Python手册Python Tutorial

Release 2.3.3

Guido van Rossum
Fred L. Drake, Jr., editor
翻译团队: **见文末名单**

译文最后修改: 2004**年**7**月**12**日**

PythonLabs
Email: docs@python.org

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

Python **是一种容易学习的强大语言。它包括了高效的高级数据结构，提供了一个简单但很有效的方式进行面向对象编程。**Python **优雅的语法，动态类型，以及它天然的解释能力，使其成为了大多数平台上应用于各领域理想的脚本语言以及开发环境。**

The Python interpreter and the extensive standard library are freely available in source or binary form for all major platforms from the Python Web site, http://www.python.org/, and can be freely distributed. The same site also contains distributions of and pointers to many free third party Python modules, programs and tools, and additional documentation.

Python **解释器及其扩展标准库的源码和编译版本可以从**Python **的**Web **站点**, http://www.python.org/, **及其所有镜像站上免费获得，并且可以自由发布。该站点上也提供了**Python **的一些第三方模块，程序，工具，以及附加的文档。**

The Python interpreter is easily extended with new functions and data types implemented in C or C++ (or other languages callable from C). Python is also suitable as an extension language for customizable applications.

Python **的解释器很容易通过**C **或**C++ **（或者其它可以由**C**来调用的语言）来实现功能和数据结构的扩展。因此**Python **也很适于做为定制应用的一种扩展语言。**

This tutorial introduces the reader informally to the basic concepts and features of the Python language and system. It helps to have a Python interpreter handy for hands-on experience, but all examples are self-contained, so the tutorial can be read off-line as well.

这个手册介绍了一些Python **语言及其系统的基本知识与概念。这有助于读者对**Python **有一个基本的认识，当然所有的例子都包括在里面文中了，所以这本手册很适合离线阅读。**

For a description of standard objects and modules, see the *Python Library Reference* document. The *Python Reference Manual* gives a more formal definition of the language. To write extensions in C or C++, read *Extending and Embedding the Python Interpreter* and *Python/C API Reference*. There are also several books covering Python in depth.

需要有关标准对象和模块的详细介绍的话，请查询*Python* ***库参考手册*** **文档。***Python* ***参考手册*** **提供了更多的关于语言方面的正式说明。需要编写**C**或**C++**扩展，请阅读***Python* ***解释器的扩展和集成*** **以及***Python/C API* ***参考手册*****。这几本书涵盖了各个深度上的**Python**知识。**

This tutorial does not attempt to be comprehensive and cover every single feature, or even every commonly used feature. Instead, it introduces many of Python's most noteworthy features, and will give you a good idea of the language's flavor and style. After reading it, you will be able to read and write Python modules and programs, and you will be ready to learn more about the various Python library modules described in the *Python Library Reference*.

本手册不会涵盖Python **的所有功能，也不会去解释所用到的所有相关的知识。相反，它介绍了许多**Python **中最引人注目的功能，这会对读者掌握这门语言的风格大有帮助。读过它后，你应该可以阅读和编写**Python **模块和程序，接下来可以从***Python* ***库参考手册*** **中进一步学习**Python**复杂多变的库和模块。**

目录

第一章

开胃菜Whetting Your Appetite

If you ever wrote a large shell script, you probably know this feeling: you'd love to add yet another feature, but it's already so slow, and so big, and so complicated; or the feature involves a system call or other function that is only accessible from C . . . Usually the problem at hand isn't serious enough to warrant rewriting the script in C; perhaps the problem requires variable-length strings or other data types (like sorted lists of file names) that are easy in the shell but lots of work to implement in C, or perhaps you're not sufficiently familiar with C.

如果你写过大规模的Shell 脚本，应该会有过这样的体会：你还非常想再加一些别的功能进去，但它已经太大、太慢、太复杂了；或者这个功能需要调用一个系统函数，或者它只适合通过C 来调用. . . 通常这些问题还不足以严肃到需要用C 重写这个脚本；可能这个功能需要一些类似变长字符串或其它一些在Shell 脚本中很容易找到的数据类型（比如文件名的有序列表），但它们用C 来实现就要做大量的工作，或者，你对C 还不是很熟悉。

Another situation: perhaps you have to work with several C libraries, and the usual C write/compile/test/re-compile cycle is too slow. You need to develop software more quickly. Possibly perhaps you've written a program that could use an extension language, and you don't want to design a language, write and debug an interpreter for it, then tie it into your application.

另一种情况：可能你需要使用几个C 库来工作，通常C 的编写/编译/测试/重编译周期太慢。你需要尽快的开发软件。也许你需要写一个使用扩展语言的程序，但不想设计一个语言，并为此编写调试一个解释器，然后再把它集成进你的程序。

In such cases, Python may be just the language for you. Python is simple to use, but it is a real programming language, offering much more structure and support for large programs than the shell has. On the other hand, it also offers much more error checking than C, and, being a *very-high-level language*, it has high-level data types built in, such as flexible arrays and dictionaries that would cost you days to implement efficiently in C. Because of its more general data types Python is applicable to a much larger problem domain than *Awk* or even *Perl*, yet many things are at least as easy in Python as in those languages.

遇到以上情况，Python 可能就是你要找的语言。Python 很容易上手，但它是一门真正的编程语言，相对于Shell，它提供的针对大型程序的支持和结构要多的多。另一方面，它提供了比C 更多的错误检查，并且，做为一门*高级语言*，它拥有内置的高级数据类型，例如可变数组和字典，如果通过C 来实现的话，这些工作可能让你大干上几天的时间。因为拥有更多的通用数据类型，Python 适合比*Awk* 甚至*Perl* 更广泛的问题领域，在其它的很多领域，Python 至少比别的语言要易用得多。

Python allows you to split up your program in modules that can be reused in other Python programs. It comes with a large collection of standard modules that you can use as the basis of your programs — or as examples to start learning to program in Python. There are also built-in modules that provide things like file I/O, system calls, sockets, and even interfaces to graphical user interface toolkits like Tk.

Python 可以让你把自己的程序分隔成不同的模块，以便在其它的Python 程序中重用。这样你就可以让自己的程序基于一个很大的标准模块集或者用它们做为示例来学习Python 编程。Python 中集成了一些类似文件I/O，系统调用，sockets，甚至像Tk 这样的用户图形接口。

Python is an interpreted language, which can save you considerable time during program development because no compilation and linking is necessary. The interpreter can be used interactively, which makes it easy to experiment with features of the language, to write throw-away programs, or to test functions during bottom-up program development. It is also a handy desk calculator.

Python**是一门解释型语言，因为不需要编译和链接的时间，它可以帮你省下一些开发时间。解释器可以交互式使用，这样就可以很方便的测试语言中的各种功能，以便于编写发布用的程序，或者进行自下而上的开发。还可以当它是一个随手可用的计算器。**

Python allows writing very compact and readable programs. Programs written in Python are typically much shorter than equivalent C or C++ programs, for several reasons:

Python **可以写出很紧凑和可读性很强的程序。用**Python **写的程序通常比同样的**C **或**C++ **程序要短得多，这是因为以下几个原因：**

- the high-level data types allow you to express complex operations in a single statement; **高级数据结构使你可以在一个单独的语句中表达出很复杂的操作；**
- statement grouping is done by indentation instead of beginning and ending brackets; **语句的组织依赖于缩进而不是**begin/end **块；**
- no variable or argument declarations are necessary. **不需要变量或参数声明。**

Python is *extensible*: if you know how to program in C it is easy to add a new built-in function or module to the interpreter, either to perform critical operations at maximum speed, or to link Python programs to libraries that may only be available in binary form (such as a vendor-specific graphics library). Once you are really hooked, you can link the Python interpreter into an application written in C and use it as an extension or command language for that application.

Python **是*可扩展的*：如果你会用**C **语言写程序，那就可以很容易的为解释器添加新的集成模块和功能，或者优化瓶颈，使其达到最大速度，或者使**Python **能够链接到所需的二进制架构上（比如某个专用的商业图形库）。等你真正熟悉这一切了，你就可以把**Python **集成进由**C **写成的程序，把**Python **当做这个程序的扩展或命令行语言。**

By the way, the language is named after the BBC show “Monty Python’s Flying Circus” and has nothing to do with nasty reptiles. Making references to Monty Python skits in documentation is not only allowed, it is encouraged!

顺便说一下，这个语言的名字来源于BBC **的“** Monty Python’s Flying Circus **”节目，和凶猛的爬虫没有任何关系。在文档中引用**Monty Python **典故不仅是允许的，而且还受到鼓励！**

Now that you are all excited about Python, you’ll want to examine it in some more detail. Since the best way to learn a language is using it, you are invited here to do so.

现在你已经了解了Python **中所有激动人心的东西，大概你想仔细的试试它了。学习一门语言最好的办法就是使用它，你会很乐于这样做。**

In the next chapter, the mechanics of using the interpreter are explained. This is rather mundane information, but essential for trying out the examples shown later.

下一节中，我们会很机械的说明解释器的用法。这没有什么神秘的，不过有助于我们练习后面展示的例子。

The rest of the tutorial introduces various features of the Python language and system through examples, beginning with simple expressions, statements and data types, through functions and modules, and finally touching upon advanced concepts like exceptions and user-defined classes.

本指南其它部分通过例子介绍了Python **语言和系统的各种功能，开始是简单表达式、语法和数据类型，接下来是函数和模块，最后是诸如异常和自定义类这样的高级内容。**

第二章

使用Python解释器Using the Python Interpreter

2.1 调用解释器Invoking the Interpreter

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

通常Python 的解释器被安装在目标机器的'/usr/local/bin/python' 目录下；把'/usr/local/bin' 目录放进你的UNIX Shell 的搜索路径里，确保它可以通过输入

```
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.)

来启动。因为安装路径是可选的，所以也有可能安装在其它位置，你可以与安装Python 的用户或系统管理员联系。（例如，'/usr/local/python'就是一个很常见的选择）

Typing an end-of-file character (`Control-D` on UNIX, `Control-Z` on Windows) at the primary prompt causes the interpreter to exit with a zero exit status. If that doesn't work, you can exit the interpreter by typing the following commands: '`import sys; sys.exit()`'.

输入一个文件结束符（UNIX上是`Ctrl+D`，Windows上是`Ctrl+Z`）解释器会以0值退出（就是说，没有什么错误，正常退出－－译者）。如果这没有起作用，你可以输入以下命令退出：'`import sys; sys.exit()`'。

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 A 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.

解释器的行编辑功能并不很复杂。装在UNIX上的解释器可能会有GNU readline 库支持，这样就可以额外得到精巧的交互编辑和历史记录功能。可能检查命令行编辑器支持能力最方便的方式是在主提示符下输入Ctrl-P。如果有嘟嘟声（计算机扬声器），说明你可以使用命令行编辑功能，从附录A A 可以查到快捷键的介绍。如果什么也没有发声，或者`^P`显示了出来，说明命令行编辑功能不可用，你只有用退格键删掉输入的命令了。

The interpreter operates somewhat like the UNIX shell: when called with standard input connected to a tty device, it reads and executes commands interactively; when called with a file name argument or with a file as standard input, it reads and executes a *script* from that file.

解释器的操作有些像UNIXShell**：使用终端设备做为标准输入来调用它时，解释器交互的解读和执行命令，通过文件名参数或以文件做为标准输入设备时，它从文件中解读并执行*脚本*。**

A second way of starting the interpreter is '**python -c** *command* `[arg] ...`', which executes the statement(s) in *command*, analogous to the shell's **-c** option. Since Python statements often contain spaces or other characters that are special to the shell, it is best to quote *command* in its entirety with double quotes.

启动解释器的第二个方法是'**python -c** *command* `[arg] ...`'**，这种方法可以在命令行中直接执行语句，等同于**Shell**的-c选项。因为**Python**语句通常会包括空格之类的特殊字符，所以最好把整个*语句*用双引号包起来。**

Note that there is a difference between '`python file`' and '`python <file`'. In the latter case, input requests from the program, such as calls to `input()` and `raw_input()`, are satisfied from *file*. Since this file has already been read until the end by the parser before the program starts executing, the program will encounter end-of-file immediately. In the former case (which is usually what you want) they are satisfied from whatever file or device is connected to standard input of the Python interpreter.

注意'`python file`'**和**'`python <file`'**是有区别的。对于后一种情况，程序中类似于调用**`input()raw_input()`**这样的输入请求，来自于确定的*文件*。因为在解析器开始执行之前，文件已经完全读入，所以程序指向文件尾。在前一种情况（这通常是你需要的）它们从来自于任何联接到**Python**解释器的标准输入，无论它们是文件还是其它设备。**

When a script file is used, it is sometimes useful to be able to run the script and enter interactive mode afterwards. This can be done by passing **-i** before the script. (This does not work if the script is read from standard input, for the same reason as explained in the previous paragraph.)

使用脚本文件时，经常会运行脚本然后进入交互模式。这也可以通过在脚本之前加上-i参数来实现。（如果脚本来自标准输入，就不能这样运行，与前一段提到的原因一样。）

2.1.1 参数传递Argument Passing

When known to the interpreter, the script name and additional arguments thereafter are passed to the script in the variable `sys.argv`, which is a list of strings. Its length is at least one; when no script and no arguments are given, `sys.argv[0]` is an empty string. When the script name is given as `'-'` (meaning standard input), `sys.argv[0]` is set to `'-'`. When **-c** *command* is used, `sys.argv[0]` is set to `'-c'`. Options found after **-c** *command* are not consumed by the Python interpreter's option processing but left in `sys.argv` for the command to handle.

调用解释器时，脚本名和附加参数之传入一个名为`sys.argv`**的字符串列表。没有脚本和参数时，它至少也有一个元素：**`sys.argv[0]`**此时为空字符串。脚本名指定为**`'-'`**（表示标准输入）时，**`sys.argv[0]`**被设置为**`'-'`**，使用-c *指令*时，**`sys.argv[0]`**被设定为**`'-c'`**。-c *命令*之后的参数不会被**Python**解释器的选项处理机制所截获，而是留在**`sys.argv`**中，供脚本命令操作。**

2.1.2 交互模式Interactive Mode

When commands are read from a tty, the interpreter is said to be in *interactive mode*. In this mode it prompts for the next command with the *primary prompt*, usually three greater-than signs ('`>>> `'); for continuation lines it prompts with the *secondary prompt*, by default three dots ('`... `'). The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt:

从tty**读取命令时，我们称解释器工作于*交互模式*。这种模式下它根据*主提示符*来执行，主提示符通常标识为三个大于号（**'`>>> `'**）；继续的部分被称为*从属提示符*，由三个点标识（**'`... `'**）。在第一行之前，解释器打印欢迎信息、版本号和授权提示：**

