入了模块一样，但是 `__name__` 被赋值为 `"__main__"`。这意味着通过在你的模块末尾添加这些代码:

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

你既可以把这个文件当作脚本又可当作一个可调入的模块来使用，因为那段解析命令行的代码只有在当模块是以 “main” 文件的方式执行的时候才会运行:

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

如果模块是被导入的，那些代码是不运行的:

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

这经常用于为模块提供一个方便的用户接口，或用于测试（以脚本的方式运行模块从而执行一些测试套件）。

6.1.2 模块搜索路径

当一个名为 `spam` 的模块被导入的时候，解释器首先寻找具有该名称的内置模块。如果没有找到，然后解释器从 `sys.path` 变量给出的目录列表里寻找名为 `spam.py` 的文件。`sys.path` 初始有这些目录地址:

- the directory containing the input script (or the current directory).
- `PYTHONPATH` （一个包含目录名称的列表，它和 shell 变量 `PATH` 有一样的语法）。
- the installation-dependent default.

在初始化后，Python 程序可以更改 `sys.path`。包含正在运行脚本的文件目录被放在搜索路径的开头处，在标准库路径之前。这意味着将加载此目录里的脚本，而不是标准库中的同名模块。除非有意更换，否则这是错误。更多信息请参阅标准模块。

6.1.3 "编译过的" Python 文件

As an important speed-up of the start-up time for short programs that use a lot of standard modules, if a file called `spam.pyc` exists in the directory where `spam.py` is found, this is assumed to contain an already- "byte-compiled" version of the module `spam`. The modification time of the version of `spam.py` used to create `spam.pyc` is recorded in `spam.pyc`, and the `.pyc` file is ignored if these don' t match.

Normally, you don' t need to do anything to create the `spam.pyc` file. Whenever `spam.py` is successfully compiled, an attempt is made to write the compiled version to `spam.pyc`. It is not an error if this attempt fails; if for any reason the file is not written completely, the resulting `spam.pyc` file will be recognized as invalid and thus ignored later. The contents of the `spam.pyc` file are platform independent, so a Python module directory can be shared by machines of different architectures.

给专业人士的一些小建议:

- When the Python interpreter is invoked with the `-O` flag, optimized code is generated and stored in `.pyo` files. The optimizer currently doesn' t help much; it only removes `assert` statements. When `-O` is used, *all* *bytecode* is optimized; `.pyc` files are ignored and `.py` files are compiled to optimized bytecode.
- Passing two `-O` flags to the Python interpreter (`-OO`) will cause the bytecode compiler to perform optimizations that could in some rare cases result in malfunctioning programs. Currently only `__doc__` strings are removed from the bytecode, resulting in more compact `.pyo` files. Since some programs may rely on having these available, you should only use this option if you know what you' re doing.
- A program doesn' t run any faster when it is read from a `.pyc` or `.pyo` file than when it is read from a `.py` file; the only thing that' s faster about `.pyc` or `.pyo` files is the speed with which they are loaded.
- When a script is run by giving its name on the command line, the bytecode for the script is never written to a `.pyc` or `.pyo` file. Thus, the startup time of a script may be reduced by moving most of its code to a module and having a small bootstrap script that imports that module. It is also possible to name a `.pyc` or `.pyo` file directly on the command line.
- It is possible to have a file called `spam.pyc` (or `spam.pyo` when `-O` is used) without a file `spam.py` for the same module. This can be used to distribute a library of Python code in a form that is moderately hard to reverse engineer.
- The module `compileall` can create `.pyc` files (or `.pyo` files when `-O` is used) for all modules in a directory.

6.2 标准模块

Python 附带了一个标准模块库，在单独的文档 Python 库参考（以下称为 "库参考"）中进行了描述。一些模块内置于解释器中；它们提供对不属于语言核心但仍然内置的操作的访问，以提高效率或提供对系统调用等操作系统原语的访问。这些模块的集合是一个配置选项，它也取决于底层平台。例如，`winreg` 模块只在 Windows 操作系统上提供。一个特别值得注意的模块 `sys`，它被内嵌到每一个 Python 解释器中。变量 `sys.ps1` 和 `sys.ps2` 定义用作主要和辅助提示的字符串:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print 'Yuck!'
Yuck!
C>
```

这两个变量只有在编译器是交互模式下才被定义。

sys.path 变量是一个字符串列表，用于确定解释器的模块搜索路径。该变量被初始化为从环境变量 PYTHONPATH 获取的默认路径，或者如果 PYTHONPATH 未设置，则从内置默认路径初始化。你可以使用标准列表操作对其进行修改:

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

6.3 dir() 函数

内置函数 dir() 用于查找模块定义的名称。它返回一个排序过的字符串列表:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__',
 '__stderr__', '__stdin__', '__stdout__', '_clear_type_cache',
 '_current_frames', '_getframe', '_mercurial', 'api_version', 'argv',
 'builtin_module_names', 'byteorder', 'call_tracing', 'callstats',
 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_clear', 'exc_info',
 'exc_traceback', 'exc_type', 'exc_value', 'excepthook', 'exec_prefix',
 'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
 'getrefcount', 'getsizeof', 'gettotalrefcount', 'gettrace', 'hexversion',
 'long_info', 'maxint', 'maxsize', 'maxunicode', 'meta_path', 'modules',
 'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
 'py3kwarning', 'setcheckinterval', 'setdlopenflags', 'setprofile',
 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'subversion',
 'version', 'version_info', 'warnoptions']
```

如果没有参数，dir() 会列出你当前定义的名称:

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

注意：它列出所有类型的名称：变量，模块，函数，等等。

dir() does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module __builtin__:

```
>>> import __builtin__
>>> dir(__builtin__)
['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',
```

(下页继续)

(续上页)

```
'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError',
'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True',
'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning',
'ZeroDivisionError', '_', '__debug__', '__doc__', '__import__',
'__name__', '__package__', 'abs', 'all', 'any', 'apply', 'basestring',
'bin', 'bool', 'buffer', 'bytearray', 'bytes', 'callable', 'chr',
'classmethod', 'cmp', 'coerce', 'compile', 'complex', 'copyright',
'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval',
'execfile', 'exit', 'file', 'filter', 'float', 'format', 'frozenset',
'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input',
'int', 'intern', 'isinstance', 'issubclass', 'iter', 'len', 'license',
'list', 'locals', 'long', 'map', 'max', 'memoryview', 'min', 'next',
'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit',
'range', 'raw_input', 'reduce', 'reload', 'repr', 'reversed', 'round',
'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super',
'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

6.4 包

包是一种通过用 “带点号的模块名” 来构造 Python 模块命名空间的方法。例如，模块名 `A.B` 表示 `A` 包中名为 `B` 的子模块。正如模块的使用使得不同模块的作者不必担心彼此的全局变量名称一样，使用加点的模块名可以使得 NumPy 或 Pillow 等多模块软件包的作者不必担心彼此的模块名称一样。

假设你想为声音文件和声音数据的统一处理，设计一个模块集合（一个 “包”）。由于存在很多不同的声音文件格式（通常由它们的扩展名来识别，例如：`.wav`，`.aiff`，`.au`），因此为了不同文件格式间的转换，你可能需要创建和维护一个不断增长的模块集合。你可能还想对声音数据还做很多不同的处理（例如，混声，添加回声，使用均衡器功能，创造人工立体声效果），因此为了实现这些处理，你将另外写一个无穷尽的模块流。这是你的包的可能结构（以分层文件系统的形式表示）：

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

当导入这个包时，Python 搜索 sys.path 里的目录，查找包的子目录。

The __init__.py files are required to make Python treat the directories as containing packages; this is done to prevent directories with a common name, such as string, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, __init__.py can just be an empty file, but it can also execute initialization code for the package or set the __all__ variable, described later.

包的用户可以从包中导入单个模块，例如:

```
import sound.effects.echo
```

这会加载子模块 sound.effects.echo 。但引用它时必须使用它的全名。

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

导入子模块的另一种方法是

```
from sound.effects import echo
```

这也会加载子模块 echo ，并使其在没有包前缀的情况下可用，因此可以按如下方式使用:

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

另一种形式是直接导入所需的函数或变量:

```
from sound.effects.echo import echofilter
```

同样，这也会加载子模块 echo，但这会使其函数 echofilter() 直接可用:

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

请注意，当使用 from package import item 时，item 可以是包的子模块（或子包），也可以是包中定义的其他名称，如函数，类或变量。import 语句首先测试是否在包中定义了 item；如果没有，它假定它是一个模块并尝试加载它。如果找不到它，则引发 ImportError 异常。

相反，当使用 import item.subitem.subsubitem 这样的语法时，除了最后一项之外的每一项都必须是一个包；最后一项可以是模块或包，但不能是前一项中定义的类或函数或变量。

6.4.1 从包中导入 *

当用户写 from sound.effects import * 会发生什么？理想情况下，人们希望这会以某种方式传递给文件系统，找到包中存在哪些子模块，并将它们全部导入。这可能需要很长时间，导入子模块可能会产生不必要的副作用，这种副作用只有在显式导入子模块时才会发生。

唯一的解决方案是让包作者提供一个包的显式索引。import 语句使用下面的规范：如果一个包的 __init__.py 代码定义了一个名为 __all__ 的列表，它会被视为在遇到 from package import * 时应该导入的模块名列表。在发布该包的新版本时，包作者可以决定是否让此列表保持更新。包作者如果认为从他们的包中导入 * 的操作没有必要被使用，也可以决定不支持此列表。例如，文件 sound/effects/__init__.py 可以包含以下代码:

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

这意味着 from sound.effects import * 将导入 sound 包的三个命名子模块。

如果没有定义 __all__，from sound.effects import * 语句 不会从包 sound.effects 中导入所有子模块到当前命名空间；它只确保导入了包 sound.effects （可能运行任何在 __init__.py 中的初始化代码），然后导入包中定义的任何名称。这包括 __init__.py` 定义的任何名称（以及显式加载的子模块）。它还包括由之前的 import 语句显式加载的包的任何子模块。思考下面的代码:

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

在这个例子中，echo 和 surround 模块是在执行 from...import 语句时导入到当前命名空间中的，因为它们定义在 sound.effects 包中。(这在定义了 __all__ 时也有效。)

虽然某些模块被设计为在使用 import * 时只导出遵循某些模式的名称，但在生产代码中它仍然被认为是不好的做法。

请记住，使用 from package import specific_submodule 没有任何问题！实际上，除非导入的模块需要使用来自不同包的同名子模块，否则这是推荐的表示法。

6.4.2 子包参考

The submodules often need to refer to each other. For example, the surround module might use the echo module. In fact, such references are so common that the import statement first looks in the containing package before looking in the standard module search path. Thus, the surround module can simply use import echo or from echo import echofilter. If the imported module is not found in the current package (the package of which the current module is a submodule), the import statement looks for a top-level module with the given name.

当包被构造成子包时（与示例中的 sound 包一样），你可以使用绝对导入来引用兄弟包的子模块。例如，如果模块 sound.filters.vocoder 需要在 sound.effects 包中使用 echo 模块，它可以使用 from sound.effects import echo 。

Starting with Python 2.5, in addition to the implicit relative imports described above, you can write explicit relative imports with the from module import name form of import statement. These explicit relative imports use leading dots to indicate the current and parent packages involved in the relative import. From the surround module for example, you might use:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Note that both explicit and implicit relative imports are based on the name of the current module. Since the name of the main module is always "__main__", modules intended for use as the main module of a Python application should always use absolute imports.

6.4.3 多个目录中的包

包支持另一个特殊属性，__path__ 。它被初始化为一个列表，其中包含在执行该文件中的代码之前保存包的文件 __init__.py 的目录的名称。这个变量可以修改；这样做会影响将来对包中包含的模块和子包的搜索。

虽然通常不需要此功能，但它可用于扩展程序包中的模块集。

备注

CHAPTER 7

输入输出

有几种方法可以显示程序的输出；数据可以以人类可读的形式打印出来，或者写入文件以供将来使用。本章将讨论一些可能性。

7.1 更漂亮的输出格式

So far we’ ve encountered two ways of writing values: *expression statements* and the `print` statement. (A third way is using the `write()` method of file objects; the standard output file can be referenced as `sys.stdout`. See the Library Reference for more information on this.)

Often you’ ll want more control over the formatting of your output than simply printing space-separated values. There are two ways to format your output; the first way is to do all the string handling yourself; using string slicing and concatenation operations you can create any layout you can imagine. The string types have some methods that perform useful operations for padding strings to a given column width; these will be discussed shortly. The second way is to use the `str.format()` method.

The `string` module contains a `Template` class which offers yet another way to substitute values into strings.

One question remains, of course: how do you convert values to strings? Luckily, Python has ways to convert any value to a string: pass it to the `repr()` or `str()` functions.

The `str()` function is meant to return representations of values which are fairly human-readable, while `repr()` is meant to generate representations which can be read by the interpreter (or will force a `SyntaxError` if there is no equivalent syntax). For objects which don’ t have a particular representation for human consumption, `str()` will return the same value as `repr()`. Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function. Strings and floating point numbers, in particular, have two distinct representations.

几个例子:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
```

(下页继续)

(续上页)

```
>>> str(1.0/7.0)
'0.142857142857'
>>> repr(1.0/7.0)
'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'))"
```

Here are two ways to write a table of squares and cubes:

```
>>> for x in range(1, 11):
...     print repr(x).rjust(2), repr(x*x).rjust(3),
...     # Note trailing comma 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
```

(Note that in the first example, one space between each column was added by the way `print` works: by default it adds spaces between its arguments.)

This example demonstrates the `str.rjust()` method of string objects, which right-justifies a string in a field of a given width by padding it with spaces on the left. There are similar methods `str.ljust()` and `str.center()`. These methods do not write anything, they just return a new string. If the input string is too long, they don’t truncate it, but return it unchanged; this will mess up your column lay-out but that’s usually better than the alternative, which would be lying about a value. (If you really want truncation you can always add a slice operation, as in `x.ljust(n)[:n]`.)