```
python
Python 1.5.2b2 (#1, Feb 28 1999, 00:02:06)  [GCC 2.8.1] on sunos5
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>>
```

Continuation lines are needed when entering a multi-line construct. As an example, take a look at this `if` statement:

输入多行结构时需要从属提示符了，例如，下面这个`if` 语句：

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

2.2 解释器及其环境The Interpreter and Its Environment

2.2.1 错误处理Error Handling

When an error occurs, the interpreter prints an error message and a stack trace. In interactive mode, it then returns to the primary prompt; when input came from a file, it exits with a nonzero exit status after printing the stack trace. (Exceptions handled by an `except` clause in a `try` statement are not errors in this context.) Some errors are unconditionally fatal and cause an exit with a nonzero exit; this applies to internal inconsistencies and some cases of running out of memory. All error messages are written to the standard error stream; normal output from the executed commands is written to standard output.

有错误发生时，解释器打印一个错误信息和栈跟踪（监视）器？。交互模式下，它返回主提示符，如果从文件输入执行，它在打印栈跟踪器后以非零状态退出。（异常可以由`try` 语句中的`except` 子句来控制，这样就不会出现上文中的错误信息）有一些非常致命的错误会导致非零状态下退出，这由通常由内部矛盾和内存溢出造成。所有的错误信息都写入标准错误流；命令中执行的普通输出写入标准输出。

Typing the interrupt character (usually Control-C or DEL) to the primary or secondary prompt cancels the input and returns to the primary prompt.[1] Typing an interrupt while a command is executing raises the `KeyboardInterrupt` exception, which may be handled by a `try` statement.

在主提示符或附属提示符输入中断符（通常是Control-C or DEL）就会取消当前输入，回到主命令行。[2].执行命令时输入一个中断符会抛出一个`KeyboardInterrupt` 异常，它可以被`try` 句截获。

2.2.2 执行Python脚本Executable Python Scripts

On BSD'ish UNIX systems, Python scripts can be made directly executable, like shell scripts, by putting the line

BSD类的UNIX系统中，Python 脚本可以像Shell 脚本那样直接执行，只要在脚本文件开头写一行命令，指定文件和模式：

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

(assuming that the interpreter is on the user's PATH) at the beginning of the script and giving the file an executable mode. The '`#!`' must be the first two characters of the file. On some platforms, this first line must end with a UNIX-style line ending ('`\n`'), not a Mac OS ('`\r`') or Windows ('`\r\n`') line ending. Note that the hash, or pound, character, '#', is used to start a comment in Python.

(将用户路径通知解释器) '`#!`' 必须是文件的前两个字符，在某些平台上，第一行必须以UNIX风格的行结束符（'`\n`'）结束，不能用Mac（'`\r`'）或Windows（'`\r\n`'）的结束符。注意，'#'是Python中是行注释的起始符。

The script can be given a executable mode, or permission, using the **chmod** command:

脚本可以通过chmod 命令指定执行模式和许可权。

[1]A problem with the GNU Readline package may prevent this.
[2]GNU readline **包的一个错误可能会造成无法正常工作。**

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

2.2.3 源程序编码Source Code Encoding

It is possible to use encodings different than ASCII in Python source files. The best way to do it is to put one more special comment line right after the #! line to define the source file encoding:

Python 的源文件可以通过编码使用ASCII 以外的字符集。最好的做法是在#! 行后面用一个特殊的注释行来定义字符集。

```
# -*- coding: iso-8859-1 -*-
```

With that declaration, all characters in the source file will be treated as `iso-8859-1`, and it will be possible to directly write Unicode string literals in the selected encoding. The list of possible encodings can be found in the *Python Library Reference*, in the section on `codecs`.

根据这个声明，Python 会将文件中的字符尽可能的从指定的编码转为Unicode，在本例中，这个字符集是iso-8859-1 。在*Python 库参考手册* 中`codecs`部份可以找到可用的编码列表（根据个人经验，推荐使用cp-936或utf-8处理中文－－译者注）。

If your editor supports saving files as `UTF-8` with a UTF-8 *byte order mark* (aka BOM), you can use that instead of an encoding declaration. IDLE supports this capability if `Options/General/Default Source Encoding/UTF-8` is set. Notice that this signature is not understood in older Python releases (2.2 and earlier), and also not understood by the operating system for #! files.

如果你的文件编辑器支持`UTF-8` 格式，并且可以保存`UTF-8` 标记（*aka BOM - Byte Order Mark*），你可以用这个来代替编码声明。IDLE可以通过设定`Options/General/Default Source Encoding/UTF-8` 来支持它。需要注意的是旧版Python不支持这个标记（Python 2.2或更早的版本），也同样不能使操作系统支持#!文件。

By using UTF-8 (either through the signature or an encoding declaration), characters of most languages in the world can be used simultaneously in string literals and comments. Using non-ASCIIcharacters in identifiers is not supported. To display all these characters properly, your editor must recognize that the file is UTF-8, and it must use a font that supports all the characters in the file.

使用UTF-8 内码（无论是用标记还是编码声明），我们可以在字符串和注释中使用世界上的大部分语言。标识符中不能使用非ASCII 字符集。为了正确显示所有的字符，你一定要在编辑器中将文件保存为UTF-8 格式，而且要使用支持文件中所有字符的字体。

2.2.4 交互式环境的启动文件The Interactive Startup File

When you use Python interactively, it is frequently handy to have some standard commands executed every time the interpreter is started. You can do this by setting an environment variable named PYTHONSTARTUP to the name of a file containing your start-up commands. This is similar to the '.profile' feature of the UNIX shells.

使用Python 解释器的时候，我们可能需要在每次解释器启动时执行一些命令。你可以在一个文件中包含你想要执行的命令，设定一个名为PYTHONSTARTUP 的环境变量来指定这个文件。这类似于Unix shell的'.profile' 文件。

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.

这个文件在交互会话期是只读的，当Python 从脚本中解读文件或以终端'/dev/tty' 做为外部命令源时则不会如此（尽管它们的行为很像是处在交互会话期。）它与解释器执行的命令处在同一个命名空间，

所以由它定义或引用的一切可以在解释器中不受限制的使用。你也可以在这个文件中改变`sys.ps1` **和**`sys.ps2` **指令。**

If you want to read an additional start-up file from the current directory, you can program this in the global start-up file using code like '`if os.path.isfile('.pythonrc.py'):  execfile('.pythonrc.py')`'. If you want to use the startup file in a script, you must do this explicitly in the script:

如果你想要在当前目录中执行附加的启动文件，你可以在全局启动文件中加入类似以下的代码：'`if os.path.isfile('.pythonrc.py'):  execfile('.pythonrc.py')`'**。如果你想要在某个脚本中使用启动文件，必须要在脚本中写入这样的语句：**

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    execfile(filename)
```

第三章

Python的非正式介绍An Informal Introduction to Python

In the following examples, input and output are distinguished by the presence or absence of prompts (‘>>> ’ and ‘... ’): to repeat the example, you must type everything after the prompt, when the prompt appears; lines that do not begin with a prompt are output from the interpreter. Note that a secondary prompt on a line by itself in an example means you must type a blank line; this is used to end a multi-line command.

在后面的例子中，区分输入和输出的方法是看是否有提示符（‘»> ’和‘... ’）：想要重现这些例子的话，你就要在提示符显示后输入所有的一切；没有以提示符开始的行，是解释器输出的信息。需要注意的是示例中的从属提示符用于多行命令的结束，它表示你需要输入一个空行。

Many of the examples in this manual, even those entered at the interactive prompt, include comments. Comments in Python start with the hash character, ‘#’, and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character.

本手册中的很多示例都包括注释，甚至有一些在交互提示符中折行。Python**中的注释以符号**‘#’ **起始，一直到当前行的结尾。注释可能出现在一行的开始，也可能跟在空格或程序代码之后，但不会出现在字符串中，字符串中的**# **号只代表**# **号。**

Some examples: **示例：**

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

3.1 初步认识Python Using Python as a Calculator

Let’s try some simple Python commands. Start the interpreter and wait for the primary prompt, ‘>>> ’. (It shouldn’t take long.)

让我们试验一些简单的Python **命令。启动解释器然后等待主提示符**‘>>> ’**出现（这用不了太久）。**

3.1.1 数值Numbers

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators `+`, `-`, `*` and `/` work just like in most other languages (for example, Pascal or C); parentheses can be used for grouping. For example:

解释器的行为就像是一个计算器。你可以向它输入一个表达式，它会返回结果。表达式的语法简明易懂：`+`，`-`，`*`，`/`和大多数语言中的用法一样（比如C**或**Pascal**），括号用于分组。例如：**

```
>>> 2+2
4
>>> # This is a comment
... 2+2
4
>>> 2+2  # and a comment on the same line as code
4
>>> (50-5*6)/4
5
>>> # Integer division returns the floor:
... 7/3
2
>>> 7/-3
-3
```

Like in C, the equal sign ('`=`') is used to assign a value to a variable. The value of an assignment is not written:

像C**一样，等号（'`=`'）用于给变量赋值。被分配的值是只读的。**

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

A value can be assigned to several variables simultaneously:

同一个值可以同时赋给几个变量：

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

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

Python**完全支持浮点数，不同类型的操作数混在一起时，操作符会把整型转化为浮点数。**

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

Complex numbers are also supported; imaginary numbers are written with a suffix of '`j`' or '`J`'. Complex numbers with a nonzero real component are written as '`(`*real*+*imag*`j)`', or can be created with the '`complex(`*real*`,` *imag*`)`' function.

复数也同样得到了支持，虚部由一个后缀'`j`'或者'`J`'来表示。带有非零实部的复数记为'*real*+*imag*`j)`'，或者也可以通过'`complex(`*real*`,` *img*`)`'函数创建。

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

Complex numbers are always represented as two floating point numbers, the real and imaginary part. To extract these parts from a complex number *z*, use *z*`.real` and *z*`.imag`.

复数总是由实部和虚部两部分浮点数来表示。可以从z`.real` **和**z`.imag` **得到复数**z**的实部和虚部。**

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

The conversion functions to floating point and integer (`float()`, `int()` and `long()`) don't work for complex numbers — there is no one correct way to convert a complex number to a real number. Use `abs(`*z*`)` to get its magnitude (as a float) or `z.real` to get its real part.

用于向浮点数和整型转化的函数（float(), int() **和**long()**）不能对复数起作用－－没有什么方法可以将复数转化为实数。可以使用**abs(z)**取得它的模，也可以通过**z.real**得到它的实部。**

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

In interactive mode, the last printed expression is assigned to the variable _. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

交互模式下，最近一次表达式输出保存在_ 变量中。这意味着把 Python **当做桌面计算器使用时，它可以更方便的进行连续计算，例如：**

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

This variable should be treated as read-only by the user. Don't explicitly assign a value to it — you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

这个变量对于用户来说是只读的。不要试图去给它赋值－－限于 Python **的语法规则，你只会创建一个同名的局部变量覆盖它。**

3.1.2 **字符串**Strings

Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes or double quotes:

除了数值，Python **还可以通过几种不同的方法操作字符串。字符串用单引号或双引号标识：**

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

String literals can span multiple lines in several ways. Continuation lines can be used, with a backslash as the last character on the line indicating that the next line is a logical continuation of the line:

字符串可以通过几种方式分行。可以在行加反斜杠做为继续符，这表示下一行是当前行的逻辑沿续。

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

print hello
```

Note that newlines would still need to be embedded in the string using `\n`; the newline following the trailing backslash is discarded. This example would print the following:

注意换行用\n **来表示；反斜杠后面的新行标识（**newline**，缩写"** n **"）会转换为换行符，示例会按如下格式打印：**

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

If we make the string literal a "raw" string, however, the `\n` sequences are not converted to newlines, but the backslash at the end of the line, and the newline character in the source, are both included in the string as data. Thus, the example:

然而，如果我们创建一个"行"("raw")**字符串，**\n**序列就不会转为换行，示例源码最后的反斜杠和换行符**n**都会做为字符串中的数据处理。如下所示：**

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

print hello
```

would print:

会打印为：

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

Or, strings can be surrounded in a pair of matching triple-quotes: `"""` or `'''`. End of lines do not need to be escaped when using triple-quotes, but they will be included in the string.

或者，字符串可以用一对三重引号” ” ”或”’来标识。三重引号中的字符串在行尾不需要换行标记，所有的格式都会包括在字符串中。

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

produces the following output: **产生以下结果**

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

The interpreter prints the result of string operations in the same way as they are typed for input: inside quotes, and with quotes and other funny characters escaped by backslashes, to show the precise value. The string is enclosed in double quotes if the string contains a single quote and no double quotes, else it's enclosed in single quotes. (The `print` statement, described later, can be used to write strings without quotes or escapes.)

解释器打印出来的字符串与它们输入的形式完全相同：内部的引号，用反斜杠标识的引号和各种怪字符，都精确的显示出来。如果字符串中包含单引号，不包含双引号，可以用双引号引用它，反之可以用单引号。（后面介绍的print **语句，可以在不使用引号和反斜杠的情况下输出字符串）。**

Strings can be concatenated (glued together) with the `+` operator, and repeated with `*`:

字符串可以用+ **号联接（或者说粘合），也可以用*** **号循环。**

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

Two string literals next to each other are automatically concatenated; the first line above could also have been written '`word = 'Help' 'A'`'; this only works with two literals, not with arbitrary string expressions:

两个字符串值之间的联接是自动的，上例第一行可以写成“ word = 'Help' 'A' **”。这种方式只对字符串值有效，任何字符串表达式都不适用这种方法。**

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

Strings can be subscripted (indexed); like in C, the first character of a string has subscript (index) 0. There is no separate character type; a character is simply a string of size one. Like in Icon, substrings can be specified with the *slice notation*: two indices separated by a colon.

字符串可以用下标（索引）查询；就像C **一样，字符串的第一个字符下标是**0 **。这里没有独立的字符类型，字符仅仅是大小为一的字符串。就像在**Icon **中那样，字符串的子串可以通过切片标志来表示：两个由冒号隔开的索引。**

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

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

切片索引可以使用默认值；省略前一个索引表示0 **，省略后一个索引表示被切片的字符串的长度。**

```
>>> word[:2]    # The first two characters
’He’
>>> word[2:]    # All but the first two characters
’lpA’
```

Unlike a C string, Python strings cannot be changed. Assigning to an indexed position in the string results in an error:

和C **字符串不同，**Python **字符串不能改写。按字符串索引赋值会产生错误。**

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

However, creating a new string with the combined content is easy and efficient:

然而，可以通过简单有效的组合方式生成新的字符串：

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

Here’s a useful invariant of slice operations:

切片操作有一个很有用的不变性：

`s[:i] + s[i:]` equals `s`.

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

Degenerate slice indices are handled gracefully: an index that is too large is replaced by the string size, an upper bound smaller than the lower bound returns an empty string.

退化的切片索引被处理的很优美：过大的索引代替为字符串大小，下界比上界大的返回空字符串。

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

Indices may be negative numbers, to start counting from the right. For example:

索引可以是负数，计数从右边开始，例如：

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

But note that -0 is really the same as 0, so it does not count from the right!

不过-0还是0，它没有从右边计数！

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

Out-of-range negative slice indices are truncated, but don’t try this for single-element (non-slice) indices:

越界的负切片索引会被截断，不过不要尝试在前元素索引（非切片的）中这样做：

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

The best way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of *n* characters has index *n*, for example:

理解切片的最好方式是把索引视为两个字符之间的点，第一个字符的左边是0**，字符串中第**n**个字符的右边是索引**n**，例如：**

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

The first row of numbers gives the position of the indices 0...5 in the string; the second row gives the corresponding negative indices. The slice from *i* to *j* consists of all characters between the edges labeled *i* and *j*, respectively.

第一行是字符串中给定的0**到**5**各个索引的位置，第二行是对应的负索引。从**i**到**j**的切片由这两个标志之间的字符组成。**

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of `word[1:3]` is 2.

对于非负索引，切片长度就是两索引的差。例如，word[1:3]**的长度是**2**。**

The built-in function `len()` returns the length of a string:

内置函数len() **返回字符串长度：**

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

See Also:

Sequence Types
(../lib/typesseq.html)
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
(../lib/string-methods.html)
Both strings and Unicode strings support a large number of methods for basic transformations and searching.

String Formatting Operations
(../lib/typesseq-strings.html)
The 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 字符串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.

从Python2.0**开始，程序员们可以使用一种新的数据类型来存储文本数据：**Unicode **对象。它可以用于存储多种**Unicode**数据（请参阅**http://www.unicode.org/ **），并且，通过必要时的自动转换，它可以与现有的字符串对象良好的结合。**

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 and 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.

Unicode **针对现代和旧式的文本中所有的字符提供了一个序列。以前，字符只能使用**256**个序号，文本通常通过绑定代码页来与字符映射。这很容易导致混乱，特别是软件的国际化（**internationalization **－－通常写做“**i18n**”－－“**i**”**+18 characters +“n”**）。**Unicode **通过为所有字符定义一个统一的代码页解决了这个问题。**

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

Python **中定义一个**Unicode **字符串和定义一个普通字符串一样简单：**

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

The small '`u`' in front of the quote indicates that an 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:

引号前小写的“u**”表示这里创建的是一个**Unicode **字符串。如果你想加入一个特殊字符，可以使用**Python **的**Unicode-Escape **编码。如下例所示：**

```
>>> 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.

被替换的`\u0020` **标识表示在给定位置插入编码值为**0x0020 **的**Unicode **字符（空格符）。**

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.

其它字符也会被直接解释成对应的Unicode **码。如果你有一个在西方国家常用的**Latin-1 **编码字符串，你可以发现**Unicode **字符集的前**256**个字符与**Latin-1 **的对应字符编码完全相同。**

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 `\u`XXXX conversion if there is an uneven number of backslashes in front of the small 'u'.

另外，有一种与普通字符串相同的行模式。想要使用 Python **的**Raw-Unicode-Escape **编码，你需要在字符串的引号前加上**ur **前缀。如果在小写“**u**”前可能有不止一个反斜杠，它只会把那些单独的**
uXXXX **转化为**Unicode**字符。**

```
>>> 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.

行模式在你需要输入很多个反斜杠时很有用，可能会用于正则表达式。

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

作为这些编码标准的一部分，Python **提供了一个完备的方法集用于从已知的编码集创建**Unicode **字符串。**

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.

内置函数unicode() **提供了访问（编码和解码）所有已注册的**Unicode **编码的方法。它能转换众所周知的**Latin-1, ASCII, UTF-8, **和**UTF-16**。后面的两个可变长编码字符集用一个或多个**byte **存储**Unicode **字符。默认的字符集是**ASCII **，它只处理**0**到**127**的编码，拒绝其它的字符并返回一个错误。当一个**Unicode **字符串被打印、写入文件或通过**str() **转化时，它们被替换为默认的编码。**

```
>>> u"abc"
u'abc'
>>> str(u"abc")
'abc'
>>> u"漩"
u'\xe4\xf6\xfc'
>>> str(u"漩")
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.

要把一个Unicode **字符串用指定的字符集转化成**8**位字符串，可以使用**Unicode **对象提供的**encode() **方法，它有一个参数用以指定编码名称。编码名称小写。**

```
>>> u"漩".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.

如果你有一个特定编码的字符串，想要把它转为 Unicode **字符集，，可以使用**encode() **函数，它以编码名做为第二个参数。**

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

3.1.4 链表Lists

Python knows a number of *compound* data types, used to group together other values. The most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets. List items need not all have the same type.

Python **已经有了几个复合数据类型，用于组织其它的值。最通用的是链表，它写为中括之间用逗号分隔的一列值（子项），链表的子项不一定是同一类型的值。**

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

Like string indices, list indices start at 0, and lists can be sliced, concatenated and so on:

像字符串一样，链表也以零开始，可以被切片，联接，等等：

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

Unlike strings, which are *immutable*, it is possible to change individual elements of a list:

与不变的字符串不同，链表可以改变每个独立元素的值：

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

Assignment to slices is also possible, and this can even change the size of the list:

可以进行切片操作，甚至还可以改变链表的大小：

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

The built-in function `len()` also applies to lists:

内置函数len()**也同样可以用于链表：**

```
>>> len(a)
8
```

It is possible to nest lists (create lists containing other lists), for example:

它也可以嵌套链表（在链表中创建其它链表），例如：

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')     # See section 5.1
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

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

注意最后一个例子，p[1]**和**q**实际上指向同一个对象！我们在后面会讲到对象语法。**

3.2 开始编程First Steps Towards Programming

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:

当然，我们可以用 Python **做比**2**加**2**更复杂的事。例如，我们可以用以下的方法输出菲波那契（**Fibonacci**）序列的子序列：**

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

This example introduces several new features.

示例中介绍了一些新功能：

- The first line contains a *multiple assignment*: the variables `a` and `b` simultaneously get the new values 0 and 1. On the last line this is used again, demonstrating that the expressions on the right-hand side are all evaluated first before any of the assignments take place. The right-hand side expressions are evaluated from the left to the right.

  第一行包括了复合参数：变量a **和**b **同时被赋值为**0**和**1**。最后一行又一次使用了这种技术，证明了在赋值之前表达式右边先进行了运算。右边的表达式从左到右运算。**

- 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).

while 循环运行在条件为真时执行。在Python 中，类似于C 任何非零值为真，零为假。这个条件也可以用于字符串或链表，事实上于对任何序列类型，长度非零时为真，空序列为假。示例所用的是一个简单的比较。标准的比较运算符写法和C 相同：< （小于），> （大于），== （等于），<= （小于等于），>=（大于等于）和!= （不等于）。

- The *body* of the loop is *indented*: indentation is Python's way of grouping statements. Python does not (yet!) provide an intelligent input line editing facility, so you have to type a tab or space(s) for each indented line. In practice you will prepare more complicated input for Python with a text editor; most text editors have an auto-indent facility. When a compound statement is entered interactively, it must be followed by a blank line to indicate completion (since the parser cannot guess when you have typed the last line). Note that each line within a basic block must be indented by the same amount.

  循环体是缩进的：缩进是Python 对语句分组的方法。Python 还没有提供一个智能编辑功能，你要在每一个缩进行输入一个tab 或（一个或多个）空格。实际上你可能会准备更为复杂的文本编辑器来编写你的Python 程序，大多数文本编辑器都提供了自动缩进功能。交互式的输入一个复杂语句时，需要用一个空行表示完成（因为解释器没办法猜出你什么时候输入最后一行）。需要注意的是每一行都要有相同的缩进来标识这是同一个语句块。

- 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:

  print 语句打印给定表达式的值。它与你仅仅输入你需要的表达式（就像前面的计算器示例）不同，它可以同时输出多个表达式。字符串输出时没有引号，各项之间用一个空格分开，你可以很容易区分它们，如下所示：

```
>>> 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:

  print 语句末尾的逗号避免了输出中的换行：

```
>>> 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.

  需要注意的是，如果最后一行仍没有写完，解释器会在它打印下一个命令时插入一个新行。

第四章

深入流程控制More Control Flow Tools

Besides the `while` statement just introduced, Python knows the usual control flow statements known from other languages, with some twists.

除了前面介绍的while **语句，**Python **还从别的语言中借鉴了一些流程控制功能，并有所改变。**

4.1 `if` 语句`if` Statements

Perhaps the most well-known statement type is the `if` statement. For example:

也许最有名的是if **语句。例如：**

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

可能会有0 **或很多个**elif **部分，**else **是可选的。关键字 "** elif **" 是 "** else if **" 的缩写，这个可以有效避免过深的缩进。**if ... elif ... elif ... **序列用于替代其它语言中的**switch **或**case **语句。**

4.2 `for` 语句`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):

Python**中的**for **语句和你在**C **或**Pascal **中使用的略有不同。通常的循环可能会依据一个等差数值步进过程（如**Pascal**）或由用户来定义迭代步骤和中止条件（如**C **），**Python **的**for **语句依据任意序列（链表或字符串）中的子项，按它们在序列中的顺序来进行迭代。例如（没有暗指）：**

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

It is not safe to modify the sequence being iterated over in the loop (this can only happen for mutable sequence types, such as lists). If you need to modify the list you are iterating over (for example, to duplicate selected items) you must iterate over a copy. The slice notation makes this particularly convenient:

在迭代过程中修改迭代序列不安全（只有在使用链表这样的可变序列时才会有这样的情况）。如果你想要修改你迭代的序列（例如，复制选择项），你可以迭代它的复本。通常使用切片标识就可以很方便的做到这一点：

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

4.3 `range()` 函数The `range()` Function

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:

如果你需要一个数值序列，内置函数range()**可能会很有用，它生成一个等差级数链表。**

```
>>> 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, exactly 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(10)**生成了一个包含**10**个值的链表，它准确的用链表的索引值填充了这个长度为**10**的列表，所生成的链表中不包括范围中的结束值。也可以让**range**操作从另一个数值开始，或者可以指定一个不同的步进值（甚至是负数，有时这也被称为“步长”）：**

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

To iterate over the indices of a sequence, combine `range()` and `len()` as follows:

需要迭代链表索引的话，如下所示结合使用range()**和**len()**：**

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

4.4 break 和continue 语句, 以及循环中的else 子句break and continue Statements, and else Clauses on Loops

The break statement, like in C, breaks out of the smallest enclosing for or while loop.

break 语句和C 中的类似，用于跳出最近的一级for 或while 循环。

The continue statement, also borrowed from C, continues with the next iteration of the loop.

continue 语句是从C 中借鉴来的，它表示循环继续执行下一次迭代。

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:

循环可以有一个else 子句;它在循环迭代完整个列表（对于for ）或执行条件为false （对于while ）时执行，但循环被break 中止的情况下不会执行。以下搜索素数的示例程序演示了这个子句：

```
>>> 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
```

4.5 pass 语句pass Statements

The pass statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

pass 语句什么也不做。它用于那些语法上必须要有什么语句，但程序什么也不做的场合，例如：

```
>>> while True:
...       pass # Busy-wait for keyboard interrupt
...
```

4.6 定义函数Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

我们可以编写一个函数来生成给定上界的菲波那契数列：

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> # Now call the function we just defined:
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword `def` introduces a function *definition*. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented. The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or *docstring*.

关键字def **引入了一个函数定义。在其后必须跟有函数名和包括形式参数的圆括号。函数体语句从下一行开始，必须是缩进的。函数体的第一行可以是一个字符串值，这个字符串是该函数的**(**文档字符串（**documentation string**））**，**也可称作***docstring* **。**

There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's good practice to include docstrings in code that you write, so try to make a habit of it.

有些文档字符串工具可以在线处理或打印文档，或让用户交互的浏览代码;**在代码中加入文档字符串是一个好的作法，应该养成这个习惯。**

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 global symbol table, and then 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.

调用函数时会为局部变量引入一个新的符号表。所有的局部变量都存储在这个局部符号表中。引用参数时，会先从局部符号表中查找，然后是全局符号表，然后是内置命名表。因此，全局参数虽然可以被引用，但它们不能在函数中直接赋值（除非它们用global **语句命名）。**

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using *call by value* (where the *value* is always an object *reference*, not the value of the object).[1] When a function calls another function, a new local symbol table is created for that call.

函数引用的实际参数在函数调用时引入局部符号表，因此，实参总是传值调用（这里的值总是一个对象引用，而不是该对象的值）。4.1 **一个函数被另一个函数调用时，一个新的局部符号表在调用过程中被创建。**

[1]Actually, *call by object reference* would be a better description, since if a mutable object is passed, the caller will see any changes the callee makes to it (items inserted into a list).

A function definition introduces the function name in the current symbol table. The value of the function name has a type that is recognized by the interpreter as a user-defined function. This value can be assigned to another name which can then also be used as a function. This serves as a general renaming mechanism:

函数定义在当前符号表中引入函数名。作为用户定义函数，函数名有一个为解释器认可的类型值。这个值可以赋给其它命名，使其能够作为一个函数来使用。这就像一个重命名机制：

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

You might object that `fib` is not a function but a procedure. In Python, like in C, procedures are just functions that don't return a value. In fact, technically speaking, procedures 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:

你可能认为fib**不是一个函数（**function**），而是一个过程（**procedure**）。**Python **和**C **一样，过程只是一个没有返回值的函数。实际上，从技术上讲，过程也有一个返回值，虽然是一个不讨人喜欢的。这个值被称为**None **（这是一个内置命名）。如果一个值只是**None **的话，通常解释器不会写一个**None **出来，如果你真想要看它的话，可以这样做：**

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

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

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

```
>>> 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 b < n:
...         result.append(b)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

This example, as usual, demonstrates some new Python features:

和以前一样，这个例子演示了一些新的 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 procedure also returns `None`.

  return **语句从函数中返回一个值，不带表达式的**return **返回**None **。过程结束后也会返回**None **。**

- The statement `result.append(b)` calls a *method* of the list object `result`. A method is a function that 'belongs' to an object and is named `obj.methodname`, where `obj` is some object (this may be an expression), and `methodname` is the name of a method that is defined by the object's type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and methods, using *classes*, as discussed later in this tutorial.) The method `append()` shown in the example, is defined for list objects; it adds a new element at the end of the list. In this example it is equivalent to '`result = result + [b]`', but more efficient.