还有另外一个方法，`str.zfill()` ，它会在数字字符串的左边填充零。它能识别正负号:

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

`str.format()` 方法的基本用法如下所示:

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

The brackets and characters within them (called format fields) are replaced with the objects passed into the `str.format()` method. A number in the brackets refers to the position of the object passed into the `str.format()` method.

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

如果在 `str.format()` 方法中使用关键字参数，则使用参数的名称引用它们的值。:

```
>>> 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任意组合:

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

`'!s'` (apply `str()`) and `'!r'` (apply `repr()`) can be used to convert the value before it is formatted.

```
>>> 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.
```

An optional `':'` and format specifier can follow the field name. This allows greater control over how the value is formatted. The following example rounds Pi to three places after the decimal.

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

Passing an integer after the `':'` will cause that field to be a minimum number of characters wide. This is useful for making tables pretty.

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

(下页继续)

(续上页)

```
...
Jack        ==>        4098
Dcab        ==>        7678
Sjoerd      ==>        4127
```

如果你有一个非常长的格式字符串，你不想把它拆开，那么你最好按名称而不是位置引用变量来进行格式化。这可以通过简单地传递字典和使用方括号 '[]' 访问键来完成:

```
>>> 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
```

这也可以通过使用 '**' 符号将 table 作为关键字参数传递。

```
>>> 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
```

这在与内置函数 vars() 结合使用时非常有用，它会返回包含所有局部变量的字典。

关于使用 str.format() 进行字符串格式化的完整概述，请参阅 formatstrings 。

7.1.1 旧的字符串格式化方法

% 操作符也可以用作字符串格式化。它将左边的参数解释为一个很像 sprintf() 风格的格式字符串，应用到右边的参数，并返回一个由此格式化操作产生的字符串。例如:

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

More information can be found in the string-formatting section.

7.2 读写文件

open() returns a file object, and is most commonly used with two arguments: open(filename, mode).

```
>>> f = open('workfile', 'w')
>>> print f
<open file 'workfile', mode 'w' at 80a0960>
```

第一个参数是包含文件名的字符串。第二个参数是另一个字符串，其中包含一些描述文件使用方式的字符。*mode* 可以是 'r'，表示文件只能读取，'w' 表示只能写入（已存在的同名文件会被删除），还有 'a' 表示打开文件以追加内容；任何写入的数据会自动添加到文件的末尾。'r+' 表示打开文件进行读写。*mode* 参数是可选的；省略时默认为 'r'。

On Windows, 'b' appended to the mode opens the file in binary mode, so there are also modes like 'rb', 'wb', and 'r+b'. Python on Windows makes a distinction between text and binary files; the end-of-line characters in text files are automatically altered slightly when data is read or written. This behind-the-scenes modification to file data is fine for ASCII text files, but it' ll corrupt binary data like that in JPEG or EXE files. Be very careful to use binary mode when reading and writing such files. On Unix, it doesn' t hurt to append a 'b' to the mode, so you can use it platform-independently for all binary files.

7.2.1 文件对象的方法

本节中剩下的例子将假定你已创建名为 f 的文件对象。

To read a file' s contents, call `f.read(size)`, which reads some quantity of data and returns it as a string. *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned; it' s your problem if the file is twice as large as your machine' s memory. Otherwise, at most *size* bytes are read and returned. If the end of the file has been reached, `f.read()` will return an empty string (`""`).

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

`f.readline()` reads a single line from the file; a newline character (`\n`) is left at the end of the string, and is only omitted on the last line of the file if the file doesn' t end in a newline. This makes the return value unambiguous; if `f.readline()` returns an empty string, the end of the file has been reached, while a blank line is represented by `'\n'`, a string containing only a single newline.

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

要从文件中读取行，你可以循环遍历文件对象。这是内存高效，快速的，并简化代码:

```
>>> for line in f:
        print line,

This is the first line of the file.
Second line of the file
```

如果你想以列表的形式读取文件中的所有行，你也可以使用 `list(f)` 或 `f.readlines()`。

`f.write(string)` writes the contents of *string* to the file, returning `None`.

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

To write something other than a string, it needs to be converted to a string first:

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

`f.tell()` returns an integer giving the file object' s current position in the file, measured in bytes from the beginning of the file. To change the file object' s position, use `f.seek(offset, from_what)`. The position is computed from adding *offset* to a reference point; the reference point is selected by the *from_what* argument. A *from_what* value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. *from_what* can be omitted and defaults to 0, using the beginning of the file as the reference point.

```
>>> f = open('workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)      # Go to the 6th byte in the file
>>> f.read(1)
'5'
```

(下页继续)

(续上页)

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

When you're done with a file, call `f.close()` to close it and free up any system resources taken up by the open file. After calling `f.close()`, attempts to use the file object will automatically fail.

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

It is good practice to use the `with` keyword when dealing with file objects. This has the advantage that the file is properly closed after its suite finishes, even if an exception is raised on the way. It is also much shorter than writing equivalent `try-finally` blocks:

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

文件对象有一些额外的方法，例如 `isatty()` 和 `truncate()` ，它们使用频率较低；有关文件对象的完整指南请参阅库参考。

7.2.2 使用 json 保存结构化数据

字符串可以很轻松地写入文件并从文件中读取出来。数字可能会费点劲，因为 `read()` 方法只能返回字符串，这些字符串必须传递给类似 `int()` 的函数，它会接受类似 `'123'` 这样的字符串并返回其数字值 123。当你想保存诸如嵌套列表和字典这样更复杂的数据类型时，手动解析和序列化会变得复杂。

Python 允许你使用称为 JSON (JavaScript Object Notation) 的流行数据交换格式，而不是让用户不断的编写和调试代码以将复杂的数据类型保存到文件中。名为 `json` 的标准模块可以采用 Python 数据层次结构，并将它们转化为字符串表示形式；这个过程称为 *serializing* 。从字符串表示中重建数据称为 *deserializing* 。在序列化和反序列化之间，表示对象的字符串可能已存储在文件或数据中，或通过网络连接发送到某个远程机器。

注解： JSON 格式通常被现代应用程序用于允许数据交换。许多程序员已经熟悉它，这使其成为互操作性的良好选择。

如果你有一个对象 `x` ，你可以用一行简单的代码来查看它的 JSON 字符串表示：

```
>>> import json
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

Another variant of the `dumps()` function, called `dump()`, simply serializes the object to a file. So if `f` is a *file object* opened for writing, we can do this:

```
json.dump(x, f)
```

To decode the object again, if `f` is a *file object* which has been opened for reading:

```
x = json.load(f)
```

这种简单的序列化技术可以处理列表和字典，但是在 JSON 中序列化任意类的实例需要额外的努力。`json` 模块的参考包含对此的解释。

参见:

`pickle` - 封存模块

与*JSON* 不同，*pickle* 是一种允许对任意复杂 Python 对象进行序列化的协议。因此，它为 Python 所特有，不能用于与其他语言编写的应用程序通信。默认情况下它也是不安全的：如果数据是由熟练的攻击者精心设计的，则反序列化来自不受信任来源的 pickle 数据可以执行任意代码。

CHAPTER 8

错误和异常

到目前为止，我们还没有提到错误消息，但是如果你已经尝试过那些例子，你可能已经看过了一些错误消息。目前（至少）有两种可区分的错误：语法错误和 异常。

8.1 语法错误

语法错误又称解析错误，可能是你在学习 Python 时最容易遇到的错误:

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

The parser repeats the offending line and displays a little ‘arrow’ pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token *preceding* the arrow: in the example, the error is detected at the keyword `print`, since a colon (`':'`) is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

8.2 异常

即使语句或表达式在语法上是正确的，但在尝试执行时，它仍可能会引发错误。在执行时检测到的错误被称为 异常，异常不一定会导致严重后果：你将很快学会如何在 Python 程序中处理它们。但是，大多数异常并不会被程序处理，此时会显示如下所示的错误信息:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
```

(下页继续)

(续上页)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

错误信息的最后一行告诉我们程序遇到了什么类型的错误。异常有不同的类型，而其类型名称将会作为错误信息的一部分中打印出来：上述示例中的异常类型依次是：ZeroDivisionError，NameError 和 TypeError。作为异常类型打印的字符串是发生的内置异常的名称。对于所有内置异常都是如此，但对于用户定义的异常则不一定如此（虽然这是一个有用的规范）。标准的异常类型是内置的标识符（而不是保留关键字）。

这一行的剩下的部分根据异常类型及其原因提供详细信息。

错误信息的前一部分以堆栈回溯的形式显示发生异常时的上下文。通常它包含列出源代码行的堆栈回溯；但是它不会显示从标准输入中读取的行。

bltin-exceptions 列出了内置异常和它们的含义。

8.3 处理异常

可以编写处理所选异常的程序。请看下面的例子，它会要求用户一直输入，直到输入的是一个有效的整数，但允许用户中断程序（使用 Control-C 或操作系统支持的其他操作）；请注意用户引起的中断可以通过引发 KeyboardInterrupt 异常来指示。:

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

try 语句的工作原理如下。

- 首先，执行 *try* 子句（try 和 except 关键字之间的（多行）语句）。
- 如果没有异常发生，则跳过 *except* 子句并完成 try 语句的执行。
- 如果在执行 try 子句时发生了异常，则跳过该子句中剩下的部分。然后，如果异常的类型和 except 关键字后面的异常匹配，则执行 except 子句，然后继续执行 try 语句之后的代码。
- 如果发生的异常和 except 子句中指定的异常不匹配，则将其传递到外部的 try 语句中；如果没有找到处理程序，则它是一个 未处理异常，执行将停止并显示如上所示的消息。

A try statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement. An except clause may name multiple exceptions as a parenthesized tuple, for example:

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

Note that the parentheses around this tuple are required, because except ValueError, e: was the syntax used for what is normally written as except ValueError as e: in modern Python (described below). The old syntax is still supported for backwards compatibility. This means except RuntimeError, TypeError is not equivalent to `except (RuntimeError, TypeError):` but to `except RuntimeError as TypeError:` which is not what you want.

最后的 except 子句可以省略异常名，以用作通配符。但请谨慎使用，因为以这种方式很容易掩盖真正的编程错误！它还可用于打印错误消息，然后重新引发异常（同样允许调用者处理异常）：

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    print "I/O error({0}): {1}".format(e.errno, e.strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

`try` …`except` 语句有一个可选的 *else* 子句，在使用时必须放在所有的 except 子句后面。对于在 try 子句不引发异常时必须执行的代码来说很有用。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()
```

The use of the `else` clause is better than adding additional code to the `try` clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try` …`except` statement.

发生异常时，它可能具有关联值，也称为异常 *参数*。参数的存在和类型取决于异常类型。

The except clause may specify a variable after the exception name (or tuple). The variable is bound to an exception instance with the arguments stored in `instance.args`. For convenience, the exception instance defines `__str__()` so the arguments can be printed directly without having to reference `.args`.

One may also instantiate an exception first before raising it and add any attributes to it as desired.

```
>>> 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 be printed directly
...     x, y = inst.args
...     print 'x =', x
...     print 'y =', y
...
<type 'exceptions.Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

If an exception has an argument, it is printed as the last part ('detail') of the message for unhandled exceptions.

异常处理程序不仅处理 try 子句中遇到的异常，还处理 try 子句中调用（即使是间接地）的函数内部发生的异常。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
```

8.4 抛出异常

`raise` 语句允许程序员强制发生指定的异常。例如:

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

The sole argument to `raise` indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from `Exception`).

如果你需要确定是否引发了异常但不打算处理它，则可以使用更简单的 `raise` 语句形式重新引发异常

```
>>> 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 <module>
NameError: HiThere
```

8.5 用户自定义异常

Programs may name their own exceptions by creating a new exception class (see 类 for more about Python classes). Exceptions should typically be derived from the `Exception` class, either directly or indirectly. For example:

```
>>> 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:
```

(下页继续)

(续上页)

```
...     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 <module>
__main__.MyError: 'oops!'
```

In this example, the default `__init__()` of `Exception` has been overridden. The new behavior simply creates the *value* attribute. This replaces the default behavior of creating the *args* attribute.

可以定义异常类，它可以执行任何其他类可以执行的任何操作，但通常保持简单，通常只提供许多属性，这些属性允许处理程序为异常提取有关错误的信息。在创建可能引发多个不同错误的模块时，通常的做法是为该模块定义的异常创建基类，并为不同错误条件创建特定异常类的子类:

```
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
```

大多数异常都定义为名称以 “Error” 结尾，类似于标准异常的命名。

许多标准模块定义了它们自己的异常，以报告它们定义的函数中可能出现的错误。有关类的更多信息，请参见类类。

8.6 定义清理操作

try 语句有另一个可选子句，用于定义必须在所有情况下执行的清理操作。例如:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Goodbye, world!'
...
Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

A *finally clause* is always executed before leaving the `try` statement, whether an exception has occurred or not. When an exception has occurred in the `try` clause and has not been handled by an `except` clause (or it has occurred in an `except` or `else` clause), it is re-raised after the `finally` clause has been executed. The `finally` clause is also executed "on the way out" when any other clause of the `try` statement is left via a `break`, `continue` or `return` statement. A more complicated example (having `except` and `finally` clauses in the same `try` statement works as of Python 2.5):

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print "division by zero!"
...     else:
...         print "result is", 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 "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

As you can see, the `finally` clause is executed in any event. The `TypeError` raised by dividing two strings is not handled by the `except` clause and therefore re-raised after the `finally` clause has been executed.

在实际应用程序中，`finally` 子句对于释放外部资源（例如文件或者网络连接）非常有用，无论是否成功使用资源。

8.7 预定义的清理操作

某些对象定义了在不再需要该对象时要执行的标准清理操作，无论使用该对象的操作是成功还是失败。请查看下面的示例，它尝试打开一个文件并把其内容打印到屏幕上。:

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

The problem with this code is that it leaves the file open for an indeterminate amount of time after the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The `with` statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

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

After the statement is executed, the file *f* is always closed, even if a problem was encountered while processing the lines. Other objects which provide predefined clean-up actions will indicate this in their documentation.