语句`result.append(b)` 称为链表对象result **的一个(方法（**method **））。方法是一个“ 属于 ” 某个对象的函数，它被命名为**(obj.methodename)**，这里的**(obj) **是某个对象（可能是一个表达式），**(methodename) **是某个在该对象类型定义中的方法的命名。不同的类型定义不同的方法。不同类型可能有同样名字的方法，但不会混淆。（当你定义自己的对象类型和方法时，可能会出现这种情况，本指南后面的章节会介绍如何使用）。示例中演示的**append() **方法由链表对象定义，它向链表中加入一个新元素。在示例中它等同于‘(**’"result = result + [b]")**，不过效率更高。**

4.7 深入函数定义More on Defining Functions

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

有时需要定义参数个数可变的函数。有三个方法可以达到目的，我们可以组合使用它们。

4.7.1 参数默认值Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined

最有用的形式是给一个或多个参数指定默认值。这样创建的函数可以在调用时使用较少的参数。

```
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
```

This function can be called either like this: `ask_ok('Do you really want to quit?')` or like this: `ask_ok('OK to overwrite the file?', 2)`.

这个函数还可以用以下的方式调用：`ask_ok('Do you really want to quit?')`**，或者像这样：**`ask_ok('OK to overwrite the file?', 2)`**。**

The default values are evaluated at the point of function definition in the *defining* scope, so that

默认值在函数定义段被解析，如下所示：

```
i = 5

def f(arg=i):
    print arg

i = 6
f()
```

will print `5`.

将得到5.

Important warning: The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

重要警告：默认值只会解析一次。当默认值是一个可变对象，诸如链表、字典或大部分类实例时，会产生一些差异。例如，以下函数在后继的调用中会累积它的参数值：

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

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

This will print

这会打印出：

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

If you don’t want the default to be shared between subsequent calls, you can write the function like this instead:

如果你不想在不同的函数调用之间共享参数默认值，可以如下面的实例一样编写函数：

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

4.7.2 参数关键字Keyword Arguments

Functions can also be called using keyword arguments of the form ‘*keyword* = *value*’. For instance, the following function:

函数可以通过参数关键字的形式来调用，形如‘*keyword* = *value*’**。例如，以下的函数：**

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

could be called in any of the following ways:

可以用以下的任一方法调用：

```
parrot(1000)
parrot(action = ’VOOOOOM’, voltage = 1000000)
parrot(’a thousand’, state = ’pushing up the daisies’)
parrot(’a million’, ’bereft of life’, ’jump’)
```

but the following calls would all be invalid:

不过以下几种调用是无效的：

```
parrot()                     # required argument missing
parrot(voltage=5.0, ’dead’)  # non-keyword argument following keyword
parrot(110, voltage=220)     # duplicate value for argument
parrot(actor=’John Cleese’)  # unknown keyword
```

In general, an argument list must have any positional arguments followed by any keyword arguments, where the keywords must be chosen from the formal parameter names. It’s not important whether a formal parameter has a default value or not. No argument may receive a value more than once — formal parameter names corresponding to positional arguments cannot be used as keywords in the same calls. Here’s an example that fails due to this restriction:

通常，参数列表中的每一个关键字都必须来自于形式参数，每个参数都有对应的关键字。形式参数有没有默认值并不重要。实际参数不能一次赋多个值－－形式参数不能在同一次调用中同时使用位置和关键字绑定值。这里有一个例子演示了在这种约束下所出现的失败情况：

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

When a final formal parameter of the form `**`*name* is present, it receives a dictionary containing all keyword arguments whose keyword doesn’t correspond 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:

引入一个形如`**`name 的参数时，它接收一个字典，该字典包含了所有未出现在形式参数列表中的关键字参数。这里可能还会组合使用一个形如`*`*name* 的形式参数，它接收一个拓扑（下一节中会详细介绍），包含了所有没有出现在形式参数列表中的参数值。（`*`*name* 必须在`**`*name* 之前出现）例如，我们这样定义一个函数：

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

It could be called like this:

它可以像这样调用：

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

and of course it would print:

当然它会按如下内容打印：

```
-- 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 `sort()` method of the list of keyword argument names is called before printing the contents of the `keywords` dictionary; if this is not done, the order in which the arguments are printed is undefined.

注意`sort()`**方法在关键字字典内容打印前被调用，否则的话，打印参数时的顺序是未定义的。**

4.7.3 可变参数表Arbitrary Argument Lists

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple. Before the variable number of arguments, zero or more normal arguments may occur.

最后，一个最不常用的选择是可以让函数调用可变个数的参数。这些参数被包装进一个拓扑。在这些可变个数的参数之前，可以有零到多个普通的参数：

```
def fprintf(file, format, *args):
    file.write(format % args)
```

4.7.4 参数列表的分拆Unpacking Argument Lists

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:

另有一种相反的情况: **当你要传递的参数已经是一个列表但要调用的函数却接受分开一个个的参数值**. **这时候你要把已有的列表拆开来**. **例如内建函数**`range()` **要独立的***start*, *stop* **参数**. **你可以在调用函数时加一个**`*`**操作符来自动把参数列表拆开**:

```
>>> 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]
```

4.7.5 Lambda 形式Lambda Forms

By popular demand, a few features commonly found in functional programming languages and Lisp have been added to Python. With the `lambda` keyword, small anonymous functions can be created. Here’s a function that returns the sum of its two arguments: ‘`lambda a, b:  a+b`’. Lambda forms can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda forms can reference variables from the containing scope:

出于适当的需要，有几种通常在功能性语言和Lisp 中出现的功能加入到了Python 。通过`lambda`关键字，可以创建很小的匿名函数。这里有一个函数返回它的两个参数的和：‘`lambda a, b: a+b`’。Lambda 形式可以用于任何需要的函数对象。出于语法限制，它们只能有一个单独的表达式。语义上讲，它们只是普通函数定义中的一个语法技巧。类似于嵌套函数定义，lambda 形式可以从包含范围内引用变量：

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

4.7.6 文档字符串Documentation Strings

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

这里介绍文档字符串的概念和格式。

The first line should always be a short, concise summary of the object’s purpose. For brevity, it should not explicitly state the object’s name or type, since these are available by other means (except if the name happens to be a verb describing a function’s operation). This line should begin with a capital letter and end with a period.

第一行应该是关于对象用途的简介。简短起见，不用明确的陈述对象名或类型，因为它们可以从别的途径了解到（除非这个名字碰巧就是描述这个函数操作的动词）。这一行应该以大写字母开头，以句号结尾。

If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description. The following lines should be one or more paragraphs describing the object’s calling conventions, its side effects, etc.

如果文档字符串有多行，第二行应该空出来，与接下来的详细描述明确分隔。接下来的文档应该有一或多段描述对象的调用约定、边界效应等。

The Python parser does not strip indentation from multi-line string literals in Python, so tools that process documentation have to strip indentation if desired. This is done using the following convention. The first non-blank line *after* the first line of the string determines the amount of indentation for the entire documentation string. (We can’t use the first line since it is generally adjacent to the string’s opening quotes so its indentation is not apparent in the string literal.) Whitespace “equivalent” to this indentation is then stripped from the start of all lines of the string. Lines that are indented less should not occur, but if they occur all their leading whitespace should be stripped. Equivalence of whitespace should be tested after expansion of tabs (to 8 spaces, normally).

Python的解释器不会从多行的文档字符串中去除缩进，所以必要的时候应当自己清除缩进。这符合通常的习惯。第一行*之后*的第一个非空行决定了整个文档的缩进格式。（我们不用第一行是因为它通常紧靠着起始的引号，缩进格式显示的不清楚。）留白“ 相当于 ”是字符串的起始缩进。每一行都不应该有缩进，如果有缩进的话，所有的留白都应该清除掉。留白的长度应当等于扩展制表符的宽度（通常是8个空格）。

Here is an example of a multi-line docstring:

以下是一个多行文档字符串的示例：

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

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

第五章

数据结构Data Structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

本章节深入讲述一些你已经学习过的东西，并且还加入了新的内容。

5.1 深入链表More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

链表类型有很多方法，这里是链表类型的所有方法：

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

  把一个元素添加到链表的结尾，相当于`a[len(a):] = [x]`

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

  通过添加指定链表的所有元素来扩充链表，相当于`a[len(a):] = L`**。**

`insert`(*i, x*)
: Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, `*x*`)` inserts at the front of the list, and `a.insert(len(a), `*x*`)` is equivalent to `a.append(`*x*`)`.

  在指定位置插入一个元素。第一个参数是准备插入到其前面的那个元素的索引，例如`a.insert(0, x)` **会插入到整个链表之前，而**`a.insert(len(a), x)` **相当于**`a.append(x)`**。**

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

  删除链表中值为*x*的第一个元素。如果没有这样的元素，就会返回一个错误。

`pop`([*i*])
: Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` returns the last item in the list. The item is also removed from the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the *Python Library Reference*.)

  从链表的指定位置删除元素，并将其返回。如果没有指定索引，`a.pop()` **返回最后一个元素。元素随即从链表中被删除。（方法中*i*两边的方括号表示这个参数是可选的，而不是要求你输入一对方括号，你会经常在*Python 库参考手册*中遇到这样的标记。）**

`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.

  返回链表中第一个值为*x*的元素的索引。如果没有匹配的元素就会返回一个错误。

count(*x*)
Return the number of times *x* appears in the list.

返回*x*在链表中出现的次数。

sort()
Sort the items of the list, in place.

对链表中的元素进行适当的排序。

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

倒排链表中的元素。

An example that uses most of the list methods:

下面这个示例演示了链表的大部分方法：

```
>>> a = [66.6, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.6), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.6, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.6, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.6]
>>> a.sort()
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
```

5.1.1 把链表当作堆栈使用Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved ("last-in, first-out"). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index. For example:

链表方法使得链表可以很方便的做为一个堆栈来使用，堆栈作为特定的数据结构，最先进入的元素最后一个被释放（后进先出）。用`append()`方法可以把一个元素添加到堆栈顶。用不指定索引的`pop()`方法可以把一个元素从堆栈顶释放出来。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 把链表当作队列使用Using Lists as Queues

You can also use a list conveniently as a queue, where the first element added is the first element retrieved ("first-in, first-out"). To add an item to the back of the queue, use `append()`. To retrieve an item from the front of the queue, use `pop()` with `0` as the index. For example:

你也可以把链表当做队列使用，队列作为特定的数据结构，最先进入的元素最先释放（先进先出）。使用`append()`方法可以把元素添加到队列最后，以0为参数调用`pop()` 方法可以把最先进入的元素释放出来。例如：

```
>>> queue = ["Eric", "John", "Michael"]
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.pop(0)
'Eric'
>>> queue.pop(0)
'John'
>>> queue
['Michael', 'Terry', 'Graham']
```

5.1.3 函数化编程工具Functional Programming Tools

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

对于链表来讲，有三个内置函数非常有用：`filter()`，`map()`，和`reduce()`。

'`filter(`*function*, *sequence*`)`' returns a sequence (of the same type, if possible) consisting of those items from the sequence for which *function*`(`*item*`)` is true. For example, to compute some primes:

'`filter(function, sequence)`' **返回一个序列（*sequence*），包括了给定序列中所有调用*function*(*item*)后返回值为true的元素。（如果可能的话，会返回相同的类型）。例如，以下程序可以计算部分素数：**

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

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

'map(*function*, *sequence*)' **为每一个元素依次调用***function*(*item*) **并将返回值组成一个链表返回。例如，以下程序计算立方：**

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

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:

可以传入多个序列，函数也必须要有对应数量的参数，执行时会依次用各序列上对应的元素来调用函数（如果某些序列比其它的短，就用None来代替）。如果把 None**做为一个函数传入，则直接返回参数做为替代。组合这两种情况，我们会发现"** map(None, list1, list2) **"是把一对序列变成元素对序列的便捷方式。例如：**

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

'reduce(*func*, *sequence*)' returns a single value constructed by calling the binary function *func* 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:

'reduce(*func*, *sequence*)' **返回一个单值，它是这样构造的：首先以序列的前两个元素调用函数，再以返回值和第三个参数调用，依次执行下去。例如，以下程序计算**1**到**10**的整数之和：**

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

如果序列中只有一个元素，就返回它，如果序列是空的，就抛出一个异常。

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,

可以传入第三个参数做为初始值。如果序列是空的，就返回初始值，否则函数会先接收初始值和序列的第一个元素，然后是返回值和下一个元素，依此类推。例如：

```
>>> 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. New in version 2.3.

不要像示例中这样定义`sum()`**：因为合计数值是一个通用的需求，在**2.3**版中，提供了内置的**`sum(`*sequence*`)` **函数。** New in version 2.3.

5.1.4 链表推导式List Comprehensions

List comprehensions provide a concise way to create lists without resorting to use of `map()`, `filter()` and/or `lambda`. The resulting list definition tends often to be clearer than lists built using those constructs. Each list comprehension consists of an expression followed by a `for` clause, then zero or more `for` or `if` clauses. The result will be a list resulting from evaluating the expression in the context of the `for` and `if` clauses which follow it. If the expression would evaluate to a tuple, it must be parenthesized.

链表推导式提供了一个创建链表的简单途径，无需使用`map()`**，**`filter()` **以及**`lambda` **。返回链表的定义通常要比创建这些链表更清晰。每一个链表推导式包括在一个**`for` **语句之后的表达式，零或多个**`for` **或**if**语句。返回值是由**`for` **或**`if` **子句之后的表达式得到的元素组成的链表。如果想要得到一个元组，必须要加上括号。**

```
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
>>> [x, x**2 for x in vec]  # error - parens required for tuples
  File "<stdin>", line 1, in ?
    [x, x**2 for x in vec]
               ^
SyntaxError: invalid syntax
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

List comprehensions are much more flexible than `map()` and can be applied to functions with more than one argument and to nested functions:

链表推导式比`map()`**更复杂，可调用多个参数和嵌套函数。**

```
>>> [str(round(355/113.0, i)) for i in range(1,6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.2 del 语句

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

有一个方法可从链表中删除指定索引的元素：del语句。这个方法也可以从链表中删除切片（之前我们是把一个空链表赋给切片）。例如：

```
>>> a = [-1, 1, 66.6, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.6, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.6, 1234.5]
```

del can also be used to delete entire variables:

del **也可以用于删除整个变量：**

```
>>> del a
```

Referencing the name a hereafter is an error (at least until another value is assigned to it). We'll find other uses for del later.

此后再引用这个名字会发生错误（至少要到给它赋另一个值为止）。后面我们还会发现del的其它用法。

5.3 元组（Tuples）和序列（Sequences ）Tuples and Sequences

We saw that lists and strings have many common properties, such as indexing and slicing operations. They are two examples of *sequence* data types. Since Python is an evolving language, other sequence data types may be added. There is also another standard sequence data type: the *tuple*.

我们知道链表和字符串有很多通用的属性，例如索引和切片操作。它们是序列类型中的两种。因为 Python**是一个在不停进化的语言，也可能会加入其它的序列类型，这里有另一种标准序列类型：元组。**

A tuple consists of a number of values separated by commas, for instance:

一个元组由数个逗号分隔的值组成，例如：

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

As you see, on output tuples are alway enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression).

如你所见，元组在输出时总是有括号的，以便于正确表达嵌套结构。在输入时可能有或没有括号都可以，不过经常括号都是必须的（如果元组是一个更大的表达式的一部分）。

Tuples have many uses. For example: (x, y) coordinate pairs, employee records from a database, etc. Tuples, like strings, are immutable: it is not possible to assign to the individual items of a tuple (you can simulate much of the same effect with slicing and concatenation, though). It is also possible to create tuples which contain mutable objects, such as lists.

元组有很多用途。例如（x, y**）坐标点，数据库中的员工记录等等。元组就像字符串，不可改变：不能给元组的一个独立的元素赋值（尽管你可以通过联接和切片来模仿）。也可以通过包含可变对象来创建元组，例如链表。**

A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these. Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses). Ugly, but effective. For example:

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

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

The statement `t = 12345, 54321, 'hello!'` is an example of *tuple packing*: the values `12345`, `54321` and `'hello!'` are packed together in a tuple. The reverse operation is also possible:

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

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

This is called, appropriately enough, *sequence unpacking*. Sequence unpacking requires that the list of variables on the left 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!

这个调用被称为*序列拆封*非常合适。序列拆封要求左侧的变量数目与序列的元素个数相同。要注意的是可变参数（multiple assignment **）其实只是元组封装和序列拆封的一个结合！**

There is a small bit of asymmetry here: packing multiple values always creates a tuple, and unpacking works for any sequence.

这里有一点不对称：封装多重参数通常会创建一个元组，而拆封操作可以作用于任何序列。

5.4 Sets

Python also includes a data type for *sets*. A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Python **也有对应于集合的数据类型**. **集合代表一群没有特定排列次序**, **且并不重覆的元素**. **合可以应用到会员身份验证**, **或是剔除重覆项目上**. **集合类也可以作数学上的**union, intersection, difference, symmetric difference **操作**.

Here is a brief demonstration:

以下为简单示范:

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

5.5 字典Dictionaries

Another useful data type built into Python is the *dictionary*. Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can't use lists as keys, since lists can be modified in place using their `append()` and `extend()` methods, as well as slice and indexed assignments.

另一个非常有用的Python**内建数据类型是*字典*。字典在某些语言中可能称为“联合内存”（**“associative memories”**）或“联合数组”（**“associative arrays”**）。序列是以连续的整数为索引，与此不同的是，字典以*关键字*为索引，关键字可以是任意不可变类型，通常用字符串或数值。如果元组中只包含字符串和数字，它可以做为关键字，如果它直接或间接的包含了可变对象，就不能当做关键字。不能用链表做关键字，因为链表可以用它们的**`append()`**和**`extend()`**方法，或者用切片、或者通过检索变量来即时改变。**

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.

理解字典的最佳方式是把它看做无序的*关键字：值* 对（key:value pairs **）集合，关键字必须是互不相同的（在同一个字典之内）。一对大括号创建一个空的字典：**`{}`**。初始化链表时，在大括号内放置一组逗号分隔的关键字：值对，这也是字典输出的方式。**

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with `del`. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

字典的主要操作是依据关键字来存储和析取值。也可以用`del`**来删除关键字：值对。如果你用一个已经存在的关键字存储值，以前为该关键字分配的值就会被遗忘。试图析取从一个不存在的关键字中读取值会导致错误。**

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

字典的`keys()`方法返回由所有关键字组成的链表，该链表的顺序不定（如果你需要它有序，只能调用关键字链表的`sort()`方法）。使用字典的`has_key()`方法可以检查字典中是否存在某一关键字。

Here is a small example using a dictionary:

这是一个关于字典应用的小示例：

```
>>> 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']
>>> tel.has_key('guido')
True
```

The `dict()` constructor builds dictionaries directly from lists of key-value pairs stored as tuples. When the pairs form a pattern, list comprehensions can compactly specify the key-value list.

链表中存储关键字-值对元组的话，字典可以从中直接构造。关键字-值对来自一个模式时，可以用链表推导式简单的表达关键字-值链表。

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

5.6 循环技巧Looping Techniques

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

在字典中循环时，关键字和对应的值可以使用`iteritems()`方法同时解读出来。

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

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the `enumerate()` function.

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

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

To loop over two or more sequences at the same time, the entries can be paired with the `zip()` function.

同时循环两个或更多的序列，可以使用`zip()` 整体解读。

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

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the `reversed()` function.

打算反向循环序列时, **可以先计算出序列**, **然后调用**`reversed()` **函数**.

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

To loop over a sequence in sorted order, use the `sorted()` function which returns a new sorted list while leaving the source unaltered.

要按次序循环一个序列, **用**`sorted()` **函数**, **它传回一个新的排好序的序列**. **而原先的序列则保留不变**.

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

5.7 深入条件控制More on Conditions

The conditions used in `while` and `if` statements above can contain other operators besides comparisons.

用于`while` **和**`if` **语句的条件包括了比较之外的操作符。**

The comparison operators `in` and `not in` check whether a value occurs (does not occur) in a sequence. The operators `is` and `is not` compare whether two objects are really the same object; this only matters for mutable objects like lists. All comparison operators have the same priority, which is lower than that of all numerical operators.

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

Comparisons can be chained. For example, `a < b == c` tests whether `a` is less than `b` and moreover `b` equals `c`.

比较操作可以传递。例如`a < b == c` 审核是否`a` 小于`b` 并`b` 等于`c`。

Comparisons may be combined by the Boolean operators `and` and `or`, and the outcome of a comparison (or of any other Boolean expression) may be negated with `not`. These all have lower priorities than comparison operators again; between them, `not` has the highest priority, and `or` the lowest, so that `A and not B or C` is equivalent to `(A and (not B)) or C`. Of course, parentheses can be used to express the desired composition.

比较操作可以通过逻辑操作符`and` 和`or`组合，比较的结果可以用`not` 来取反义。这些操作符的优先级又低于比较操作符，在它们之中，`not` 具有最高的优先级，`or` 优先级最低，所以`A and not B or C` 等于`(A and (not B)) or C`。当然，表达式可以用期望的方式表示。

The Boolean operators `and` and `or` are so-called *short-circuit* operators: their arguments are evaluated from left to right, and evaluation stops as soon as the outcome is determined. For example, if `A` and `C` are true but `B` is false, `A and B and C` does not evaluate the expression `C`. In general, the return value of a short-circuit operator, when used as a general value and not as a Boolean, is the last evaluated argument.

逻辑操作符`and` 和`or` 也称作*短路*操作符：它们的参数从左向右解析，一旦结果可以确定就停止。例如，如果`A` 和`C` 为真而`B` 为假，`A and B and C` 不会解析`C`。作用于一个普通的非逻辑值时，短路操作符的返回值通常是最后一个变量。

It is possible to assign the result of a comparison or other Boolean expression to a variable. For example,

可以把比较或其它逻辑表达式的返回值赋给一个变量，例如：

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

Note that in Python, unlike C, assignment cannot occur inside expressions. C programmers may grumble about this, but it avoids a common class of problems encountered in C programs: typing = in an expression when == was intended.

需要注意的是Python与C不同，在表达式内部不能赋值。C 程序员经常对此抱怨，不过它避免了一类在C程序中司空见惯的错误：想要在解析式中使== 时误用了= 操作符。

5.8 比较序列和其它类型Comparing Sequences and Other Types

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 with the same types:

序列对象可以与相同类型的其它对象比较。比较操作按*字典序* 进行：首先比较前两个元素，如果不同，就决定了比较的结果；如果相同，就比较后两个元素，依此类推，直到所有序列都完成比较。如果两个元素本身就是同样类型的序列，就递归字典序比较。如果两个序列的所有子项都相等，就认为序列相等。如果一个序列是另一个序列的初始子序列，较短的一个序列就小于另一个。字符串的字典序按照单

字符的ASCII **顺序。下面是同类型序列之间比较的一些例子：**

```
(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. Mixed numeric types are compared according to their numeric value, so 0 equals 0.0, etc.[1]

需要注意的是不同类型的对象比较是合法的。输出结果是确定而非任意的：类型按它们的名字排序。因而，一个链表（list**）总是小于一个字符串（**string**），一个字符串（**string**）总是小于一个元组（**tuple**）等等。数值类型比较时会统一它们的数据类型，所以**0**等于**0.0**，等等。**[2]

[1]The rules for comparing objects of different types should not be relied upon; they may change in a future version of the language.
[2]**不同类型对象的比较规则不依赖于此，它们有可能会在**Python**语言的后继版本中改变。**

第六章

模块Modules

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a *script*. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

如果你退出Python**解释器重新进入，以前创建的一切定义（变量和函数）就全部丢失了。因此，如果你想写一些长久保存的程序，最好使用一个文本编辑器来编写程序，把保存好的文件输入解释器。我们称之为创建一个*脚本*。程序变得更长一些了，你可能为了方便维护而把它分离成几个文件。你也可能想要在几个程序中都使用一个常用的函数，但是不想把它的定义复制到每一个程序里。**

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

为了满足这些需要，Python**提供了一个方法可以从文件中获取定义，在脚本或者解释器的一个交互式实例中使用。这样的文件被称为*模块*；模块中的定义可以导入到另一个模块或主模块中（在脚本执行时可以调用的变量集位于最高级，并且处于计算器模式）**

A module is a file containing Python definitions and statements. The file name is the module name with the suffix '.py' appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use your favorite text editor to create a file called 'fibo.py' in the current directory with the following contents:

模块是包括Python**定义和声明的文件。文件名就是模块名加上**'.py' **后缀。模块的模块名（做为一个字符串）可以由全局变量**`__name__` **得到。例如，你可以用自己惯用的文件编辑器在当前目录下创建一个叫**'fibo.py' **的文件，录入如下内容：**

```
# 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
```

Now enter the Python interpreter and import this module with the following command:

现在进入Python**解释器，用如下命令导入这个模块**

```
>>> import fibo
```

This does not enter the names of the functions defined in `fibo` directly in the current symbol table; it only enters the module name `fibo` there. Using the module name you can access the functions:

这样做不会直接把`fibo` **中的函数导入当前的语义表；，它只是引入了模块名**`fibo` **。你可以通过模块名按如下方式访问这个函数：**

```
>>> 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'
```

If you intend to use a function often you can assign it to a local name:

如果你想要直接调用函数，通常可以给它赋一个本地名称：

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

6.1 深入模块More on Modules

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module is imported somewhere.[1]

模块可以像函数定义一样包含执行语句。这些语句通常用于初始化模块。它们只在模块第一次导入时执行一次。[2]

[1] In fact function definitions are also 'statements' that are 'executed'; the execution enters the function name in the module's global symbol table.

[2] **事实上函数定义既是"声明"又是"可执行体"；执行体由函数在模块全局语义表中的命名导入。**（In fact function definitions are also 'statements' that are 'executed'; the execution enters the function name in the module's global symbol table.）

Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions, `modname.itemname`.

对应于定义模块中所有函数的全局语义表，每一个模块有自己的私有语义表。因此，模块作者可以在模块中使用一些全局变量，不会因为与用户的全局变量冲突而引发错误。另一方面，如果你确定你需要这个，可以像引用模块中的函数一样获取模块中的全局变量，形如：`modname.itemname`**。**

Modules can import other modules. It is customary but not required to place all `import` statements at the beginning of a module (or script, for that matter). The imported module names are placed in the importing module's global symbol table.

模块可以导入（import**）其它模块。习惯上所有的**`import`**语句都放在模块（或脚本，等等）的开头，但这并不是必须的。被导入的模块名入在本模块的全局语义表中。**

There is a variant of the `import` statement that imports names from a module directly into the importing module's symbol table. For example:

`import` **语句的一个变体直接从被导入的模块中导入命名到本模块的语义表中。例如：**

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

This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, `fibo` is not defined).

这样不会从局域语义表中导入模块名（如上所示，`fibo` **没有定义）。**

There is even a variant to import all names that a module defines:

这里还有一个变体从模块定义中导入所有命名：

```
>>> 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 (_).

这样可以导入所有除了以下划线(_)开头的命名。

6.1.1 模块搜索路径The Module Search Path

When a module named `spam` is imported, the interpreter searches for a file named 'spam.py' in the current directory, and then in the list of directories specified by the environment variable PYTHONPATH. This has the same syntax as the shell variable PATH, that is, a list of directory names. When PYTHONPATH is not set, or when the file is not found there, the search continues in an installation-dependent default path; on UNIX, this is usually '.:/usr/local/lib/python'.

导入一个叫`spam` **的模块时，解释器先在当前目录中搜索名为**'spam.py' **的文件，然后在环境变量**PYTHONPATH **表示的目录列表中搜索，然后是环境变量**PATH **中的路径列表。如果**PYTHONPATH **没有设置，或者文件没有找到，接下来搜索安装目录，在**UNIX**中，通常是**.:/usr/local/lib/python**。**

Actually, modules are searched in the list of directories given by the variable `sys.path` which is initialized from the directory containing the input script (or the current directory), PYTHONPATH and the installation-dependent default. This allows Python programs that know what they're doing to modify or replace the module search path. Note that because the directory containing the script being run is on the search path, it is important that the script not have the same name as a standard module, or Python will attempt to load the script as a module when that module is imported. This will generally be an error. See section 6.2, "Standard Modules," for more information.

实际上，解释器由sys.path 变量指定的路径目录搜索模块，该变量初始化时默认包含了输入脚本（或者当前目录），PYTHONPATH 和安装目录。这样就允许Python程序（原文如此，programs；我猜想应该是“ programer ”，程序员－－译者）了解如何修改或替换模块搜索目录。需要注意的是由于这些目录中包含有搜索路径中运行的脚本，所以这些脚本不应该和标准模块重名，否则在导入模块时Python会尝试把这些脚本当作模块来加载。这通常会引发一个错误。请参见6.2节“ 标准模块 ”以了解更多的信息。

6.1.2 “ 编译 ” Python文件“Compiled” Python files

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.

对于引用了大量标准模块的短程序，有一个提高启动速度的重要方法，如果在‘spam.py’ 所在的目录下存在一个名为‘spam.pyc’ 的文件，它会被视为spam 模块的预“ 编译 ”（“byte-compiled” ，二进制编译）版本。用于创建‘spam.pyc’ 的这一版‘spam.py’ 的修改时间记录在‘spam.pyc’文件中，如果两者不匹配，‘.pyc’ 文件就被忽略。

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.

通常你不需要为创建‘spam.pyc’ 文件做任何工作。一旦‘spam.py’ 成功编译，就会试图编译对应版本的‘spam.pyc’ 。如果有任何原因导致写入不成功，返回的‘spam.pyc’ 文件就会视为无效，随后即被忽略。‘spam.pyc’ 文件的内容是平台独立的，所以Python模块目录可以在不同架构的机器之间共享。

Some tips for experts:

部分高级技巧：

- 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.