CHAPTER 9

类

和其他编程语言相比，Python 用非常少的新语法和语义将类加入到语言中。它是 C++ 和 Modula-3 中类机制的结合。Python 的类提供了面向对象编程的所有标准特性：类继承机制允许多个基类，派生类可以覆盖它基类的任何方法，一个方法可以调用基类中相同名称的的方法。对象可以包含任意数量和类型的数据。和模块一样，类也拥有 Python 天然的动态特性：它们在运行时创建，可以在创建后修改。

在 C++ 术语中，通常类成员（包括数据成员）是 *public* (例外见下文*Private Variables and Class-local References*)，所有成员函数都是 *virtual*。与在 Modula-3 中一样，没有用于从其方法引用对象成员的简写：方法函数使用表示对象的显式第一个参数声明，该参数由调用隐式提供。与 Smalltalk 一样，类本身也是对象。这为导入和重命名提供了语义。与 C++ 和 Modula-3 不同，内置类型可以用作用户扩展的基类。此外，与 C++ 一样，大多数具有特殊语法（算术运算符，下标等）的内置运算符都可以为类实例而重新定义。

（由于缺乏关于类的公认术语，我会偶尔使用 Smalltalk 和 C++ 的用辞。我还会使用 Modula-3 的术语，因为其面向对象的语义比 C++ 更接近 Python，但我预计少有读者听说过它。）

9.1 名称和对象

对象具有个性，多个名称（在多个作用域内）可以绑定到同一个对象。这在其他语言中称为别名。乍一看 Python 时通常不会理解这一点，在处理不可变的基本类型（数字，字符串，元组）时可以安全地忽略它。但是，别名对涉及可变对象，如列表，字典和大多数其他类型，的 Python 代码的语义可能会产生惊人的影响。这通常用于程序的好处，因为别名在某些方面表现得像指针。例如，传递一个对象很便宜，因为实现只传递一个指针；如果函数修改了作为参数传递的对象，调用者将看到更改—这就不需要像 Pascal 中那样使用两个不同的参数传递机制。

9.2 Python 作用域和命名空间

在介绍类之前，我首先要告诉你一些 Python 的作用域规则。类定义对命名空间有一些巧妙的技巧，你需要知道作用域和命名空间如何工作才能完全理解正在发生的事情。顺便说一下，关于这个主题的知识对任何高级 Python 程序员都很有用。

让我们从一些定义开始。

namespace（命名空间）是一个从名字到对象的映射。大部分命名空间当前都由 Python 字典实现，但一般情况下基本不会去关注它们（除了要面对性能问题时），而且也有可能在将来更改。下面是几个命名空间的例子：存放内置函数的集合（包含 `abs()` 这样的函数，和内建的异常等）；模块中的全局名称；函数调用中的局部名称。从某种意义上说，对象的属性集合也是一种命名空间的形式。关于命名空间的重要一点是，不同命名空间中的名称之间绝对没有关系；例如，两个不同的模块都可以定义一个 `maximize` 函数而不会产生混淆—模块的用户必须在其前面加上模块名称。

顺便说明一下，我把任何跟在一个点号之后的名称都称为 *属性*—例如，在表达式 `z.real` 中，`real` 是对象 `z` 的一个属性。按严格的说法，对模块中名称的引用属于属性引用：在表达式 `modname.funcname` 中，`modname` 是一个模块对象而 `funcname` 是它的一个属性。在此情况下在模块的属性和模块中定义的全局名称之间正好存在一个直观的映射：它们共享相同的命名空间！[1]

属性可以是只读或者可写的。如果为后者，那么对属性的赋值是可行的。模块属性是可以写，你可以写出 `modname.the_answer = 42`。可写的属性同样可以用 `del` 语句删除。例如，`del modname.the_answer` 将会从名为 `modname` 的对象中移除 `the_answer` 属性。

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `__builtin__`.)

一个函数的本地命名空间在这个函数被调用时创建，并在函数返回或抛出一个不在函数内部处理的错误时被删除。（事实上，比起描述到底发生了什么，忘掉它更好。）当然，每次递归调用都会有它自己的本地命名空间。

一个 *作用域* 是一个命名空间可直接访问的 Python 程序的文本区域。这里的 "可直接访问" 意味着对名称的非限定引用会尝试在命名空间中查找名称。

作用域被静态确定，但被动态使用。在程序运行的任何时间，至少有三个命名空间可被直接访问的嵌套作用域：

- 最先搜索的最内部作用域包含局部名称
- 从最近的封闭作用域开始搜索的任何封闭函数的作用域包含非局部名称，也包括非全局名称
- 倒数第二个作用域包含当前模块的全局名称
- 最外面的作用域（最后搜索）是包含内置名称的命名空间

If a name is declared global, then all references and assignments go directly to the middle scope containing the module's global names. Otherwise, all variables found outside of the innermost scope are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

通常，当前局部作为域将（按字面文本）引用当前函数的局部名称。在函数以外，局部作用域将引用与全局作用域相一致的命名空间：模块的命名空间。类定义将在局部命名空间内再放置另一个命名空间。

[1] 存在一个例外。模块对象有一个秘密的只读属性 `__dict__`，它返回用于实现模块命名空间的字典；`__dict__` 是属性但不是全局名称。显然，使用这个将违反命名空间实现的抽象，应当仅被用于事后调试器之类的场合。

重要的是应该意识到作用域是按字面文本来确定的：在一个模块内定义的函数的全局作用域就是该模块的命名空间，无论该函数从什么地方或以什么别名被调用。另一方面，实际的名称搜索是在运行时动态完成的—但是，Python 正在朝着"编译时静态名称解析"的方向发展，因此不要过于依赖动态名称解析！（事实上，局部变量已经是被静态确定了。）

A special quirk of Python is that –if no `global` statement is in effect –assignments to names always go into the innermost scope. Assignments do not copy data —they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope. (The `global` statement can be used to indicate that particular variables live in the global scope.)

9.3 初探类

类引入了一些新语法，三种新对象类型和一些新语义。

9.3.1 类定义语法

最简单的类定义看起来像这样:

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

类定义与函数定义 (`def` 语句) 一样必须被执行才会起作用。（你可以尝试将类定义放在 `if` 语句的一个分支或是函数的内部。）

在实践中，类定义内的语句通常都是函数定义，但也允许有其他语句，有时还很有用—我们会稍后再回来说明这个问题。在类内部的函数定义通常具有一种特别形式的参数列表，这是方法调用的约定规范所指明的—这个问题也将在稍后再说明。

当进入类定义时，将创建一个新的命名空间，并将其用作局部作用域—因此，所有对局部变量的赋值都是在这个新命名空间之内。特别的，函数定义会绑定到这里的新函数名称。

当（从结尾处）正常离开类定义时，将创建一个 *类对象*。这基本上是一个包围在类定义所创建命名空间内容周围的包装器；我们将在下一节了解有关类对象的更多信息。原始的（在进入类定义之前起作用的）局部作用域将重新生效，类对象将在这里被绑定到类定义头所给出的类名称 (在这个示例中为 `ClassName`)。

9.3.2 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 Python 中所有属性引用所使用的标准语法: `obj.name`。有效的属性名称是类对象被创建时存在于类命名空间中的所有名称。因此，如果类定义是这样的:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

那么 MyClass.i 和 MyClass.f 就是有效的属性引用，将分别返回一个整数和一个函数对象。类属性也可以被赋值，因此可以通过赋值来更改 MyClass.i 的值。__doc__ 也是一个有效的属性，将返回所属类的文档字符串: "A simple example class"。

类的 *实例化* 使用函数表示法。可以把类对象视为是返回该类的一个新实例的不带参数的函数。举例来说（假设使用上述的类）：

```
x = MyClass()
```

创建类的新 *实例* 并将此对象分配给局部变量 x。

实例化操作（“调用”类对象）会创建一个空对象。许多类喜欢创建带有特定初始状态的自定义实例。为此类定义可能包含一个名为 __init__() 的特殊方法，就像这样:

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

当一个类定义了 __init__() 方法时，类的实例化操作会自动为新创建的类实例发起调用 __init__()。因此在这个示例中，可以通过以下语句获得一个经初始化的新实例:

```
x = MyClass()
```

当然，__init__() 方法还可以有额外参数以实现更高灵活性。在这种情况下，提供给类实例化运算符的参数将被传递给 __init__()。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)
```

9.3.3 实例对象

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names, data attributes and methods.

数据属性 对应于 Smalltalk 中的“实例变量”，以及 C++ 中的“数据成员”。数据属性不需要声明；像局部变量一样，它们将在第一次被赋值时产生。例如，如果 x 是上面创建的 MyClass 的实例，则以下代码段将打印数值 16，且不保留任何追踪信息:

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

另一类实例属性引用称为 *方法*。方法是“从属于”对象的函数。（在 Python 中，方法这个术语并不是类实例所特有的：其他对象也可以有方法。例如，列表对象具有 append, insert, remove, sort 等方法。然而，在以下讨论中，我们使用方法一词将专指类实例对象的方法，除非另外显式地说明。）

实例对象的有效方法名称依赖于其所属的类。根据定义，一个类中所有是函数对象的属性都是定义了其实例的相应方法。因此在我们的示例中，x.f 是有效的方法引用，因为 MyClass.f 是一个函数，而 x.i 不是方法，因为 MyClass.i 不是一个函数。但是 x.f 与 MyClass.f 并不是一回事—它是一个 *方法对象*，不是函数对象。

9.3.4 方法对象

通常，方法在绑定后立即被调用:

```
x.f()
```

在 `MyClass` 示例中，这将返回字符串 `'hello world'`。但是，立即调用一个方法并不是必须的: `x.f` 是一个方法对象，它可以被保存起来以后再调用。例如:

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

将继续打印 `hello world`，直到结束。

当一个方法被调用时到底发生了什么？你可能已经注意到上面调用 `x.f()` 时并没有带参数，虽然 `f()` 的函数定义指定了一个参数。这个参数发生了什么事？当不带参数地调用一个需要参数的函数时 Python 肯定会引发异常—即使参数实际未被使用…

Actually, you may have guessed the answer: the special thing about methods is that the object is passed as the first argument of the function. In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`. In general, calling a method with a list of *n* arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method' s object before the first argument.

如果你仍然无法理解方法的运作原理，那么查看实现细节可能会澄清问题。当一个实例的非数据属性被引用时，将搜索实例所属的类。如果名称表示一个属于函数对象的有效类属性，会通过合并打包（指向）实例对象和函数对象到一个抽象对象中的方式来创建一个方法对象：这个抽象对象就是方法对象。当附带参数列表调用方法对象时，将基于实例对象和参数列表构建一个新的参数列表，并使用这个新参数列表调用相应的函数对象。

9.3.5 类和实例变量

一般来说，实例变量用于每个实例的唯一数据，而类变量用于类的所有实例共享的属性和方法:

```
class Dog:

    kind = 'canine'         # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                  # unique to d
'Fido'
>>> e.name                  # unique to e
'Buddy'
```

正如名称和对象 中已讨论过的，共享数据可能在涉及*mutable* 对象例如列表和字典的时候导致令人惊讶的结果。例如以下代码中的 *tricks* 列表不应该被用作类变量，因为所有的 *Dog* 实例将只共享一个单独的列表:

```
class Dog:

    tricks = []             # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

正确的类设计应该使用实例变量:

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

9.4 补充说明

数据属性会覆盖掉具有相同名称的方法属性；为了避免会在大型程序中导致难以发现的错误的意外名称冲突，明智的做法是使用某种约定来最小化冲突的发生几率。可能的约定包括方法名称使用大写字母，属性名称加上独特的短字符串前缀（或许只加一个下划线），或者是用动词来命名方法，而用名词来命名数据属性。

数据属性可以被方法以及一个对象的普通用户（"客户端"）所引用。换句话说，类不能用于实现纯抽象数据类型。实际上，在 Python 中没有任何东西能强制隐藏数据—它是完全基于约定的。(而在另一方面，用 C 语言编写的 Python 实现则可以完全隐藏实现细节，并在必要时控制对象的访问；此特性可以通过用 C 编写 Python 扩展来使用。)

客户端应当谨慎地使用数据属性—客户端可能通过直接操作数据属性的方式破坏由方法所维护的固定变量。请注意客户端可以向一个实例对象添加他们自己的数据属性而不会影响方法的可用性，只要保证避免名称冲突—再次提醒，在此使用命名约定可以省去许多令人头痛的麻烦。

在方法内部引用数据属性（或其他方法！）并没有简便方式。我发现这实际上提升了方法的可读性：当浏览一个方法代码时，不会存在混淆局部变量和实例变量的机会。

方法的第一个参数常常被命名为 `self`。这也不过就是一个约定: `self` 这一名称在 Python 中绝对没有特殊含

义。但是要注意，不遵循此约定会使得你的代码对其他 Python 程序员来说缺乏可读性，而且也可以想像一个类浏览器程序的编写可能会依赖于这样的约定。

任何一个作为类属性的函数都为该类的实例定义了一个相应方法。函数定义的文本并非必须包含于类定义之内：将一个函数对象赋值给一个局部变量也是可以的。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 `C` 类的引用函数对象的属性，因而它们就都是 `C` 的实例的方法—其中 `h` 完全等同于 `g`。但请注意，本示例的做法通常只会令程序的阅读者感到迷惑。

方法可以通过使用 `self` 参数的方法属性调用其他方法:

```
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)
```

方法可以通过与普通函数相同的方式引用全局名称。与方法相关联的全局作用域就是包含其定义的模块。(类永远不会被作为全局作用域。）虽然我们很少会有充分的理由在方法中使用全局作用域，但全局作用域存在许多合法的使用场景：举个例子，导入到全局作用域的函数和模块可以被方法所使用，在其中定义的函数和类也一样。通常，包含该方法的类本身是在全局作用域中定义的，而在下一节中我们将会发现为何方法需要引用其所属类的很好的理由。

每个值都是一个对象，因此具有 *类* （也称为 *类型*），并存储为 `object.__class__` 。

9.5 继承

当然，如果不支持继承，语言特性就不值得称为“类”。派生类定义的语法如下所示:

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

名称 `BaseClassName` 必须定义于包含派生类定义的作用域中。也允许用其他任意表达式代替基类名称所在的位置。这有时也可能会用得上，例如，当基类定义在另一个模块中的时候:

```
class DerivedClassName(modname.BaseClassName):
```

派生类定义的执行过程与基类相同。当构造类对象时，基类会被记住。此信息将被用来解析属性引用：如果请求的属性在类中找不到，搜索将转往基类中进行查找。如果基类本身也派生自其他某个类，则此规则将被递归地应用。

派生类的实例化没有任何特殊之处: `DerivedClassName()` 会创建该类的一个新实例。方法引用将按以下方式解析：搜索相应的类属性，如有必要将按基类继承链逐步向下查找，如果产生了一个函数对象则方法引用就生效。

派生类可能会重载其基类的方法。因为方法在调用同一对象的其他方法时没有特殊权限，调用同一基类中定义的另一方法的基类方法最终可能会调用覆盖它的派生类的方法。(对 C++ 程序员的提示：Python 中所有的方法实际上都是 `virtual` 方法。)

在派生类中的重载方法实际上可能想要扩展而非简单地替换同名的基类方法。有一种方式可以简单地直接调用基类方法：即调用 `BaseClassName.methodname(self, arguments)`。有时这对客户端来说也是有用的。(请注意仅当此基类可在全局作用域中以 `BaseClassName` 的名称被访问时方可使用此方式。)

Python 有两个内置函数可被用于继承机制：

- 使用 `isinstance()` 来检查一个实例的类型: `isinstance(obj, int)` 仅会在 `obj.__class__` 为 `int` 或某个派生自 `int` 的类时为 `True`。
- Use `issubclass()` to check class inheritance: `issubclass(bool, int)` is `True` since `bool` is a subclass of `int`. However, `issubclass(unicode, str)` is `False` since `unicode` is not a subclass of `str` (they only share a common ancestor, `basestring`).

9.5.1 多重继承

Python supports a limited form of multiple inheritance as well. A class definition with multiple base classes looks like this:

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

For old-style classes, the only rule is depth-first, left-to-right. Thus, if an attribute is not found in `DerivedClassName`, it is searched in `Base1`, then (recursively) in the base classes of `Base1`, and only if it is not found there, it is searched in `Base2`, and so on.

(To some people breadth first —searching `Base2` and `Base3` before the base classes of `Base1` —looks more natural. However, this would require you to know whether a particular attribute of `Base1` is actually defined in `Base1` or in one of its base classes before you can figure out the consequences of a name conflict with an attribute of `Base2`. The depth-first rule makes no differences between direct and inherited attributes of `Base1`.)

For *new-style class*es, the method resolution order changes dynamically to support cooperative calls to `super()`. This approach is known in some other multiple-inheritance languages as call-next-method and is more powerful than the super call found in single-inheritance languages.

With new-style classes, dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all new-style classes inherit from `object`, so any case of multiple inheritance provides more than one path to reach `object`. To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents).

Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance. For more detail, see https://www.python.org/download/releases/2.3/mro/.

9.6 Private Variables and Class-local References

那种仅限从一个对象内部访问的 "私有" 实例变量在 Python 中并不存在。但是，大多数 Python 代码都遵循这样一个约定：带有一个下划线的名称 (例如 _spam) 应该被当作是 API 的非公有部分 (无论它是函数、方法或是数据成员)。这应当被视为一个实现细节，可能不经通知即加以改变。

由于存在对于类私有成员的有效使用场景（例如避免名称与子类所定义的名称相冲突），因此存在对此种机制的有限支持，称为 *名称改写*。任何形式为 __spam 的标识符（至少带有两个前缀下划线，至多一个后缀下划线）的文本将被替换为 _classname__spam，其中 classname 为去除了前缀下划线的当前类名称。这种改写不考虑标识符的句法位置，只要它出现在类定义内部就会进行。

名称改写有助于让子类重载方法而不破坏类内方法调用。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)
```

上面的示例即使在 MappingSubclass 引入了一个 __update 标识符的情况下也不会出错，因为它会在 Mapping 类中被替换为 _Mapping__update 而在 MappingSubclass 类中被替换为 _MappingSubclass__update。

请注意，改写规则的设计主要是为了避免意外冲突；访问或修改被视为私有的变量仍然是可能的。这在特殊情况下甚至会很有用，例如在调试器中。

Notice that code passed to exec, eval() or execfile() does not consider the classname of the invoking class to be the current class; this is similar to the effect of the global statement, the effect of which is likewise restricted to code that is byte-compiled together. The same restriction applies to getattr(), setattr() and delattr(), as well as when referencing __dict__ directly.

9.7 杂项说明

有时会需要使用类似于 Pascal 的 "record" 或 C 的 "struct" 这样的数据类型，将一些命名数据项捆绑在一起。这种情况适合定义一个空类:

```
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
```

一段需要特定抽象数据类型的 Python 代码往往可以被传入一个模拟了该数据类型的方法的类作为替代。例如，如果你有一个基于文件对象来格式化某些数据的函数，你可以定义一个带有 `read()` 和 `readline()` 方法从字符串缓存获取数据的类，并将其作为参数传入。

Instance method objects have attributes, too: `m.im_self` is the instance object with the method `m()`, and `m.im_func` is the function object corresponding to the method.

9.8 Exceptions Are Classes Too

User-defined exceptions are identified by classes as well. Using this mechanism it is possible to create extensible hierarchies of exceptions.

There are two new valid (semantic) forms for the `raise` statement:

```
raise Class, instance

raise instance
```

In the first form, `instance` must be an instance of `Class` or of a class derived from it. The second form is a shorthand for:

```
raise instance.__class__, instance
```

A class in an `except` clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around —an except clause listing a derived class is not compatible with a base class). For example, the following code will print B, C, D in that order:

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

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

(下页继续)

(续上页)

```
    except C:
        print "C"
    except B:
        print "B"
```

Note that if the except clauses were reversed (with `except B` first), it would have printed B, B, B —the first matching except clause is triggered.

When an error message is printed for an unhandled exception, the exception’ s class name is printed, then a colon and a space, and finally the instance converted to a string using the built-in function `str()`.

9.9 迭代器

到目前为止，您可能已经注意到大多数容器对象都可以使用 `for` 语句:

```
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,
```

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python. Behind the scenes, the `for` statement calls `iter()` on the container object. The function returns an iterator object that defines the method `next()` which accesses elements in the container one at a time. When there are no more elements, `next()` raises a `StopIteration` exception which tells the `for` loop to terminate. This example shows how it all works:

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

Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. Define an `__iter__()` method which returns an object with a `next()` method. If the class defines `next()`, then `__iter__()` can just return `self`:

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
```

(下页继续)

(续上页)

```
        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
```

9.10 生成器

Generator 是一个用于创建迭代器的简单而强大的工具。它们的写法类似标准的函数，但当它们要返回数据时会使用 `yield` 语句。每次对生成器调用 `next()` 时，它会从上次离开位置恢复执行（它会记住上次执行语句时的所有数据值）。显示如何非常容易地创建生成器的示例如下:

```
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
```

Anything that can be done with generators can also be done with class-based iterators as described in the previous section. What makes generators so compact is that the `__iter__()` and `next()` methods are created automatically.

另一个关键特性在于局部变量和执行状态会在每次调用之间自动保存。这使得该函数相比使用 `self.index` 和 `self.data` 这种实例变量的方式更易编写且更为清晰。

除了会自动创建方法和保存程序状态，当生成器终结时，它们还会自动引发 `StopIteration`。这些特性结合在一起，使得创建迭代器能与编写常规函数一样容易。

9.11 生成器表达式

某些简单的生成器可以写成简洁的表达式代码，所用语法类似列表推导式，将外层为圆括号而非方括号。这种表达式被设计用于生成器将立即被外层函数所使用的情况。生成器表达式相比完整的生成器更紧凑但较不灵活，相比等效的列表推导式则更为节省内存。

示例:

```
>>> 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 = 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']
```

备注

CHAPTER 10

标准库简介

10.1 操作系统接口

os 模块提供了许多与操作系统交互的函数:

```
>>> import os
>>> os.getcwd()      # Return the current working directory
'C:\\Python26'
>>> os.chdir('/server/accesslogs')   # Change current working directory
>>> os.system('mkdir today')   # Run the command mkdir in the system shell
0
```

一定要使用 import os 而不是 from os import * 。这将避免内建的 open() 函数被 os.open() 隐式替换掉，它们的使用方式大不相同。

内置的 dir() 和 help() 函数可用作交互式辅助工具，用于处理大型模块，如 os:

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

对于日常文件和目录管理任务，shutil 模块提供了更易于使用的更高级别的接口:

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

10.2 文件通配符

glob 模块提供了一个在目录中使用通配符搜索创建文件列表的函数:

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

10.3 命令行参数

通用实用程序脚本通常需要处理命令行参数。这些参数作为列表存储在 sys 模块的 *argv* 属性中。例如，以下输出来自在命令行运行 python demo.py one two three

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

The getopt module processes *sys.argv* using the conventions of the Unix getopt() function. More powerful and flexible command line processing is provided by the argparse module.

10.4 错误输出重定向和程序终止

sys 模块还具有 *stdin*，*stdout* 和 *stderr* 的属性。后者对于发出警告和错误消息非常有用，即使在 *stdout* 被重定向后也可以看到它们:

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

终止脚本的最直接方法是使用 sys.exit() 。

10.5 字符串模式匹配

re 模块为高级字符串处理提供正则表达式工具。对于复杂的匹配和操作，正则表达式提供简洁，优化的解决方案:

```
>>> 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'
```

当只需要简单的功能时，首选字符串方法因为它们更容易阅读和调试:

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

10.6 数学

math 模块提供对浮点数学的底层 C 库函数的访问:

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

random 模块提供了进行随机选择的工具:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(xrange(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
```

10.7 互联网访问

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are urllib2 for retrieving data from URLs and smtplib for sending mail:

```
>>> import urllib2
>>> for line in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
...     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()
```

(请注意，第二个示例需要在 localhost 上运行的邮件服务器。)

10.8 日期和时间

`datetime` 模块提供了以简单和复杂的方式操作日期和时间的类。虽然支持日期和时间算法，但实现的重点是有效的成员提取以进行输出格式化和操作。该模块还支持可感知时区的对象。

```
>>> # 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
```

10.9 数据压缩

Common data archiving and compression formats are directly supported by modules including: `zlib`, `gzip`, `bz2`, `zipfile` and `tarfile`.

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

10.10 性能测量

一些 Python 用户对了解同一问题的不同方法的相对性能产生了浓厚的兴趣。Python 提供了一种可以立即回答这些问题的测量工具。

例如，元组封包和拆包功能相比传统的交换参数可能更具吸引力。`timeit` 模块可以快速演示在运行效率方面一定的优势:

```
>>> 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` 的精细粒度级别相反，`profile` 和 `pstats` 模块提供了用于在较大的代码块中识别时间关键部分的工具。

10.11 质量控制

开发高质量软件的一种方法是在开发过程中为每个函数编写测试，并在开发过程中经常运行这些测试。

doctest 模块提供了一个工具，用于扫描模块并验证程序文档字符串中嵌入的测试。测试构造就像将典型调用及其结果剪切并粘贴到文档字符串一样简单。这通过向用户提供示例来改进文档，并且它允许 doctest 模块确保代码保持对文档的真实:

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

    >>> print average([20, 30, 70])
    40.0
    """
    return sum(values, 0.0) / len(values)

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

unittest 模块不像 doctest 模块那样易于使用，但它允许在一个单独的文件中维护更全面的测试集:

```
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)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

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

10.12 自带电池

Python 有“自带电池”的理念。通过其包的复杂和强大功能可以最好地看到这一点。例如:

- The xmlrpclib and SimpleXMLRPCServer modules make implementing remote procedure calls into an almost trivial task. Despite the modules names, no direct knowledge or handling of XML is needed.
- The email package is a library for managing email messages, including MIME and other RFC 2822-based message documents. Unlike smtplib and poplib which actually send and receive messages, the email package has a complete toolset for building or decoding complex message structures (including attachments) and for implementing internet encoding and header protocols.
- The xml.dom and xml.sax packages provide robust support for parsing this popular data interchange format. Likewise, the csv module supports direct reads and writes in a common database format. Together, these modules and packages greatly simplify data interchange between Python applications and other tools.
- 国际化由许多模块支持，包括 gettext ， locale ， 以及 codecs 包。

CHAPTER 11

标准库简介——第二部分

第二部分涵盖了专业编程所需要的更高级的模块。这些模块很少用在小脚本中。

11.1 格式化输出

The `repr` module provides a version of `repr()` customized for abbreviated displays of large or deeply nested containers:

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

`pprint` 模块提供了更加复杂的打印控制，其输出的内置对象和用户自定义对象能够被解释器直接读取。当输出结果过长而需要折行时，“美化输出机制”会添加换行符和缩进，以更清楚地展示数据结构:

```
>>> 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` 模块能够格式化文本段落，以适应给定的屏幕宽度:

```
>>> 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)
```

(下页继续)

(续上页)

```
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.
```

`locale` 模块处理与特定地域文化相关的数据格式。locale 模块的 format 函数包含一个 grouping 属性，可直接将数字格式化为带有组分隔符的样式:

```
>>> 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'
```

11.2 模板

`string` 模块包含一个通用的 `Template` 类，具有适用于最终用户的简化语法。它允许用户在不更改应用逻辑的情况下定制自己的应用。

上述格式化操作是通过占位符实现的，占位符由 `$` 加上合法的 Python 标识符（只能包含字母、数字和下划线）构成。一旦使用花括号将占位符括起来，就可以在后面直接跟上更多的字母和数字而无需空格分割。`$$` 将被转义成单个字符 `$`:

```
>>> 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.'
```

如果在字典或关键字参数中未提供某个占位符的值，那么 `substitute()` 方法将抛出 `KeyError`。对于邮件合并类型的应用，用户提供的数据有可能是不完整的，此时使用 `safe_substitute()` 方法更加合适——如果数据缺失，它会直接将占位符原样保留。

```
>>> 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.'
```

Template 的子类可以自定义定界符。例如，以下是某个照片浏览器的批量重命名功能，采用了百分号作为日期、照片序号和照片格式的占位符:

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
```

(下页继续)

(续上页)

```
>>> fmt = raw_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
```

模板的另一个应用是将程序逻辑与多样的格式化输出细节分离开来。这使得对 XML 文件、纯文本报表和 HTML 网络报表使用自定义模板成为可能。

11.3 使用二进制数据记录格式

struct 模块提供了 pack() 和 unpack() 函数，用于处理不定长度的二进制记录格式。下面的例子展示了在不使用 zipfile 模块的情况下，如何循环遍历一个 ZIP 文件的所有头信息。Pack 代码 "H" 和 "I" 分别代表两字节和四字节无符号整数。"<" 代表它们是标准尺寸的小尾型字节序:

```
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
```

11.4 多线程

线程是一种对于非顺序依赖的多个任务进行解耦的技术。多线程可以提高应用的响应效率，当接收用户输入的同时，保持其他任务在后台运行。一个有关的应用场景是，将 I/O 和计算运行在两个并行的线程中。

以下代码展示了高阶的 threading 模块如何在后台运行任务，且不影响主程序的继续运行:

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
```

(下页继续)

(续上页)

```
        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.'
```

多线程应用面临的主要挑战是，相互协调的多个线程之间需要共享数据或其他资源。为此，threading 模块提供了多个同步操作原语，包括线程锁、事件、条件变量和信号量。

While those tools are powerful, minor design errors can result in problems that are difficult to reproduce. So, the preferred approach to task coordination is to concentrate all access to a resource in a single thread and then use the `Queue` module to feed that thread with requests from other threads. Applications using `Queue.Queue` objects for inter-thread communication and coordination are easier to design, more readable, and more reliable.