  以**-O** 参数调用Python解释器时，会生成优化代码并保存在‘.pyo’ 文件中。现在的优化器没有太多帮助；它只是删除了断言（assert ）语句。使用**-O** 参参数，所有的代码都会被优化；.pyc 文件被忽略，.py文件被编译为优化代码。

- 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.

  向Python解释器传递两个**-O** 参数（**-OO**）会执行完全优化的二进制优化编译，这偶尔会生成错误的程序。现在的优化器，只是从二进制代码中删除了__doc__ 符串，生成更为紧凑的‘.pyo’ 文件。因为某些程序依赖于这些变量的可用性，你应该只在确定无误的场合使用这一选项。

- 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.

  来自‘.pyc’ 文件或‘.pyo’ 文件中的程序不会比来自‘.py’ 文件的运行更快；‘.pyc’ 或‘.pyo’ 文件只是在它们加载的时候更快一些。

- 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.

  通过脚本名在命令行运行脚本时，不会将为该脚本创建的二进制代码写入‘.pyc’ 或‘.pyo’ 文件。当然，把脚本的主要代码移进一个模块里，然后用一个小的解构脚本导入这个模块，就可以提高脚本的启动速度。也可以直接在命令行中指定一个‘.pyc’ 或‘.pyo’ 文件。

- 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.

  对于同一个模块（这里指例程'spam.py' **——译者），可以只有**'spam.pyc' **文件（或者**'spam.pyc'**，在使用-O 参数时）而没有**'spam.py' **文件。这样可以打包发布比较难于逆向工程的**Python**代码库。**

- The module `compileall` can create '.pyc' files (or '.pyo' files when **-O** is used) for all modules in a directory.

  `compileall` **模块可以为指定目录中的所有模块创建**'.pyc' **文件（或者使用**'.pyo' **参数创建**.pyo**文件）。**

6.2 标准模块Standard Modules

Python comes with a library of standard modules, described in a separate document, the *Python Library Reference* ("Library Reference" hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform For example, the `amoeba` module is only provided on systems that somehow support Amoeba primitives. One particular module deserves some attention: `sys`, which is built into every Python interpreter. The variables `sys.ps1` and `sys.ps2` define the strings used as primary and secondary prompts:

Python**带有一个标准模块库，并发布有独立的文档，名为***Python* ***库参考手册*** **（此后称其为"库参考手册"）。有一些模块内置于解释器之中，这些操作的访问接口不是语言内核的一部分，但是已经内置于解释器了。这既是为了提高效率，也是为了给系统调用等操作系统原生访问提供接口。这类模块集合是一个依赖于底层平台的配置选项。例如，**`amoeba` **模块只提供对**Amoeba **原生系统的支持。有一个具体的模块值得注意：**`sys` **，这个模块内置于所有的**Python**解释器。变量**`sys.ps1` **和**`sys.ps2`**定义了主提示符和副助提示符字符串：**

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

These two variables are only defined if the interpreter is in interactive mode.

这两个变量只在解释器的交互模式下有意义（此处原文为：These two variables are only defined if the interpreter is in interactive mode. **）。**

The variable `sys.path` is a list of strings that determine the interpreter's search path for modules. It is initialized to a default path taken from the environment variable PYTHONPATH, or from a built-in default if PYTHONPATH is not set. You can modify it using standard list operations:

变量`sys.path` **是解释器模块搜索路径的字符串列表。它由环境变量**PYTHONPATH **初始化，如果没有设定**PYTHONPATH **，就由内置的默认值初始化。你可以用标准的字符串操作修改它：**

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

6.3 `dir()` 函数`dir()` Function

The built-in function `dir()` is used to find out which names a module defines. It returns a sorted list of strings:

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

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

Without arguments, `dir()` lists the names you have defined currently:

无参数调用时，`dir()` 函数返回当前定义的名称：

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

Note that it lists all types of names: variables, modules, functions, etc.

该列表列出了所有类型的名称：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__`:

`dir()` 不会列出内置函数和变量名。如果你想列出这些内容，它们在标准模块__buildin__中定义：

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError',
 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
 'Exception', 'False', 'FloatingPointError', 'IOError', 'ImportError',
 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError', 'OverflowWarning',
 'PendingDeprecationWarning', 'ReferenceError',
 'RuntimeError', 'RuntimeWarning', 'StandardError', 'StopIteration',
 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError',
 'True', 'TypeError', 'UnboundLocalError', 'UnicodeError', 'UserWarning',
 'ValueError', 'Warning', 'ZeroDivisionError', '__debug__', '__doc__',
 '__import__', '__name__', 'abs', 'apply', 'bool', 'buffer',
 'callable', 'chr', 'classmethod', 'cmp', 'coerce', 'compile', 'complex',
 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod',
 'enumerate', 'eval', 'execfile', 'exit', 'file', 'filter', 'float',
 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id',
 'input', 'int', 'intern', 'isinstance', 'issubclass', 'iter',
 'len', 'license', 'list', 'locals', 'long', 'map', 'max', 'min',
 'object', 'oct', 'open', 'ord', 'pow', 'property', 'quit',
 'range', 'raw_input', 'reduce', 'reload', 'repr', 'round',
 'setattr', 'slice', 'staticmethod', 'str', 'string', 'sum', 'super',
 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

6.4 包Packages

Packages are a way of structuring Python's module namespace by using "dotted module names". For example, the module name `A.B` designates a submodule named 'B' in a package named 'A'. Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or the Python Imaging Library from having to worry about each other's module names.

包通常是使用用 " 圆点模块名 " 的结构化模块命名空间。例如，名为`A.B` 的模块表示了名为'B' 的包中名为'A' 的子模块。正如同用模块来保存不同的模块架构可以避免全局变量之间的相互冲突，使用圆点模块名保存像 NumPy **或**Python Imaging Library **之类的不同类库架构可以避免模块之间的命名冲突。**

Suppose you want to design a collection of modules (a "package") for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: '.wav', '.aiff', '.au'), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations. Here's a possible structure for your package (expressed in terms of a hierarchical filesystem):

假设你现在想要设计一个模块集（一个 " 包 " ）来统一处理声音文件和声音数据。存在几种不同的声音格式（通常由它们的扩展名来标识，例如：'.wav' **，**'.aiff' **，**'.au')**），于是，为了在不同类型的文件格式之间转换，你需要维护一个不断增长的包集合。可能你还想要对声音数据做很多不同的操作（例如混音，添加回声，应用平衡功能，创建一个人造效果），所以你要加入一个无限流模块来执行这些操作。你的包可能会是这个样子（通过分级的文件体系来进行分组）：**

```
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
              ...
```

When importing the package, Python searches through the directories on `sys.path` looking for the package subdirectory.

导入模块时，Python**通过**`sys.path` **中的目录列表来搜索存放包的子目录。**

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.

必须要有一个'__init__.py' **文件的存在，才能使**Python**视该目录为一个包；这是为了防止某些目录使用了**'`string`' **这样的通用名而无意中在随后的模块搜索路径中覆盖了正确的模块。最简单的情况下，**'__init__.py' **可以只是一个空文件，不过它也可能包含了包的初始化代码，或者设置了**`__all__` **变量，后面会有相关介绍。**

Users of the package can import individual modules from the package, for example:

包用户可以从包中导入合法的模块，例如：

```
import Sound.Effects.echo
```

This loads the submodule `Sound.Effects.echo`. It must be referenced with its full name.

这样就导入了`Sound.Effects.echo` **子模块。它必需通过完整的名称来引用。**

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

An alternative way of importing the submodule is:

导入包时有一个可以选择的方式：

```
from Sound.Effects import echo
```

This also loads the submodule `echo`, and makes it available without its package prefix, so it can be used as follows:

这样就加载了`echo` 子模块，并且使得它在没有包前缀的情况下也可以使用，所以它可以如下方式调用：

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

Yet another variation is to import the desired function or variable directly:

还有另一种变体用于直接导入函数或变量：

```
from Sound.Effects.echo import echofilter
```

Again, this loads the submodule `echo`, but this makes its function `echofilter()` directly available:

这样就又一次加载了`echo` 子模块，但这样就可以直接调用它的`echofilter()` 函数：

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

Note that when using `from` *package* `import` *item*, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The `import` statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an `ImportError` exception is raised.

需要注意的是使用`from` *package* `import` *item* 方式导入包时，这个子项（item）既可以是包中的一个子模块（或一个子包），也可以是包中定义的其它命名，像函数、类或变量。`import` 语句首先核对是否包中有这个子项，如果没有，它假定这是一个模块，并尝试加载它。如果没有找到它，会引发一个`ImportError` 异常。

Contrarily, when using syntax like `import` *item.subitem.subsubitem*, each item except for the last must be a package; the last item can be a module or a package but can't be a class or function or variable defined in the previous item.

相反，使用类似`import` *item.subitem.subsubitem* 这样的语法时，这些子项必须是包，最后的子项可以是包或模块，但不能是前面子项中定义的类、函数或变量。

6.4.1 以* 方式加载包Importing * From a Package

Now what happens when the user writes `from Sound.Effects import *`? Ideally, one would hope that this somehow goes out to the filesystem, finds which submodules are present in the package, and imports them all. Unfortunately, this operation does not work very well on Mac and Windows platforms, where the filesystem does not always have accurate information about the case of a filename! On these platforms, there is no guaranteed way to know whether a file 'ECHO.PY' should be imported as a module `echo`, `Echo` or `ECHO`. (For example, Windows 95 has the annoying practice of showing all file names with a capitalized first letter.) The DOS 8+3 filename restriction adds another interesting problem for long module names.

那么当用户写下`from Sound.Effects import *` 时会发生什么事？理想中，总是希望在文件系统中找出包中所有的子模块，然后导入它们。不幸的是，这个操作在Mac 和Windows 平台上工作的并不太好，这些文件系统的文件大小写并不敏感！在这些平台上没有什么方法可以确保一个叫'ECHO.PY' 的文件应该导入为模块`echo` 、`Echo` 或`ECHO` 。（例如，Windows 95有一个讨厌的习惯，它会把所有的文件名都显示为首字母大写的风格。）DOS 8+3文件名限制又给长文件名模块带来了另一个有趣的问题。

The only solution is for the package author to provide an explicit index of the package. The import statement uses the following convention: if a package's '__init__.py' code defines a list named `__all__`, it is taken to be the list of module names that should be imported when `from` *package* `import *` is encountered. It is up to the package author to keep this list up-to-date when a new version of the package is released. Package authors may also decide not to support it, if they don't see a use for importing * from their package. For example, the file 'Sounds/Effects/__init__.py' could contain the following code:

对于包的作者来说唯一的解决方案就是给提供一个明确的包索引。import **语句按如下条件进行转换：执行**`from` *package* `import *` **时，如果包中的**'__init__.py' **代码定义了一个名为**`__all__` **的链表，就会按照链表中给出的模块名进行导入。新版本的包发布时作者可以任意更新这个链表。如果包作者不想**import * **的时候导入他们的包中所有模块，那么也可能会决定不支持它（**import ***）。例如，**'Sounds/Effects/__init__.py' **这个文件可能包括如下代码：**

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

This would mean that `from Sound.Effects import *` would import the three named submodules of the `Sound` package.

这意味着`from Sound.Effects import *` **语句会从**`Sound` **包中导入以上三个已命名的子模块。**

If `__all__` is not defined, the statement `from Sound.Effects import *` does *not* import all submodules from the package `Sound.Effects` into the current namespace; it only ensures that the package `Sound.Effects` has been imported (possibly running its initialization code, '__init__.py') and then imports whatever names are defined in the package. This includes any names defined (and submodules explicitly loaded) by '__init__.py'. It also includes any submodules of the package that were explicitly loaded by previous import statements. Consider this code:

如果没有定义`__all__` **，**`from Sound.Effects import *` **语句不会从**`Sound.Effects` **包中导入所有的子模块。**Effects **导入到当前的命名空间，只能确定的是导入了**Sound.Effects **包（可能会运行**'__init__.py' **中的初始化代码）以及包中定义的所有命名会随之导入。这样就从**'__init__.py' **中导入了每一个命名（以及明确导入的子模块）。同样也包括了前述的**import**语句从包中明确导入的子模块，考虑以下代码：**

```
import Sound.Effects.echo
import Sound.Effects.surround
from Sound.Effects import *
```

In this example, the echo and surround modules are imported in the current namespace because they are defined in the `Sound.Effects` package when the `from...import` statement is executed. (This also works when `__all__` is defined.)

在这个例子中，echo**和**surround**模块导入了当前的命名空间，这是因为执行**`from...import` **语句时它们已经定义在**`Sound.Effects` **包中了（定义了**`__all__` **时也会同样工作）。**

Note that in general the practice of importing `*` from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions, and certain modules are designed to export only names that follow certain patterns.

需要注意的是习惯上不主张从一个包或模块中用import * **导入所有模块，因为这样的通常意味着可读性会很差。然而，在交互会话中这样做可以减少输入，相对来说确定的模块被设计成只导出确定的模式中命名的那一部分。**

Remember, there is nothing wrong with using `from Package import specific_submodule`! In fact, this is the recommended notation unless the importing module needs to use submodules with the same name from different packages.

记住，`from Package import specific_submodule` **没有错误！事实上，除非导入的模块需要使用其它包中的同名子模块，否则这是受到推荐的写法。**

6.4.2 内置包（Intra-package）参考Intra-package References

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.

子模块之间经常需要互相引用。例如，surround **模块可能会引用**echo **模块。事实上，这样的引用如此普遍，以致于**import **语句会先搜索包内部，然后才是标准模块搜索路径。因此**surround **模块可以简单的调用**import echo **或者**from echo import echofilter **。如果没有在当前的包中发现要导入的模块，**import **语句会依据指定名寻找一个顶级模块。**

When packages are structured into subpackages (as with the Sound package in the example), there's no shortcut to refer to submodules of sibling packages - the full name of the subpackage must be used. For example, if the module Sound.Filters.vocoder needs to use the echo module in the Sound.Effects package, it can use from Sound.Effects import echo.

如果包中使用了子包结构（就像示例中的 Sound **包），不存在什么从邻近的包中引用子模块的便捷方法－－必须使用子包的全名。例如，如果**Sound.Filters.vocoder **包需要使用**Sound.Effects **包中的**echosa **模块，它可以使用**from Sound.Effects import echo **。**

6.4.3 多重路径中的包Packages in Multiple Directories

Packages support one more special attribute, __path__. This is initialized to be a list containing the name of the directory holding the package's '__init__.py' before the code in that file is executed. This variable can be modified; doing so affects future searches for modules and subpackages contained in the package.

包支持一个另为特殊的变量，__path__ **。在包的**'__init__.py' **文件代码执行之前，该变量初始化一个目录名列表。该变量可以修改，它作用于包中的子包和模块的搜索功能。**

While this feature is not often needed, it can be used to extend the set of modules found in a package.