11.5 日志记录

`logging` 模块提供功能齐全且灵活的日志记录系统。在最简单的情况下，日志消息被发送到文件或 `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')
```

这会产生以下输出:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

默认情况下，informational 和 debugging 消息被压制，输出会发送到标准错误流。其他输出选项包括将消息转发到电子邮件，数据报，套接字或 HTTP 服务器。新的过滤器可以根据消息优先级选择不同的路由方式：`DEBUG`，`INFO`，`WARNING`，`ERROR`，和 `CRITICAL`。

日志系统可以直接从 Python 配置，也可以从用户配置文件加载，以便自定义日志记录而无需更改应用程序。

11.6 弱引用

Python 会自动进行内存管理（对大多数对象进行引用计数并使用*garbage collection* 来清除循环引用）。当某个对象的最后一个引用被移除后不久就会释放其所占用的内存。

此方式对大多数应用来说都适用，但偶尔也必须在对象持续被其他对象所使用时跟踪它们。不幸的是，跟踪它们将创建一个会令其永久化的引用。weakref 模块提供的工具可以不必创建引用就能跟踪对象。当对象不再需要时，它将自动从一个弱引用表中被移除，并为弱引用对象触发一个回调。典型应用包括对创建开销较大的对象进行缓存:

```
>>> 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:/python26/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7 用于操作列表的工具

许多对于数据结构的需求可以通过内置列表类型来满足。但是，有时也会需要具有不同效费比的替代实现。

array 模块提供了一种 array() 对象，它类似于列表，但只能存储类型一致的数据且存储密集更高。下面的例子演示了一个以两个字节为存储单元的无符号二进制数值的数组 (类型码为 "H")，而对于普通列表来说，每个条目存储为标准 Python 的 int 对象通常要占用 16 个字节:

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

collections 模块提供了一种 deque() 对象，它类似于列表，但从左端添加和弹出的速度较快，而在中间查找的速度较慢。此种对象适用于实现队列和广度优先树搜索:

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

(下页继续)

(续上页)

```
>>> 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)
```

在替代的列表实现以外，标准库也提供了其他工具，例如 `bisect` 模块具有用于操作排序列表的函数:

```
>>> 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')]
```

`heapq` 模块提供了基于常规列表来实现堆的函数。最小值的条目总是保持在位置零。这对于需要重复访问最小元素而不希望运行完整列表排序的应用来说非常有用:

```
>>> 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]
```

11.8 十进制浮点运算

`decimal` 模块提供了一种 `Decimal` 数据类型用于十进制浮点运算。相比内置的 `float` 二进制浮点实现，该类特别适用于

- 财务应用和其他需要精确十进制表示的用途，
- 控制精度，
- 控制四舍五入以满足法律或监管要求，
- 跟踪有效小数位，或
- 用户期望结果与手工完成的计算相匹配的应用程序。

例如，使用十进制浮点和二进制浮点数计算 70 美分手机和 5% 税的总费用，会产生的不同结果。如果结果四舍五入到最接近的分数差异会更大:

```
>>> from decimal import *
>>> x = Decimal('0.70') * Decimal('1.05')
>>> x
Decimal('0.7350')
>>> x.quantize(Decimal('0.01'))  # round to nearest cent
Decimal('0.74')
>>> round(.70 * 1.05, 2)         # same calculation with floats
0.73
```

Decimal 表示的结果会保留尾部的零，并根据具有两个有效位的被乘数自动推出四个有效位。Decimal 可以模拟手工运算来避免当二进制浮点数无法精确表示十进制数时会导致的问题。

精确表示特性使得 Decimal 类能够执行对于二进制浮点数来说不适用的模运算和相等性检测:

```
>>> 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 模块提供了运算所需要的足够精度:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```

CHAPTER 12

接下来？

阅读本教程可能会增强您对使用 Python 的兴趣 - 您应该热衷于应用 Python 来解决您的实际问题。你应该去哪里了解更多?

本教程是 Python 文档集的一部分。其他文档：

- library-index:

  您应该浏览本手册，该手册提供了有关标准库中的类型，功能和模块的完整（尽管简洁）参考资料。标准的 Python 发行版包含 很多的附加代码。有些模块可以读取 Unix 邮箱，通过 HTTP 检索文档，生成随机数，解析命令行选项，编写 CGI 程序，压缩数据以及许多其他任务。浏览标准库参考可以了解更多可用的内容。

- install-index explains how to install external modules written by other Python users.
- reference-index: Python 的语法和语义的详细解释。尽管阅读完非常繁重，但作为语言本身的完整指南是有用的。

更多 Python 资源：

- https://www.python.org ： 主要的 Python 网站。它包含代码，文档以及指向 Web 上与 Python 相关的页面的链接。该网站世界很多地区都有镜像，如欧洲，日本和澳大利亚；镜像可能比主站点更快，具体取决于您的地理位置。
- https://docs.python.org ： 快速访问 Python 的文档。
- https://pypi.org: The Python Package Index, previously also nicknamed the Cheese Shop, is an index of user-created Python modules that are available for download. Once you begin releasing code, you can register it here so that others can find it.
- https://code.activestate.com/recipes/langs/python/ ： Python Cookbook 是一个相当大的代码示例集，更多的模块和有用的脚本。特别值得注意的贡献收集在一本名为 Python Cookbook（O’ Reilly & Associates，ISBN 0-596-00797-3）的书中。

For Python-related questions and problem reports, you can post to the newsgroup `comp.lang.python`, or send them to the mailing list at python-list@python.org. The newsgroup and mailing list are gatewayed, so messages posted to one will automatically be forwarded to the other. There are around 120 postings a day (with peaks up to several hundred), asking (and answering) questions, suggesting new features, and announcing new modules. Before posting, be sure to check the list of Frequently Asked Questions (also called the FAQ). Mailing list archives are available at https://mail.python.

org/pipermail/. The FAQ answers many of the questions that come up again and again, and may already contain the solution for your problem.

CHAPTER 13

交互式编辑和编辑历史

Some versions of the Python interpreter support editing of the current input line and history substitution, similar to facilities found in the Korn shell and the GNU Bash shell. This is implemented using the GNU Readline library, which supports Emacs-style and vi-style editing. This library has its own documentation which I won't duplicate here; however, the basics are easily explained. The interactive editing and history described here are optionally available in the Unix and Cygwin versions of the interpreter.

This chapter does *not* document the editing facilities of Mark Hammond's PythonWin package or the Tk-based environment, IDLE, distributed with Python. The command line history recall which operates within DOS boxes on NT and some other DOS and Windows flavors is yet another beast.

13.1 Line Editing

If supported, input line editing is active whenever the interpreter prints a primary or secondary prompt. The current line can be edited using the conventional Emacs control characters. The most important of these are: `C-A` (Control-A) moves the cursor to the beginning of the line, `C-E` to the end, `C-B` moves it one position to the left, `C-F` to the right. Backspace erases the character to the left of the cursor, `C-D` the character to its right. `C-K` kills (erases) the rest of the line to the right of the cursor, `C-Y` yanks back the last killed string. `C-underscore` undoes the last change you made; it can be repeated for cumulative effect.

13.2 History Substitution

History substitution works as follows. All non-empty input lines issued are saved in a history buffer, and when a new prompt is given you are positioned on a new line at the bottom of this buffer. `C-P` moves one line up (back) in the history buffer, `C-N` moves one down. Any line in the history buffer can be edited; an asterisk appears in front of the prompt to mark a line as modified. Pressing the `Return` key passes the current line to the interpreter. `C-R` starts an incremental reverse search; `C-S` starts a forward search.

13.3 Key Bindings

The key bindings and some other parameters of the Readline library can be customized by placing commands in an initialization file called `~/.inputrc`. Key bindings have the form

```
key-name: function-name
```

or

```
"string": function-name
```

and options can be set with

```
set option-name value
```

For example:

```
# I prefer vi-style editing:
set editing-mode vi

# Edit using a single line:
set horizontal-scroll-mode On

# Rebind some keys:
Meta-h: backward-kill-word
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

Note that the default binding for `Tab` in Python is to insert a `Tab` character instead of Readline’ s default filename completion function. If you insist, you can override this by putting

```
Tab: complete
```

in your `~/.inputrc`. (Of course, this makes it harder to type indented continuation lines if you’ re accustomed to using `Tab` for that purpose.)

Automatic completion of variable and module names is optionally available. To enable it in the interpreter’ s interactive mode, add the following to your startup file:[1]

```
import rlcompleter, readline
readline.parse_and_bind('tab: complete')
```

This binds the `Tab` key to the completion function, so hitting the `Tab` key twice suggests completions; it looks at Python statement names, the current local variables, and the available module names. For dotted expressions such as `string.a`, it will evaluate the expression up to the final `'.'` and then suggest completions from the attributes of the resulting object. Note that this may execute application-defined code if an object with a `__getattr__()` method is part of the expression.

A more capable startup file might look like this example. Note that this deletes the names it creates once they are no longer needed; this is done since the startup file is executed in the same namespace as the interactive commands, and removing the names avoids creating side effects in the interactive environment. You may find it convenient to keep some of the imported modules, such as `os`, which turn out to be needed in most sessions with the interpreter.

[1] Python will execute the contents of a file identified by the `PYTHONSTARTUP` environment variable when you start an interactive interpreter. To customize Python even for non-interactive mode, see 定制模块.

```
# Add auto-completion and a stored history file of commands to your Python
# interactive interpreter. Requires Python 2.0+, readline. Autocomplete is
# bound to the Esc key by default (you can change it - see readline docs).
#
# Store the file in ~/.pystartup, and set an environment variable to point
# to it:  "export PYTHONSTARTUP=~/.pystartup" in bash.

import atexit
import os
import readline
import rlcompleter

historyPath = os.path.expanduser("~/.pyhistory")

def save_history(historyPath=historyPath):
    import readline
    readline.write_history_file(historyPath)

if os.path.exists(historyPath):
    readline.read_history_file(historyPath)

atexit.register(save_history)
del os, atexit, readline, rlcompleter, save_history, historyPath
```

13.4 默认交互式解释器的替代品

Python 解释器与早期版本的相比，向前迈进了一大步；无论怎样，还有些希望的功能：如果能在编辑连续行时建议缩进（解析器知道接下来是否需要缩进符号），那将很棒。补全机制可以使用解释器的符号表。有命令去检查（甚至建议）括号，引号以及其他符号是否匹配。

一个可选的增强型交互式解释器是 IPython，它已经存在了有一段时间，它具有 tab 补全，探索对象和高级历史记录管理功能。它还可以彻底定制并嵌入到其他应用程序中。另一个相似的增强型交互式环境是 bpython。

CHAPTER 14

浮点算术：争议和限制

浮点数在计算机硬件中表示为以 2 为基数（二进制）的小数。举例而言，十进制的小数

```
0.125
```

等于 1/10 + 2/100 + 5/1000 ，同理，二进制的小数

```
0.001
```

等于 0/2 + 0/4 + 1/8。这两个小数具有相同的值，唯一真正的区别是第一个是以 10 为基数的小数表示法，第二个则是 2 为基数。

不幸的是，大多数的十进制小数都不能精确地表示为二进制小数。这导致在大多数情况下，你输入的十进制浮点数都只能近似地以二进制浮点数形式储存在计算机中。

用十进制来理解这个问题显得更加容易一些。考虑分数 1/3 。我们可以得到它在十进制下的一个近似值

```
0.3
```

或者，更近似的，：

```
0.33
```

或者，更近似的，：

```
0.333
```

以此类推。结果是无论你写下多少的数字，它都永远不会等于 1/3 ，只是更加更加地接近 1/3 。

同样的道理，无论你使用多少位以 2 为基数的数码，十进制的 0.1 都无法精确地表示为一个以 2 为基数的小数。在以 2 为基数的情况下，1/10 是一个无限循环小数

```
0.0001100110011001100110011001100110011001100110011...
```

Stop at any finite number of bits, and you get an approximation.