这个功能可以用于扩展包中的模块集，不过它不常用。

第七章

输入和输出Input and Output

There are several ways to present the output of a program; data can be printed in a human-readable form, or written to a file for future use. This chapter will discuss some of the possibilities.

有几种方法可以表现程序的输出结果；数据可以用可读的结构打印，也可以写入文件供以后使用。本章将会讨论几种可行的做法。

7.1 设计输出格式Fancier Output Formatting

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.)

我们有两种大相径庭的输出值方法：*表达式语句*和`print` 语句。（第三种访求是使用文件对象的`write()` 方法，标准文件输出可以参考`sys.stdout` 。详细内容参见库参考手册。）

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 lay-out you can imagine. The standard module `string` contains some useful operations for padding strings to a given column width; these will be discussed shortly. The second way is to use the % operator with a string as the left argument. The % operator interprets the left argument much like a `sprintf()`-style format string to be applied to the right argument, and returns the string resulting from this formatting operation.

可能你经常想要对输出格式做一些比简单的打印空格分隔符更为复杂的控制。有两种方法可以格式化输出。第一种是由你来控制整个字符串，使用字符切片和联接操作就可以创建出任何你想要的输出形式。标准模块`string` 包括了一些操作，将字符串填充入给定列时，这些操作很有用。随后我们会讨论这部分内容。第二种方法是使用% 操作符，以某个字符串做为其左参数。% 操作符将左参数解释为类似于`sprintf()` 风格的格式字符串，并作用于右参数，从该操作中返回格式化的字符串。

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. Reverse quotes (``) are equivalent to `repr()`, but their use is discouraged.

当然，还有一个问题，如何将（不同的）值转化为字符串？很幸运，Python**总是把任意值传入**`repr()` **或**`str()` **函数，转为字符串。相对而言引号(``) 等价于**`repr()`**，不过不提倡这样用。**

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 not 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.

函数`str()` 用于将值转化为适于人阅读的形式，而`repr()` 转化为供解释器读取的形式（如果没有等价的语法，则会发生`SyntaxError` 异常）某对象没有适于人阅读的解释形式的话，`str()` 会返回

与`repr()`等同的值。很多类型，诸如数值或链表、字典这样的结构，针对各函数都有着统一的解读方式。字符串和浮点数，有着独特的解读方式。

Some examples:

以下是一些示例：

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(0.1)
'0.1'
>>> repr(0.1)
'0.10000000000000001'
>>> 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'))"
>>> # reverse quotes are convenient in interactive sessions:
... `x, y, ('spam', 'eggs')`
"(32.5, 40000, ('spam', 'eggs'))"
```

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

以下两种方法可以输出平方和立方表：

```
>>> 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 '%2d %3d %4d' % (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 one space between each column was added by the way `print` works: it always adds spaces between its arguments.)

（需要注意的是使用`print` 方法时每两列之间有一个空格：它总是在参数之间加一个空格。）

This example demonstrates the `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 `ljust()` and `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]`'.)

以上是一个`rjust()` 函数的演示，这个函数把字符串输出到一列，并通过向左侧填充空格来使其右对齐。类似的函数还有`ljust()` 和`center()`。这些函数只是输出新的字符串，并不改变什么。如果输出的字符串太长，它们也不会截断它，而是原样输出，这会使你的输出格式变得混乱，不过总强过另一种选择（截断字符串），因为那样会产生错误的输出值。（如果你确实需要截断它，可以使用切片操作，例如："'`x.ljust( n)[:n]`'。）

There is another method, `zfill()`, which pads a numeric string on the left with zeros. It understands about plus and minus signs:

还有一个函数，`zfill()` 它用于向数值的字符串表达左侧填充0。该函数可以正确理解正负号：

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

Using the `%` operator looks like this:

可以如下这样使用% 操作符：

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

If there is more than one format in the string, you need to pass a tuple as right operand, as in this example:

如果有超过一个的字符串要格式化为一体，就需要将它们传入一个元组做为右值，如下所示：

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

Most formats work exactly as in C and require that you pass the proper type; however, if you don't you get an exception, not a core dump. The `%s` format is more relaxed: if the corresponding argument is not a string object, it is converted to string using the `str()` built-in function. Using `*` to pass the width or precision in as a separate (integer) argument is supported. The C formats `%n` and `%p` are not supported.

大多数类C 的格式化操作都需要你传入适当的类型，不过如果你没有定义异常，也不会有什么从内核中主动的弹出来。（however, if you don't you get an exception, not a core dump**）使用%s 格式会更轻松些：如果对应的参数不是字符串，它会通过内置的`str()` 函数转化为字符串。**Python**支持用*** 作为一个隔离（整型的）参数来传递宽度或精度。**Python**不支持**C**的`%n` 和`%p` 操作符。**

If you have a really long format string that you don't want to split up, it would be nice if you could reference the variables to be formatted by name instead of by position. This can be done by using form `%(name)format`, as shown here:

如果可以逐点引用要格式化的变量名，就可以产生符合真实长度的格式化字符串，不会产生间隔。这一效果可以通过使用form `%(name)format` **结构来实现：**

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print 'Jack: %(Jack)d; Sjoerd: %(Sjoerd)d; Dcab: %(Dcab)d' % table
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This is particularly useful in combination with the new built-in `vars()` function, which returns a dictionary containing all local variables.

这个技巧在与新的内置函数`vars()` 组合使用时非常有用，该函数返回一个包含所有局部变量的字典。

7.2 读写文件Reading and Writing Files

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

`open()` **返回一个文件，通常的用法需要两个参数：**'`open(`*filename*`,` *mode*`)`'。

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

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. *mode* can be `'r'` when the file will only be read, 'w' for only writing (an existing file with the same name will be erased), and 'a' opens the file for appending; any data written to the file is automatically added to the end. 'r+' opens the file for both reading and writing. The *mode* argument is optional; 'r' will be assumed if it's omitted.

第一个参数是一个标识文件名的字符串。第二个参数是由有限的字母组成的字符串，描述了文件将会被如何使用。可选的*模式* 有：'r' ，此选项使文件只读；'w' ，此选项使文件只写（对于同名文件，该操作使原有文件被覆盖）；'a' ，此选项以追加方式打开文件；'r+' ，此选项以读写方式打开文件；如果没有指定，默认为'r' 模式。

On Windows and the Macintosh, 'b' appended to the mode opens the file in binary mode, so there are also modes like 'rb', 'wb', and 'r+b'. 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 JPEGs or '.EXE' files. Be very careful to use binary mode when reading and writing such files. (Note that the precise semantics of text mode on the Macintosh depends on the underlying C library being used.)

在Windows **和**Macintosh**平台上，'b' 模式以二进制方式打开文件，所以可能会有类似于'rb' ，'wb' ，'r+b' 等等模式组合。**Windows**平台上文本文件与二进制文件是有区别的，读写文本文件时，行尾会自动添加行结束符。这种后台操作方式对**ASCII **文本文件没有什么问题，但是操作**JPEG **或**'.EXE' **这样的二进制文件时就会产生破坏。在操作这些文件时一定要记得以二进制模式打开。（需要注意的是**Mactiontosh **平台上的文本模式依赖于其使用的底层C库）。**

7.2.1 文件对象（file object）的方法Methods of File Objects

The rest of the examples in this section will assume that a file object called f has already been created.

本节中的示例都默认文件对象f 已经创建。

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 ("").

要读取文件内容，需要调用f.read(*size*)，该方法读取若干数量的数据并以字符串形式返回其内容，字符串长度为数值*size* 所指定的大小。如果没有指定*size* 或者指定为负数，就会读取并返回整个文件。当文件大小为当前机器内存两倍时，就会产生问题。正常情况下，会尽可能按比较大的*size* 读取和返回数据。如果到了文件末尾，f.read() 会返回一个空字符串（""）。

```
>>> 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() 从文件中读取单独一行，字符串结尾会自动加上一个换行符，只有当文件最后一行没有以换行符结尾时，这一操作才会被忽略。这样返回值就不会有什么混淆不清，如果如果f.readline() 返回一个空字符串，那就表示到达了文件末尾，如果是一个空行，就会描述为'\n´ ，一个只包含换行符的字符串。

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

`f.readlines()` returns a list containing all the lines of data in the file. If given an optional parameter *sizehint*, it reads that many bytes from the file and enough more to complete a line, and returns the lines from that. This is often used to allow efficient reading of a large file by lines, but without having to load the entire file in memory. Only complete lines will be returned.

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

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

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

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

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

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

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

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.

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

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

File objects have some additional methods, such as `isatty()` and `truncate()` which are less frequently used; consult the Library Reference for a complete guide to file objects.

文件对象还有一些不太常用的附加方法，比如`isatty()` 和`truncate()` 在库参考手册中有文件对象的完整指南。

7.2.2 `pickle` 模块`pickle` Module

Strings can easily be written to and read from a file. Numbers take a bit more effort, since the `read()` method only returns strings, which will have to be passed to a function like `int()`, which takes a string like `'123'` and returns its numeric value 123. However, when you want to save more complex data types like lists, dictionaries, or class instances, things get a lot more complicated.

我们可以很容易的读写文件中的字符串。数值就要多费点儿周折，因为`read()` 方法只会返回字符串，应该将其传入`int()` 方法中，就可以将`'123'` 这样的字符转为对应的数值123。不过，当你需要保存更为复杂的数据类型，例如链表、字典，类的实例，事情就会变得更复杂了。

Rather than have users be constantly writing and debugging code to save complicated data types, Python provides a standard module called `pickle`. This is an amazing module that can take almost any Python object (even some forms of Python code!), and convert it to a string representation; this process is called *pickling*. Reconstructing the object from the string representation is called *unpickling*. Between pickling and unpickling, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.

好在用户不必要非得自己编写和调试保存复杂数据类型的代码。Python 提供了一个名为`pickle` 的标准模块。这是一个令人赞叹的模块，几乎可以把任何Python 对象（甚至是一些Python 代码块（form ）！）表达为为字符串，这一过程称之为*封装* （*pickling* ）。从字符串表达出重新构造对象称之为*拆封*（*unpickling* ）。封装状态中的对象可以存储在文件或对象中，也可以通过网络在远程的机器之间传输。

If you have an object `x`, and a file object `f` that's been opened for writing, the simplest way to pickle the object takes only one line of code:

如果你有一个对象`x` ，一个以写模式打开的文件对象`f` ，封装对像的最简单的方法只需要一行代码：

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

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

如果`f` 是一个以读模式打开的文件对象，就可以重装拆封这个对象：

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

(There are other variants of this, used when pickling many objects or when you don't want to write the pickled data to a file; consult the complete documentation for `pickle` in the *Python Library Reference*.)

（如果不想把封装的数据写入文件，这里还有一些其它的变化可用。完整的`pickle` 文档请见*Python 库参考手册*）。

`pickle` is the standard way to make Python objects which can be stored and reused by other programs or by a future invocation of the same program; the technical term for this is a *persistent* object. Because `pickle` is so widely used, many authors who write Python extensions take care to ensure that new data types such as matrices can be properly pickled and unpickled.

`pickle` 是存储Python 对象以供其它程序或其本身以后调用的标准方法。提供这一组技术的是一个持久化对象（*persistent* object ）。因为`pickle` 的用途很广泛，很多Python 扩展的作者都非常注意类似矩阵这样的新数据类型是否适合封装和拆封。

第八章

错误和异常Errors and Exceptions

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

至今为止还没有进一步的谈论过错误信息，不过在你已经试验过的那些例子中，可能已经遇到过一些。Python **中（至少）有两种错误：语法错误和异常（***syntax errors* and *exceptions* **）。**

8.1 语法错误Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

语法错误，也称作解析错误，可能是学习 Python **的过程中最容易犯的：**

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

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.

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

8.2 异常Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

即使是在语法上完全正确的语句，尝试执行它的时候，也有可能会发生错误。在程序运行中检测出的错误称之为异常，它通常不会导致致命的问题，你很快就会学到如何在 Python **程序中控制它们。大多数异常不会由程序处理，而是显示一个错误信息：**

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

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are `ZeroDivisionError`, `NameError` and `TypeError`. The string printed as the exception type is the name of the built-in name for the exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

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

The rest of the line is a detail whose interpretation depends on the exception type; its meaning is dependent on the exception type.

这一行后一部分是关于该异常类型的详细说明，这意味着它的内容依赖于异常类型。

The preceding part of the error message shows the context where the exception happened, in the form of a stack backtrace. In general it contains a stack backtrace listing source lines; however, it will not display lines read from standard input.

错误信息的前半部分以堆栈的形式列出异常发生的位置。通常在堆栈中列出了源代码行，然而，来自标准输入的源码不会显示出来。

The *Python Library Reference* lists the built-in exceptions and their meanings.

Python ***库参考手册*** **列出了内置异常和它们的含义。**

8.3 处理异常Handling Exceptions

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using `Control-C` or whatever the operating system supports); note that a user-generated interruption is signalled by raising the `KeyboardInterrupt` exception.

通过编程可以处理指定的异常。以下的例子重复要求用户输入一个值，直到用户输入的是一个合法的整数为止。不过这个程序允许用户中断程序（使用`Control-C` 或者其它操作系统支持的方法）。需要注意的是用户发出的中断会引发一个`KeyboardInterrupt` 异常。

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

The `try` statement works as follows.

`try` 语句按如下方式工作：

- First, the *try clause* (the statement(s) between the `try` and `except` keywords) is executed.

  首先，执行*try* 子句（在`try` 和`except` 关键字之间的部分）。

- If no exception occurs, the *except clause* is skipped and execution of the `try` statement is finished.

  如果没有异常发生，*except* 子句 在`try` 语句执行完毕后就被忽略了。

- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the `except` keyword, the rest of the try clause is skipped, the except clause is executed, and then execution continues after the `try` statement.

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

- If an exception occurs which does not match the exception named in the except clause, it is passed on to outer `try` statements; if no handler is found, it is an *unhandled exception* and execution stops with a message as shown above.

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

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 list, for example:

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

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

The last except clause may omit the exception name(s), to serve as a wildcard. Use this with extreme caution, since it is easy to mask a real programming error in this way! It can also be used to print an error message and then re-raise the exception (allowing a caller to handle the exception as well):

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

```
import sys

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

The `try` ... `except` statement has an optional *else clause*, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception. For example:

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

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

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.

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

When an exception occurs, it may have an associated value, also known as the exception's *argument*. The presence and type of the argument depend on the exception type.

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

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

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

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

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

Exception handlers don't just handle exceptions if they occur immediately in the try clause, but also if they occur inside functions that are called (even indirectly) in the try clause. For example:

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

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

8.4 抛出异常Raising Exceptions

The `raise` statement allows the programmer to force a specified exception to occur. For example:

在发生了特定的异常时，程序员可以用`raise` 语句强制抛出异常。例如：

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

The first argument to `raise` names the exception to be raised. The optional second argument specifies the exception's argument.