On a typical machine running Python, there are 53 bits of precision available for a Python float, so the value stored internally when you enter the decimal number `0.1` is the binary fraction

```
0.0001100110011001100110011001100110011001100110011010
```

which is close to, but not exactly equal to, 1/10.

It' s easy to forget that the stored value is an approximation to the original decimal fraction, because of the way that floats are displayed at the interpreter prompt. Python only prints a decimal approximation to the true decimal value of the binary approximation stored by the machine. If Python were to print the true decimal value of the binary approximation stored for 0.1, it would have to display

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

这比大多数人认为有用的数字更多，因此 Python 通过显示舍入值来保持可管理的位数

```
>>> 0.1
0.1
```

It' s important to realize that this is, in a real sense, an illusion: the value in the machine is not exactly 1/10, you' re simply rounding the *display* of the true machine value. This fact becomes apparent as soon as you try to do arithmetic with these values

```
>>> 0.1 + 0.2
0.30000000000000004
```

请注意这种情况是二进制浮点数的本质特性：它不是 Python 的错误，也不是你代码中的错误。你会在所有支持你的硬件中的浮点运算的语言中发现同样的情况（虽然某些语言在默认状态或所有输出模块下都不会 显示这种差异）。

Other surprises follow from this one. For example, if you try to round the value 2.675 to two decimal places, you get this

```
>>> round(2.675, 2)
2.67
```

The documentation for the built-in `round()` function says that it rounds to the nearest value, rounding ties away from zero. Since the decimal fraction 2.675 is exactly halfway between 2.67 and 2.68, you might expect the result here to be (a binary approximation to) 2.68. It' s not, because when the decimal string `2.675` is converted to a binary floating-point number, it' s again replaced with a binary approximation, whose exact value is

```
2.67499999999999982236431605997495353221893310546875
```

Since this approximation is slightly closer to 2.67 than to 2.68, it' s rounded down.

If you' re in a situation where you care which way your decimal halfway-cases are rounded, you should consider using the `decimal` module. Incidentally, the `decimal` module also provides a nice way to "see" the exact value that' s stored in any particular Python float

```
>>> from decimal import Decimal
>>> Decimal(2.675)
Decimal('2.67499999999999982236431605997495353221893310546875')
```

Another consequence is that since 0.1 is not exactly 1/10, summing ten values of 0.1 may not yield exactly 1.0, either:

```
>>> sum = 0.0
>>> for i in range(10):
```

(下页继续)

(续上页)

```
...     sum += 0.1
...
>>> sum
0.9999999999999999
```

二进制浮点运算会造成许多这样的“意外”。有关“0.1”的问题会在下面的“表示性错误”一节中更详细地描述。请参阅 浮点数的危险性 一文了解有关其他常见意外现象的更详细介绍。

As that says near the end, “there are no easy answers.” Still, don’t be unduly wary of floating-point! The errors in Python float operations are inherited from the floating-point hardware, and on most machines are on the order of no more than 1 part in 2**53 per operation. That’s more than adequate for most tasks, but you do need to keep in mind that it’s not decimal arithmetic, and that every float operation can suffer a new rounding error.

While pathological cases do exist, for most casual use of floating-point arithmetic you’ll see the result you expect in the end if you simply round the display of your final results to the number of decimal digits you expect. For fine control over how a float is displayed see the `str.format()` method’s format specifiers in formatstrings.

14.1 表示性错误

本小节将详细解释“0.1”的例子，并说明你可以怎样亲自对此类情况进行精确分析。假定前提是已基本熟悉二进制浮点表示法。

Representation error refers to the fact that some (most, actually) decimal fractions cannot be represented exactly as binary (base 2) fractions. This is the chief reason why Python (or Perl, C, C++, Java, Fortran, and many others) often won’t display the exact decimal number you expect:

```
>>> 0.1 + 0.2
0.30000000000000004
```

Why is that? 1/10 and 2/10 are not exactly representable as a binary fraction. Almost all machines today (July 2010) use IEEE-754 floating point arithmetic, and almost all platforms map Python floats to IEEE-754 “double precision”. 754 doubles contain 53 bits of precision, so on input the computer strives to convert 0.1 to the closest fraction it can of the form *J*/2***N* where *J* is an integer containing exactly 53 bits. Rewriting

```
1 / 10 ~= J / (2**N)
```

写为

```
J ~= 2**N / 10
```

并且由于 *J* 恰好有 53 位 (即 >= `2**52` 但 < `2**53`)，*N* 的最佳值为 56:

```
>>> 2**52
4503599627370496
>>> 2**53
9007199254740992
>>> 2**56/10
7205759403792793
```

That is, 56 is the only value for *N* that leaves *J* with exactly 53 bits. The best possible value for *J* is then that quotient rounded:

```
>>> q, r = divmod(2**56, 10)
>>> r
6
```

由于余数超过 10 的一半，最佳近似值可通过四舍五入获得:

```
>>> q+1
7205759403792794
```

Therefore the best possible approximation to 1/10 in 754 double precision is that over 2**56, or

```
7205759403792794 / 72057594037927936
```

请注意由于我们做了向上舍入，这个结果实际上略大于 1/10；如果我们没有向上舍入，则商将会略小于 1/10。但无论如何它都不会是 *精确的* 1/10！

因此计算永远不会"看到"1/10：它实际看到的就是上面所给出的小数，它所能达到的最佳 754 双精度近似值:

```
>>> .1 * 2**56
7205759403792794.0
```

If we multiply that fraction by 10**30, we can see the (truncated) value of its 30 most significant decimal digits:

```
>>> 7205759403792794 * 10**30 // 2**56
100000000000000005551115123125L
```

meaning that the exact number stored in the computer is approximately equal to the decimal value 0.100000000000000005551115123125. In versions prior to Python 2.7 and Python 3.1, Python rounded this value to 17 significant digits, giving '0.10000000000000001'. In current versions, Python displays a value based on the shortest decimal fraction that rounds correctly back to the true binary value, resulting simply in '0.1'.

CHAPTER 15

附录

15.1 交互模式

15.1.1 错误处理

当发生错误时，解释器会打印错误信息和错误堆栈。在交互模式下，将返回到主命令提示符；如果输入内容来自文件，在打印错误堆栈之后，程序会以非零状态退出。(这里所说的错误不包括 `try` 语句中由 `except` 所捕获的异常。) 有些错误是无条件致命的，会导致程序以非零状态退出；比如内部逻辑矛盾或内存耗尽。所有错误信息都会被写入标准错误流；而命令的正常输出则被写入标准输出流。

将中断字符（通常为 `Control-C` 或 `Delete` ）键入主要或辅助提示会取消输入并返回主提示符。[1] 在执行命令时键入中断引发的 `KeyboardInterrupt` 异常，可以由 `try` 语句处理。

15.1.2 可执行的 Python 脚本

在 BSD 等类 Unix 系统上，Python 脚本可以直接执行，就像 shell 脚本一样，第一行添加:

```
#!/usr/bin/env python
```

（假设解释器位于用户的 `PATH` ）脚本的开头，并将文件设置为可执行。`#!` 必须是文件的前两个字符。在某些平台上，第一行必须以 Unix 样式的行结尾（`'\n'`）结束，而不是以 Windows（`'\r\n'`）行结尾。请注意，散列或磅字符 `'#'` 在 Python 中代表注释开始。

可以使用 **`chmod`** 命令为脚本提供可执行模式或权限。

```
$ chmod +x myscript.py
```

在 Windows 系统上，没有 “可执行模式” 的概念。Python 安装程序自动将 `.py` 文件与 `python.exe` 相关联，这样双击 Python 文件就会将其作为脚本运行。扩展也可以是 `.pyw` ，在这种情况下，会隐藏通常出现的控制台窗口。

[1] GNU Readline 包的问题可能会阻止这种情况。

15.1.3 交互式启动文件

当您以交互方式使用 Python 时，每次启动解释器时都会执行一些标准命令，这通常很方便。您可以通过将名为 PYTHONSTARTUP 的环境变量设置为包含启动命令的文件名来实现。这类似于 Unix shell 的 .profile 功能。

This file is only read in interactive sessions, not when Python reads commands from a script, and not when /dev/tty is given as the explicit source of commands (which otherwise behaves like an interactive session). It is executed in the same namespace where interactive commands are executed, so that objects that it defines or imports can be used without qualification in the interactive session. You can also change the prompts sys.ps1 and sys.ps2 in this file.

如果你想从当前目录中读取一个额外的启动文件，你可以使用像 if os.path.isfile('.pythonrc.py'): exec(open('.pythonrc.py').read()) 这样的代码在全局启动文件中对它进行编程。如果要在脚本中使用启动文件，则必须在脚本中显式执行此操作:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    with open(filename) as fobj:
        startup_file = fobj.read()
    exec(startup_file)
```

15.1.4 定制模块

Python 提供了两个钩子来让你自定义它：sitecustomize 和 usercustomize。要查看其工作原理，首先需要找到用户 site-packages 目录的位置。启动 Python 并运行此代码:

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python2.7/site-packages'
```

现在，您可以在该目录中创建一个名为 usercustomize.py 的文件，并将所需内容放入其中。它会影响 Python 的每次启动，除非它以 -s 选项启动，以禁用自动导入。

sitecustomize 以相同的方式工作，但通常由计算机管理员在全局 site-packages 目录中创建，并在 usercustomize 之前被导入。有关详情请参阅 site 模块的文档。

备注

APPENDIX A

术语对照表

>>> 交互式终端中默认的 Python 提示符。往往会显示于能以交互方式在解释器里执行的样例代码之前。

... The default Python prompt of the interactive shell when entering code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.

2to3 一个将 Python 2.x 代码转换为 Python 3.x 代码的工具，能够处理大部分通过解析源码并遍历解析树可检测到的不兼容问题。

2to3 包含在标准库中，模块名为 `lib2to3`；并提供一个独立入口点 `Tools/scripts/2to3`。参见 2to3-reference。

abstract base class –抽象基类 Abstract base classes complement *duck-typing* by providing a way to define interfaces when other techniques like `hasattr()` would be clumsy or subtly wrong (for example with magic methods). ABCs introduce virtual subclasses, which are classes that don' t inherit from a class but are still recognized by `isinstance()` and `issubclass()`; see the `abc` module documentation. Python comes with many built-in ABCs for data structures (in the `collections` module), numbers (in the `numbers` module), and streams (in the `io` module). You can create your own ABCs with the `abc` module.

argument –参数 A value passed to a *function* (or *method*) when calling the function. There are two types of arguments:

- 关键字参数: 在函数调用中前面带有标识符（例如 `name=`）或者作为包含在前面带有 `**` 的字典里的值传入。举例来说，`3` 和 `5` 在以下对 `complex()` 的调用中均属于关键字参数:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- 位置参数: 不属于关键字参数的参数。位置参数可出现于参数列表的开头以及/或者作为前面带有 `*` 的*iterable* 里的元素被传入。举例来说，`3` 和 `5` 在以下调用中均属于位置参数:

```
complex(3, 5)
complex(*(3, 5))
```

参数会被赋值给函数体中对应的局部变量。有关赋值规则参见 calls 一节。根据语法，任何表达式都可用来表示一个参数；最终算出的值会被赋给对应的局部变量。

See also the *parameter* glossary entry and the FAQ question on the difference between arguments and parameters.

attribute –属性 关联到一个对象的值，可以使用点号表达式通过其名称来引用。例如，如果一个对象 *o* 具有一个属性 *a*，就可以用 *o.a* 来引用它。

BDFL Benevolent Dictator For Life, a.k.a. Guido van Rossum, Python' s creator.

bytes-like object –字节类对象 An object that supports the buffer protocol, like `str`, `bytearray` or `memoryview`. Bytes-like objects can be used for various operations that expect binary data, such as compression, saving to a binary file or sending over a socket. Some operations need the binary data to be mutable, in which case not all bytes-like objects can apply.

bytecode –字节码 Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in `.pyc` and `.pyo` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This "intermediate language" is said to run on a *virtual machine* that executes the machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.

字节码指令列表可以在 dis 模块的文档中查看。

class –类 用来创建用户定义对象的模板。类定义通常包含对该类的实例进行操作的方法定义。

classic class Any class which does not inherit from `object`. See *new-style class*. Classic classes have been removed in Python 3.

coercion –强制类型转换 The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer `3`, but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Coercion between two operands can be performed with the `coerce` built-in function; thus, `3+4.5` is equivalent to calling `operator.add(*coerce(3, 4.5))` and results in `operator.add(3.0, 4.5)`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.

complex number –复数 对普通实数系统的扩展，其中所有数字都被表示为一个实部和一个虚部的和。虚数是虚数单位（`-1` 的平方根）的实倍数，通常在数学中写为 `i`，在工程学中写为 `j`。Python 内置了对复数的支持，采用工程学标记方式；虚部带有一个 `j` 后缀，例如 `3+1j`。如果需要 `math` 模块内对象的对应复数版本，请使用 `cmath`，复数的使用是一个比较高级的数学特性。如果你感觉没有必要，忽略它们也几乎不会有任何问题。

context manager –上下文管理器 在 `with` 语句中使用，通过定义 `__enter__()` 和 `__exit__()` 方法来控制环境状态的对象。参见 **PEP 343**。

CPython Python 编程语言的规范实现，在 python.org 上发布。" CPython" 一词用于在必要时将此实现与其他实现例如 Jython 或 IronPython 相区别。

decorator –装饰器 返回值为另一个函数的函数，通常使用 `@wrapper` 语法形式来进行函数变换。装饰器的常见例子包括 `classmethod()` 和 `staticmethod()`。

装饰器语法只是一种语法糖，以下两个函数定义在语义上完全等价:

```
def f(...):
    ...
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

同的样概念也适用于类，但通常较少这样使用。有关装饰器的详情可参见 函数定义和 类定义的文档。

descriptor –描述器 Any *new-style* object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

有关描述符的方法的详情可参看 descriptors。

dictionary –字典 An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

dictionary view –字典视图 The objects returned from `dict.viewkeys()`, `dict.viewvalues()`, and `dict.viewitems()` are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use `list(dictview)`. See dict-views.

docstring –文档字符串 作为类、函数或模块之内的第一个表达式出现的字符串字面值。它在代码执行时会被忽略，但会被解释器识别并放入所在类、函数或模块的 `__doc__` 属性中。由于它可用于代码内省，因此是对象存放文档的规范位置。

duck-typing –鸭子类型 指一种编程风格，它并不依靠查找对象类型来确定其是否具有正确的接口，而是直接调用或使用其方法或属性（"看起来像鸭子，叫起来也像鸭子，那么肯定就是鸭子。"）由于强调接口而非特定类型，设计良好的代码可通过允许多态替代来提升灵活性。鸭子类型避免使用 `type()` 或 `isinstance()` 检测。(但要注意鸭子类型可以使用抽象基类 作为补充。) 而往往会采用 `hasattr()` 检测或是*EAFP* 编程。

EAFP "求原谅比求许可更容易" 的英文缩写。这种 Python 常用代码编写风格会假定所需的键或属性存在，并在假定错误时捕获异常。这种简洁快速风格的特点就是大量运用 `try` 和 `except` 语句。于其相对的则是所谓*LBYL* 风格，常见于 C 等许多其他语言。

expression –表达式 A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also *statement*s which cannot be used as expressions, such as `print` or `if`. Assignments are also statements, not expressions.

extension module –扩展模块 以 C 或 C++ 编写的模块，使用 Python 的 C API 来与语言核心以及用户代码进行交互。

file object –文件对象 对外提供面向文件 API 以使用下层资源的对象（带有 `read()` 或 `write()` 这样的方法）。根据其创建方式的不同，文件对象可以处理对真实磁盘文件，对其他类型存储，或是对通讯设备的访问（例如标准输入/输出、内存缓冲区、套接字、管道等等）。文件对象也被称为 文件类对象或 流。

There are actually three categories of file objects: raw binary files, buffered binary files and text files. Their interfaces are defined in the `io` module. The canonical way to create a file object is by using the `open()` function.

file-like object –文件类对象 *file object* 的同义词。

finder –查找器 An object that tries to find the *loader* for a module. It must implement a method named `find_module()`. See **PEP 302** for details.

floor division –向下取整除法 向下舍入到最接近的整数的数学除法。向下取整除法的运算符是 `//` 。例如，表达式 `11 // 4` 的计算结果是 `2` ，而与之相反的是浮点数的真正除法返回 `2.75` 。注意 `(-11) // 4` 会返回 `-3` 因为这是 `-2.75` 向下舍入得到的结果。见 **PEP 238** 。

function –函数 可以向调用者返回某个值的一组语句。还可以向其传入零个或多个参数 并在函数体执行中被使用。另见*parameter*, *method* 和 function 等节。

__future__ A pseudo-module which programmers can use to enable new language features which are not compatible with the current interpreter. For example, the expression `11/4` currently evaluates to `2`. If the module in which it is executed had enabled *true division* by executing:

```
from __future__ import division
```

the expression 11/4 would evaluate to 2.75. By importing the `__future__` module and evaluating its variables, you can see when a new feature was first added to the language and when it will become the default:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection –垃圾回收 The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles.

generator –生成器 A function which returns an iterator. It looks like a normal function except that it contains `yield` statements for producing a series of values usable in a for-loop or that can be retrieved one at a time with the `next()` function. Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

generator expression –生成器表达式 An expression that returns an iterator. It looks like a normal expression followed by a `for` expression defining a loop variable, range, and an optional `if` expression. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

GIL 参见*global interpreter lock*。

global interpreter lock –全局解释器锁 *CPython* 解释器所采用的一种机制，它确保同一时刻只有一个线程在执行 Python *bytecode*。此机制通过设置对象模型（包括 `dict` 等重要内置类型）针对并发访问的隐式安全简化了 CPython 实现。给整个解释器加锁使得解释器多线程运行更方便，其代价则是牺牲了在多处理器上的并行性。

不过，某些标准库或第三方库的扩展模块被设计为在执行计算密集型任务如压缩或哈希时释放 GIL。此外，在执行 I/O 操作时也总是会释放 GIL。

创建一个（以更精细粒度来锁定共享数据的）“自由线程”解释器的努力从未获得成功，因为这会牺牲在普通单处理器情况下的性能。据信克服这种性能问题的措施将导致实现变得更复杂，从而更难以维护。

hashable –可哈希 An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` or `__cmp__()` method). Hashable objects which compare equal must have the same hash value.

可哈希性使得对象能够作为字典键或集合成员使用，因为这些数据结构要在内部使用哈希值。

All of Python’s immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. Objects which are instances of user-defined classes are hashable by default; they all compare unequal (except with themselves), and their hash value is derived from their `id()`.

IDLE Python 的 IDE，“集成开发与学习环境”的英文缩写。是 Python 标准发行版附带的基本编程器和解释器环境。

immutable –不可变 具有固定值的对象。不可变对象包括数字、字符串和元组。这样的对象不能被改变。如果必须存储一个不同的值，则必须创建新的对象。它们在需要常量哈希值的地方起着重要作用，例如作为字典中的键。

integer division Mathematical division discarding any remainder. For example, the expression 11/4 currently evaluates to 2 in contrast to the 2.75 returned by float division. Also called *floor division*. When dividing two integers the outcome will always be another integer (having the floor function applied to it). However, if one of the operands is another numeric type (such as a `float`), the result will be coerced (see *coercion*) to a common type. For example, an integer divided by a float will result in a float value, possibly with a decimal fraction. Integer division can be forced by using the `//` operator instead of the `/` operator. See also *__future__*.

importing –导入 令一个模块中的 Python 代码能为另一个模块中的 Python 代码所使用的过程。

importer –导入器 查找并加载模块的对象；此对象既属于*finder* 又属于*loader*。

interactive –交互 Python 带有一个交互式解释器，即你可以在解释器提示符后输入语句和表达式，立即执行并查看其结果。只需不带参数地启动 `python` 命令（也可以在你的计算机开始菜单中选择相应菜单项）。在测试新想法或检验模块和包的时候用这种方式会非常方便（请记得使用 `help(x)`）。

interpreted –解释型 Python 一是种解释型语言，与之相对的是编译型语言，虽然两者的区别由于字节码编译器的存在而会有所模糊。这意味着源文件可以直接运行而不必显式地创建可执行文件再运行。解释型语言通常具有比编译型语言更短的开发/调试周期，但是其程序往往运行得更慢。参见*interactive*。

iterable –可迭代对象 An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict` and `file` and objects of any classes you define with an `__iter__()` or `__getitem__()` method. Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ⋯). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

iterator –迭代器 An object representing a stream of data. Repeated calls to the iterator’s `next()` method return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `next()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

更多信息可查看 typeiter。

key function –键函数 键函数或称整理函数，是能够返回用于排序或排位的值的可调用对象。例如，`locale.strxfrm()` 可用于生成一个符合特定区域排序约定的排序键。

A number of tools in Python accept key functions to control how elements are ordered or grouped. They include `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.nsmallest()`, `heapq.nlargest()`, and `itertools.groupby()`.

There are several ways to create a key function. For example. the `str.lower()` method can serve as a key function for case insensitive sorts. Alternatively, an ad-hoc key function can be built from a `lambda` expression such as `lambda r: (r[0], r[2])`. Also, the `operator` module provides three key function constructors: `attrgetter()`, `itemgetter()`, and `methodcaller()`. See the Sorting HOW TO for examples of how to create and use key functions.

keyword argument –关键字参数 参见*argument*。

lambda 由一个单独*expression* 构成的匿名内联函数，表达式会在调用时被求值。创建 lambda 函数的句法为 `lambda [parameters]: expression`

LBYL "先查看后跳跃" 的英文缩写。这种代码编写风格会在进行调用或查找之前显式地检查前提条件。此风格与*EAFP* 方式恰成对比，其特点是大量使用 `if` 语句。

在多线程环境中，LBYL 方式会导致 "查看" 和 "跳跃" 之间发生条件竞争风险。例如，以下代码 `if key in mapping: return mapping[key]` 可能由于在检查操作之后其他线程从 *mapping* 中移除了 *key* 而出错。这种问题可通过加锁或使用 EAFP 方式来解决。

list –列表 Python 内置的一种*sequence*。虽然名为列表，但更类似于其他语言中的数组而非链接列表，因为访问元素的时间复杂度为 O(1)。

list comprehension –列表推导式 A compact way to process all or part of the elements in a sequence and return a list with the results. `result = ["0x%02x" % x for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed.

loader –加载器 An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a *finder*. See **PEP 302** for details.

magic method –魔术方法 *special method* 的非正式同义词。

mapping –映射 A container object that supports arbitrary key lookups and implements the methods specified in the `Mapping` or `MutableMapping` abstract base classes. Examples include `dict`, `collections.defaultdict`, `collections.OrderedDict` and `collections.Counter`.

metaclass –元类 一种用于创建类的类。类定义包含类名、类字典和基类列表。元类负责接受上述三个参数并创建相应的类。大部分面向对象的编程语言都会提供一个默认实现。Python 的特别之处在于可以创建自定义元类。大部分用户永远不需要这个工具，但当需要出现时，元类可提供强大而优雅的解决方案。它们已被用于记录属性访问日志、添加线程安全性、跟踪对象创建、实现单例，以及其他许多任务。

更多详情参见 metaclasses。

method 方法 在类内部定义的函数。如果作为该类的实例的一个属性来调用，方法将会获取实例对象作为其第一个*argument* (通常命名为 `self`)。参见*function* 和*nested scope*。

method resolution order –方法解析顺序 方法解析顺序就是在查找成员时搜索全部基类所用的先后顺序。请查看 Python 2.3 方法解析顺序 了解自 2.3 版起 Python 解析器所用相关算法的详情。

module 模块 此对象是 Python 代码的一种组织单位。各模块具有独立的命名空间，可包含任意 Python 对象。模块可通过*importing* 操作被加载到 Python 中。

另见*package*。

MRO 参见*method resolution order*。

mutable –可变 可变对象可以在其 `id()` 保持固定的情况下改变其取值。另请参见*immutable*。

named tuple –具名元组 Any tuple-like class whose indexable elements are also accessible using named attributes (for example, `time.localtime()` returns a tuple-like object where the *year* is accessible either with an index such as `t[0]` or with a named attribute like `t.tm_year`).

A named tuple can be a built-in type such as `time.struct_time`, or it can be created with a regular class definition. A full featured named tuple can also be created with the factory function `collections.namedtuple()`. The latter approach automatically provides extra features such as a self-documenting representation like `Employee(name='jones', title='programmer')`.

namespace –命名空间 The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `__builtin__.open()` and `os.open()` are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.izip()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.

nested scope –嵌套作用域 The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes work only for reference and not for assignment which will always write to the innermost scope. In contrast, local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace.

new-style class –新式类 Any class which inherits from `object`. This includes all built-in types like `list` and `dict`. Only new-style classes can use Python’s newer, versatile features like `__slots__`, descriptors, properties, and `__getattribute__()`.

More information can be found in newstyle.

object –对象 任何具有状态（属性或值）以及预定义行为（方法）的数据。object 也是任何*new-style class* 的最顶层基类名。

package –包 一种可包含子模块或递归地包含子包的 Python *module*。从技术上说，包是带有 `__path__` 属性的 Python 模块。

parameter –形参 A named entity in a *function* (or method) definition that specifies an *argument* (or in some cases, arguments) that the function can accept. There are four types of parameters:

- *positional-or-keyword*：位置或关键字，指定一个可以作为位置参数 传入也可以作为关键字参数 传入的实参。这是默认的形参类型，例如下面的 *foo* 和 *bar*:

```
def func(foo, bar=None): ...
```

- *positional-only*：仅限位置，指定一个只能按位置传入的参数。Python 中没有定义仅限位置形参的语法。但是一些内置函数有仅限位置形参（比如 `abs()`）。
- *var-positional*：可变位置，指定可以提供由一个任意数量的位置参数构成的序列（附加在其他形参已接受的位置参数之后）。这种形参可通过在形参名称前加缀 `*` 来定义，例如下面的 *args*:

```
def func(*args, **kwargs): ...
```

- *var-keyword*：可变关键字，指定可以提供任意数量的关键字参数（附加在其他形参已接受的关键字参数之后）。这种形参可通过在形参名称前加缀 `**` 来定义，例如上面的 *kwargs*。

形参可以同时指定可选和必选参数，也可以为某些可选参数指定默认值。

See also the *argument* glossary entry, the FAQ question on the difference between arguments and parameters, and the function section.

PEP “Python 增强提议”的英文缩写。一个 PEP 就是一份设计文档，用来向 Python 社区提供信息，或描述一个 Python 的新增特性及其进度或环境。PEP 应当提供精确的技术规格和所提议特性的原理说明。

PEP 应被作为提出主要新特性建议、收集社区对特定问题反馈以及为必须加入 Python 的设计决策编写文档的首选机制。PEP 的作者有责任在社区内部建立共识，并应将不同意见也记入文档。

参见 **PEP 1**。

positional argument –位置参数 参见*argument*。

Python 3000 Python 3.x 发布路线的昵称（这个名字在版本 3 的发布还遥遥无期的时候就已出现了）。有时也被缩写为“Py3k”。

Pythonic 指一个思路或一段代码紧密遵循了 Python 语言最常用的风格和理念，而不是使用其他语言中通用的概念来实现代码。例如，Python 的常用风格是使用 `for` 语句循环来遍历一个可迭代对象中的所有元素。许多其他语言没有这样的结构，因此不熟悉 Python 的人有时会选择使用一个数字计数器:

```
for i in range(len(food)):
    print food[i]
```

而相应的更简洁更 Pythonic 的方法是这样的:

```
for piece in food:
    print piece
```

reference count –引用计数 对特定对象的引用的数量。当一个对象的引用计数降为零时，所分配资源将被释放。引用计数对 Python 代码来说通常是不可见的，但它是*CPython* 实现的一个关键元素。sys 模块定义了一个 getrefcount() 函数，程序员可调用它来返回特定对象的引用计数。

__slots__ A declaration inside a *new-style class* that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

sequence –序列 An *iterable* which supports efficient element access using integer indices via the __getitem__() special method and defines a len() method that returns the length of the sequence. Some built-in sequence types are list, str, tuple, and unicode. Note that dict also supports __getitem__() and __len__(), but is considered a mapping rather than a sequence because the lookups use arbitrary *immutable* keys rather than integers.

slice –切片 An object usually containing a portion of a *sequence*. A slice is created using the subscript notation, [] with colons between numbers when several are given, such as in variable_name[1:3:5]. The bracket (subscript) notation uses slice objects internally (or in older versions, __getslice__() and __setslice__()).

special method –特殊方法 一种由 Python 隐式调用的方法，用来对某个类型执行特定操作例如相加等等。这种方法的名称的首尾都为双下划线。特殊方法的文档参见 specialnames。

statement –语句 语句是程序段（一个代码“块”）的组成单位。一条语句可以是一个*expression* 或某个带有关键字的结构，例如 if、while 或 for。

struct sequence A tuple with named elements. Struct sequences expose an interface similiar to *named tuple* in that elements can be accessed either by index or as an attribute. However, they do not have any of the named tuple methods like _make() or _asdict(). Examples of struct sequences include sys.float_info and the return value of os.stat().

triple-quoted string –三引号字符串 首尾各带三个连续双引号（"）或者单引号（'）的字符串。它们在功能上与首尾各用一个引号标注的字符串没有什么不同，但是有多种用处。它们允许你在字符串内包含未经转义的单引号和双引号，并且可以跨越多行而无需使用连接符，在编写文档字符串时特别好用。

type –类型 类型决定一个 Python 对象属于什么种类；每个对象都具有一种类型。要知道对象的类型，可以访问它的 __class__ 属性，或是通过 type(obj) 来获取。

universal newlines –通用换行 A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention '\n', the Windows convention '\r\n', and the old Macintosh convention '\r'. See **PEP 278** and **PEP 3116**, as well as str.splitlines() for an additional use.

virtual environment –虚拟环境 一种采用协作式隔离的运行时环境，允许 Python 用户和应用程序在安装和升级 Python 分发包时不会干扰到同一系统上运行的其他 Python 应用程序的行为。

virtual machine –虚拟机 一台完全通过软件定义的计算机。Python 虚拟机可执行字节码编译器所生成的*bytecode*。

Zen of Python –Python 之禅 列出 Python 设计的原则与哲学，有助于理解与使用这种语言。查看其具体内容可在交互模式提示符中输入“import this”。

APPENDIX B

文档说明

这些文档生成自 reStructuredText 原文档，由 Sphinx （一个专门为 Python 文档写的文档生成器）创建。

本文档和它所用工具链的开发完全是由志愿者完成的，这和 Python 本身一样。如果您想参与进来，请阅读 reporting-bugs 了解如何参与。我们随时欢迎新的志愿者！

特别鸣谢：

- Fred L. Drake, Jr.，创造了用于早期 Python 文档的工具链，以及撰写了非常多的文档；
- Docutils 软件包 项目，创建了 reStructuredText 文本格式和 Docutils 软件套件；
- Fredrik Lundh，Sphinx 从他的 Alternative Python Reference 项目中获得了很多好的想法。

B.1 Python 文档的贡献者

有很多对 Python 语言，Python 标准库和 Python 文档有贡献的人，随 Python 源代码发布的 Misc/ACKS 文件列出了部分贡献者。

有了 Python 社区的输入和贡献，Python 才有了如此出色的文档 - 谢谢你们！

APPENDIX C

历史和许可证

C.1 该软件的历史

Python 由荷兰数学和计算机科学研究学会（CWI，见 https://www.cwi.nl/ ）的 Guido van Rossum 于 1990 年代初设计，作为一门叫做 ABC 的语言的替代品。尽管 Python 包含了许多来自其他人的贡献，Guido 仍是其主要作者。

1995 年，Guido 在弗吉尼亚州的国家创新研究公司（CNRI，见 https://www.cnri.reston.va.us/ ）继续他在 Python 上的工作，并在那里发布了该软件的多个版本。

2000 年五月，Guido 和 Python 核心开发团队转到 BeOpen.com 并组建了 BeOpen PythonLabs 团队。同年十月，PythonLabs 团队转到 Digital Creations (现为 Zope Corporation；见 https://www.zope.org/)。2001 年，Python 软件基金会 (PSF，见 https://www.python.org/psf/) 成立，这是一个专为拥有 Python 相关知识产权而创建的非营利组织。Zope Corporation 现在是 PSF 的赞助成员。

所有的 Python 版本都是开源的（有关开源的定义参阅 https://opensource.org/ ）。历史上，绝大多数 Python 版本是 GPL 兼容的；下表总结了各个版本情况。

发布版本	源自	年份	所有者	GPL 兼容？
0.9.0 至 1.2	n/a	1991-1995	CWI	是
1.3 至 1.5.2	1.2	1995-1999	CNRI	是
1.6	1.5.2	2000	CNRI	否
2.0	1.6	2000	BeOpen.com	否
1.6.1	1.6	2001	CNRI	否
2.1	2.0+1.6.1	2001	PSF	否
2.0.1	2.0+1.6.1	2001	PSF	是
2.1.1	2.1+2.0.1	2001	PSF	是
2.1.2	2.1.1	2002	PSF	是
2.1.3	2.1.2	2002	PSF	是
2.2 及更高	2.1.1	2001 至今	PSF	是

注解： GPL 兼容并不意味着 Python 在 GPL 下发布。与 GPL 不同，所有 Python 许可证都允许您分发修改后

的版本，而无需开源所做的更改。GPL 兼容的许可证使得 Python 可以与其它在 GPL 下发布的软件结合使用；但其它的许可证则不行。

感谢众多在 Guido 指导下工作的外部志愿者，使得这些发布成为可能。

C.2 获取或以其他方式使用 Python 的条款和条件

C.2.1 用于 PYTHON 2.7.18 的 PSF 许可协议

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),␣
↪and
   the Individual or Organization ("Licensee") accessing and otherwise using␣
↪Python
   2.7.18 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to␣
↪reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 2.7.18 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice␣
↪of
   copyright, i.e., "Copyright © 2001-2020 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 2.7.18 alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 2.7.18 or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee␣
↪hereby
   agrees to include in any such work a brief summary of the changes made to␣
↪Python
   2.7.18.

4. PSF is making Python 2.7.18 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION␣
↪OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT␣
↪THE
   USE OF PYTHON 2.7.18 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.7.18
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.7.18, OR ANY␣
↪DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
```

```
6. This License Agreement will automatically terminate upon a material breach␣
↪of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any␣
↪relationship
   of agency, partnership, or joint venture between PSF and Licensee.  This␣
↪License
   Agreement does not grant permission to use PSF trademarks or trade name in␣
↪a
   trademark sense to endorse or promote products or services of Licensee, or␣
↪any
   third party.

8. By copying, installing or otherwise using Python 2.7.18, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 用于 PYTHON 2.0 的 BEOPEN.COM 许可协议

BEOPEN PYTHON 开源许可协议第 1 版

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
```

(下页继续)

(续上页)

```
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 用于 PYTHON 1.6.1 的 CNRI 许可协议

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   Internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
```

(下页继续)

(续上页)

```
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 用于 PYTHON 0.9.0 至 1.2 的 CWI 许可协议

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.3 被收录软件的许可证与鸣谢

本节是 Python 发行版中收录的第三方软件的许可和致谢清单，该清单是不完整且不断增长的。

C.3.1 Mersenne Twister

`_random` 模块包含基于 http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html 下载的代码。以下是原始代码的完整注释（声明）：

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.
```

(下页继续)

(续上页)

C.3.2 套接字

socket 模块使用 getaddrinfo() 和 getnameinfo() 函数，这些函数源代码在 WIDE 项目（http://www.wide.ad.jp/）的单独源文件中。

(下页继续)

(续上页)

```
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Floating point exception control

The source for the `fpectl` module includes the following notice:

```
  ---------------------------------------------------------------------
 /                       Copyright (c) 1996.                           \
|          The Regents of the University of California.                 |
|                        All rights reserved.                           |
|                                                                       |
|   Permission to use, copy, modify, and distribute this software for   |
|   any purpose without fee is hereby granted, provided that this en-   |
|   tire notice is included in all copies of any software which is or   |
|   includes  a  copy  or  modification  of  this software and in all   |
|   copies of the supporting documentation for such software.           |
|                                                                       |
|   This  work was produced at the University of California, Lawrence   |
|   Livermore National Laboratory under  contract  no.  W-7405-ENG-48   |
|   between  the  U.S.  Department  of  Energy and The Regents of the   |
|   University of California for the operation of UC LLNL.              |
|                                                                       |
|                              DISCLAIMER                               |
|                                                                       |
|   This  software was prepared as an account of work sponsored by an   |
|   agency of the United States Government. Neither the United States   |
|   Government  nor the University of California nor any of their em-   |
|   ployees, makes any warranty, express or implied, or  assumes  any   |
|   liability  or  responsibility  for the accuracy, completeness, or   |
|   usefulness of any information,  apparatus,  product,  or  process   |
|   disclosed,   or  represents  that  its  use  would  not  infringe   |
|   privately-owned rights. Reference herein to any specific  commer-   |
|   cial  products,  process,  or  service  by trade name, trademark,   |
|   manufacturer, or otherwise, does not  necessarily  constitute  or   |
|   imply  its endorsement, recommendation, or favoring by the United   |
|   States Government or the University of California. The views  and   |
|   opinions  of authors expressed herein do not necessarily state or   |
|   reflect those of the United States Government or  the  University   |
|   of  California,  and shall not be used for advertising or product   |
 \  endorsement purposes.                                              /
  ---------------------------------------------------------------------
```

C.3.4 MD5 message digest algorithm

The source code for the `md5` module contains the following notice:

```
Copyright (C) 1999, 2002 Aladdin Enterprises.  All rights reserved.

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

L. Peter Deutsch
ghost@aladdin.com

Independent implementation of MD5 (RFC 1321).

This code implements the MD5 Algorithm defined in RFC 1321, whose
text is available at
      http://www.ietf.org/rfc/rfc1321.txt
The code is derived from the text of the RFC, including the test suite
(section A.5) but excluding the rest of Appendix A.  It does not include
any code or documentation that is identified in the RFC as being
copyrighted.

The original and principal author of md5.h is L. Peter Deutsch
<ghost@aladdin.com>.  Other authors are noted in the change history
that follows (in reverse chronological order):

2002-04-13 lpd Removed support for non-ANSI compilers; removed
      references to Ghostscript; clarified derivation from RFC 1321;
      now handles byte order either statically or dynamically.
1999-11-04 lpd Edited comments slightly for automatic TOC extraction.
1999-10-18 lpd Fixed typo in header comment (ansi2knr rather than md5);
      added conditionalization for C++ compilation from Martin
      Purschke <purschke@bnl.gov>.
1999-05-03 lpd Original version.
```

C.3.5 异步套接字服务

asynchat and asyncore 模块包含以下声明:

C.3.6 Cookie 管理

The Cookie module contains the following notice:

C.3.7 执行追踪

`trace` 模块包含以下声明:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.8 UUencode 与 UUdecode 函数

`uu` 模块包含以下声明:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
```

(下页继续)

(续上页)

```
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.9 XML 远程过程调用

The `xmlrpclib` module contains the following notice:

C.3.10 test_epoll

The `test_epoll` contains the following notice:

(下页继续)

(续上页)

```
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.11 Select kqueue

The `select` and contains the following notice for the kqueue interface:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.12 strtod and dtoa

`Python/dtoa.c` 文件提供了 C 语言的 dtoa 和 strtod 函数，用于将 C 语言的双精度型和字符串进行转换，该文件由 David M. Gay 的同名文件派生而来，当前可从 http://www.netlib.org/fp/ 下载。2009 年 3 月 16 日检索到的原始文件包含以下版权和许可声明:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
```

(下页继续)

(续上页)

```
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.13 OpenSSL

如果操作系统可用，则 hashlib, posix, ssl, crypt 模块使用 OpenSSL 库来提高性能。此外，适用于 Python 的 Windows 和 Mac OS X 安装程序可能包括 OpenSSL 库的拷贝，所以在此处也列出了 OpenSSL 许可证的拷贝:

```
LICENSE ISSUES
==============

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
---------------

  /* ====================================================================
   * Copyright (c) 1998-2008 The OpenSSL Project.  All rights reserved.
   *
   * Redistribution and use in source and binary forms, with or without
   * modification, are permitted provided that the following conditions
   * are met:
   *
   * 1. Redistributions of source code must retain the above copyright
   *    notice, this list of conditions and the following disclaimer.
   *
   * 2. Redistributions in binary form must reproduce the above copyright
   *    notice, this list of conditions and the following disclaimer in
   *    the documentation and/or other materials provided with the
   *    distribution.
   *
   * 3. All advertising materials mentioning features or use of this
   *    software must display the following acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
   *
   * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
   *    endorse or promote products derived from this software without
   *    prior written permission. For written permission, please contact
   *    openssl-core@openssl.org.
   *
   * 5. Products derived from this software may not be called "OpenSSL"
   *    nor may "OpenSSL" appear in their names without prior written
   *    permission of the OpenSSL Project.
   *
   * 6. Redistributions of any form whatsoever must retain the following
```

(下页继续)

(续上页)

(下页继续)

(续上页)

C.3.14 expat

除非使用 `--with-system-expat` 配置了构建，否则 `pyexpat` 扩展都是用包含 expat 源的拷贝构建的:

C.3.15 libffi

除非使用 `--with-system-libffi` 配置了构建，否则 `_ctypes` 扩展都是包含 libffi 源的拷贝构建的:

C.3.16 zlib

如果系统上找到的 zlib 版本太旧而无法用于构建，则使用包含 zlib 源代码的拷贝来构建 `zlib` 扩展:

APPENDIX D

Copyright

有关完整的许可证和许可信息，参见历史和许可证。

索引