第一个参数指定了所抛出异常的名称，第二个指定了异常的参数。

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the `raise` statement allows you to re-raise the exception:

如果你决定抛出一个异常而不处理它，`raise` 语句可以让你很简单的重新抛出该异常。

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

8.5 用户自定义异常User-defined Exceptions

Programs may name their own exceptions by creating a new exception class. Exceptions should typically be derived from the `Exception` class, either directly or indirectly. For example:

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

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

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception. When creating a module which can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions:

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

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition is not allowed
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

Most exceptions are defined with names that end in “Error,” similar to the naming of the standard exceptions.

与标准异常相似，大多数异常的命名都以“ Error ”结尾。

Many standard modules define their own exceptions to report errors that may occur in functions they define. More information on classes is presented in chapter 9, “Classes.”

很多标准模块中都定义了自己的异常，用以报告在他们所定义的函数中可能发生的错误。关于类的进一步信息请参见第9章9，“ 类 ”。

8.6 定义清理行为Defining Clean-up Actions

The `try` statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

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

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

A *finally clause* is executed whether or not an exception has occurred in the try clause. When an exception has occurred, it is re-raised after the finally clause is executed. The finally clause is also executed “on the way out” when the `try` statement is left via a `break` or `return` statement.

不管try子句中有没有发生异常，finally 子句都一定会被执行。如果发生异常，在finally 子句执行完后它会被重新抛出。`try` 子句经由`break` 或`return` 退出也一样会执行finally 子句。

The code in the finally clause is useful for releasing external resources (such as files or network connections), regardless of whether or not the use of the resource was successful.

在finally 子句中的代码用于释放外部资源（例如文件或网络连接），不管这些资源是否已经成功利用。

A `try` statement must either have one or more except clauses or one finally clause, but not both.

在`try` 语句中可以使用若干个except 子句或一个finally 子句，但两者不能共存。

第九章

类Classes

Python's class mechanism adds classes to the language with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. As is true for modules, classes in Python do not put an absolute barrier between definition and user, but rather rely on the politeness of the user not to "break into the definition." The most important features of classes are retained with full power, however: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, a method can call the method of a base class with the same name. Objects can contain an arbitrary amount of private data.

Python在尽可能不增加新的语法和语义的情况下加入了类机制。这种机制是C++**和**Python's Modula-3**的混合。**Python**中的类没有在用户和定义之间建立一个绝对的屏障，而是依赖于用户自觉的不去" 破坏定义 "。然而，类机制最重要的功能都完整的保留下来。类继承机制允许多继承，派生类可以覆盖（**override**）基类中的任何方法，方法中可以调用基类中的同名方法。对象可以包含任意数量的私有成员。**

In C++ terminology, all class members (including the data members) are *public*, and all member functions are *virtual*. There are no special constructors or destructors. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects, albeit in the wider sense of the word: in Python, all data types are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++ but unlike in Modula-3, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

用C++ **术语来讲，所有的类成员（包括数据成员）都是公有（***public* **）的，所有的成员函数都是虚拟（***virtual* **）的。用**Modula-3**的术语来讲，在成员方法中没有什么简便的方式（**shorthands **）可以引用对象的成员：方法函数在定义时需要以引用的对象做为第一个参数，调用时则会隐式引用对象。这样就形成了语义上的引入和重命名。（**This provides semantics for importing and renaming. **）但是，像**C++ **而非**Modula-3 **中那样，大多数带有特殊语法的内置操作符（算法运算符、下标等）都可以针对类的需要重新定义。**

9.1 有关术语的话题A Word About Terminology

Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. (I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)

由于没有什么关于类的通用术语，我从 Smalltalk **和**C++ **中借用一些（我更希望用**Modula-3 **的，因为它的面向对象机制比**C++**更接近**Python**，不过我想没多少读者听说过它）。**

I also have to warn you that there's a terminological pitfall for object-oriented readers: the word "object" in Python does not necessarily mean a class instance. Like C++ and Modula-3, and unlike Smalltalk, not all types in Python are classes: the basic built-in types like integers and lists are not, and even somewhat more exotic types like files aren't. However, *all* Python types share a little bit of common semantics that is best described by using the word object.

我要提醒读者，这里有一个面向对象方面的术语陷阱，在Python 中“ 对象 ”这个词不一定指类实例。Python中并非所有的类型都是类：例如整型、链表这些内置数据类型就不是，甚至某些像文件这样的外部类型也不是，这一点类似于C++ 和Modula-3，而不像Smalltalk。然而，所有的Python类型在语义上都有一点相同之处：描述它们的最贴切词语是“ 对象 ”。

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has an (intended!) effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most types representing entities outside the program (files, windows, etc.). This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

对象是被特化的，多个名字（在多个作用域中）可以绑定同一个对象。这相当于其它语言中的别名。通常对Python 的第一印象中会忽略这一点，使用那些不可变的基本类型（数值、字符串、元组）时也可以很放心的忽视它。然而，在Python 代码调用字典、链表之类可变对象，以及大多数涉及程序外部实体（文件、窗体等等）的类型时，这一语义就会有影响。这通用有助于优化程序，因为别名的行为在某些方面类似于指针。例如，很容易传递一个对象，因为在行为上只是传递了一个指针。如果函数修改了一个通过参数传递的对象，调用者可以接收到变化－－在Pascal 中这需要两个不同的参数传递机制。

9.2 Python 作用域和命名空间Python Scopes and Name Spaces

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

在介绍类之前，我首先介绍一些有关Python 作用域的规则：类的定义非常巧妙的运用了命名空间，要完全理解接下来的知识，需要先理解作用域和命名空间的工作原理。另外，这一切的知识对于任何高级Python 程序员都非常有用。

Let's begin with some definitions.

我们从一些定义开始。

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (functions such as `abs()`, and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function "maximize" without confusion — users of the modules must prefix it with the module name.

***命名空间*是从命名到对象的映射。当前*命名空间*主要是通过Python 字典实现的，不过通常不关心具体的实现方式（除非出于性能考虑），以后也有可能会改变其实现方式。以下有一些命名空间的例子：内置命名（像`abs()` 这样的函数，以及内置异常名）集，模块中的全局命名，函数调用中的局部命名。某种意义上讲对象的属性集也是一个*命名空间*。关于*命名空间*需要了解的一件很重要的事就是不同*命名空间*中的命名没有任何联系，例如两个不同的模块可能都会定义一个名为“ maximize ”的函数而不会发生混淆－－用户必须以模块名为前缀来引用它们。**

By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace! [1]

顺便提一句，我称Python 中任何一个“ . ”之后的命名为*属性*－－例如，表达式`z.real` 中的`real`

[1]Except for one thing. Module objects have a secret read-only attribute called `__dict__` which returns the dictionary used to implement the module's namespace; the name `__dict__` is an attribute but not a global name. Obviously, using this violates the abstraction of namespace implementation, and should be restricted to things like post-mortem debuggers.

是对象z 的一个属性。严格来讲，从模块中引用命名是引用属性：表达式modname.funcname 中，modname 是一个模块对象，funcname 是它的一个属性。因此，模块的属性和模块中的全局命名有直接的映射关系：它们共享同一命名空间！[2]

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write '`modname.the_answer = 42`'. Writable attributes may also be deleted with the `del` statement. For example, '`del modname.the_answer`' will remove the attribute `the_answer` from the object named by `modname`.

属性可以是只读过或写的。后一种情况下，可以对属性赋值。你可以这样作：'`modname.the_answer = 42`'**。可写的属性也可以用**`del` **语句删除。例如：**'`del modname.the_answer`' **会从**`modname` **对象中删除**`the_answer` **属性。**

Name spaces 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__`.)

不同的命名空间在不同的时刻创建，有不同的生存期。包含内置命名的命名空间在 Python **解释器启动时创建，会一直保留，不被删除。模块的全局命名空间在模块定义被读入时创建，通常，模块命名空间也会一直保存到解释器退出。由解释器在最高层调用执行的语句，不管它是从脚本文件中读入还是来自交互式输入，都是**__main__ **模块的一部分，所以它们也拥有自己的命名空间。（内置命名也同样被包含在一个模块中，它被称作**__builtin__ **。）**

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

当函数被调用时创建一个局部命名空间，函数反正返回过抛出一个未在函数内处理的异常时删除。（实际上，说是遗忘更为贴切）。当然，每一个递归调用拥有自己的命名空间。

A *scope* is a textual region of a Python program where a namespace is directly accessible. "Directly accessible" here means that an unqualified reference to a name attempts to find the name in the namespace.

***作用域*是指**Python **程序可以直接访问到的命名空间。" 直接访问 " 在这里意味着访问命名空间中的命名时无需加入附加的修饰符。**

Although scopes are determined statically, they are used dynamically. At any time during execution, there are at least three nested scopes whose namespaces are directly accessible: the innermost scope, which is searched first, contains the local names; the namespaces of any enclosing functions, which are searched starting with the nearest enclosing scope; the middle scope, searched next, contains the current module's global names; and the outermost scope (searched last) is the namespace containing built-in names.

尽管作用域是静态定义，在使用时他们都是动态的。每次执行时，至少有三个命名空间可以直接访问的作用域嵌套在一起：包含局部命名的使用域在最里面，首先被搜索；其次搜索的是中层的作用域，这里包含了同级的函数；最后搜索最外面的作用域，它包含内置命名。

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.

如果一个命名声明为全局的，那么所有的赋值和引用都直接针对包含模全局命名的中级作用域。另外，从外部访问到的所有内层作用域的变量都是只读的。

Usually, the local scope references the local names of the (textually) current function. Outside of functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

从文本意义上讲，局部作用域引用当前函数的命名。在函数之外，局部作用域与全局使用域引用同一命名空间：模块命名空间。类定义也是局部作用域中的另一个命名空间。

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the

[2]**有一个例外。模块对象有一个隐秘的只读对象，名为**__dict__ **，它返回用于实现模块命名空间的字典，命名**__dict__ **是一个属性而非全局命名。显然，使用它违反了命名空间实现的抽象原则，应该被严格限制于调试中。**

actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at "compile" time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

作用域决定于源程序的文本：一个定义于某模块中的函数的全局作用域是该模块的命名空间，而不是该函数的别名被定义或调用的位置，了解这一点非常重要。另一方面，命名的实际搜索过程是动态的，在运行时确定的——然而，Python **语言也在不断发展，以后有可能会成为静态的"编译"时确定，所以不要依赖于动态解析！（事实上，局部变量已经是静态确定了。）**

A special quirk of Python is that assignments 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.)

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

9.3 初识类A First Look at Classes

Classes introduce a little bit of new syntax, three new object types, and some new semantics.

类引入了一点新的语法，三种新的对象类型，以及一些新的语义。

9.3.1 类定义语法Class Definition Syntax

The simplest form of class definition looks like this:

最简单的类定义形式如下：

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

Class definitions, like function definitions (`def` statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an `if` statement, or inside a function.)

类的定义就像函数定义（`def` **语句），要先执行才能生效。（你当然可以把它放进**`if` **语句的某一分支，或者一个函数的内部。）**

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful — we'll come back to this later. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods — again, this is explained later.

习惯上，类定义语句的内容通常是函数定义，不过其它语句也可以，有时会很有用——后面我们再回过头来讨论。类中的函数定义通常包括了一个特殊形式的参数列表，用于方法调用约定——同样我们在后面讨论这些。

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

定义一个类的时候，会创建一个新的命名空间，将其作为局部作用域使用——因此，所以对局部变量的赋值都引入新的命名空间。特别是函数定义将新函数的命名绑定于此。

When a class definition is left normally (via the end), a *class object* is created. This is basically a wrapper around the contents of the namespace created by the class definition; we'll learn more about class objects in the next section. The original local scope (the one in effect just before the class definitions was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (`ClassName` in the example).

类定义完成时（正常退出），就创建了一个类对象。基本上它是对类定义创建的命名空间进行了一个包装；我们在下一节进一步学习类对象的知识。原始的局部作用域（类定义引入之前生效的那个）得到恢复，类对象在这里绑定到类定义头部的类名（例子中是`ClassName`）。

9.3.2 类对象Class Objects

Class objects support two kinds of operations: attribute references and instantiation.

类对象支持两种操作：属性引用和实例化。

Attribute references use the standard syntax used for all attribute references in Python: `obj.name`. Valid attribute names are all the names that were in the class's namespace when the class object was created. So, if the class definition looked like this:

属性引用使用和Python中所有的属性引用一样的标准语法：`obj.name`。类对象创建后，类命名空间中所有的命名都是有效属性名。所以如果类定义是这样：

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

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a method object, respectively. Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment. `__doc__` is also a valid attribute, returning the docstring belonging to the class: `"A simple example class"`.

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

Class *instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

类的实例化使用函数符号。只要将类对象看作是一个返回新的类实例的无参数函数即可。例如（假设沿用前面的类）：

```
x = MyClass()
```

creates a new *instance* of the class and assigns this object to the local variable `x`.

以上创建了一个新的类*实例*并将该对象赋给局部变量`x`。

The instantiation operation ("calling" a class object) creates an empty object. Many classes like to create objects in a known initial state. Therefore a class may define a special method named `__init__()`, like this:

这个实例化操作（"调用"一个类对象）来创建一个空的对象。很多类都倾向于将对象创建为有初始状态的。因此类可能会定义一个名为`__init__()`的特殊方法，像下面这样：

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

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly-created class instance. So in this example, a new, initialized instance can be obtained by:

类定义了`__init__()` 方法的话，类的实例化操作会自动为新创建的类实例调用`__init__()` 方法。所以在下例中，可以这样创建一个新的实例：

```
x = MyClass()
```

Of course, the `__init__()` method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to `__init__()`. For example,

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

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

9.3.3 实例对象Instance Objects

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.

现在我们可以用实例对象作什么？实例对象唯一可用的操作就是属性引用。有两种有效的属性名。

The first I'll call *data attributes*. These correspond to "instance variables" in Smalltalk, and to "data members" in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if `x` is the instance of `MyClass` created above, the following piece of code will print the value `16`, without leaving a trace:

第一种称作数据属性。这相当于Smalltalk 中的 " 实例变量 " 或C++中的 " 数据成员 " 。和局部变量一样，数据属性不需要声明，第一次使用时它们就会生成。例如，如果`x` 是前面创建的`MyClass` 实例，下面这段代码会打印出`16` 而不会有任何多余的残留：

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

The second kind of attribute references understood by instance objects are *methods*. A method is a function that "belongs to" an object. (In Python, the term method is not unique to class instances: other object types can have methods as well. For example, list objects have methods called append, insert, remove, sort, and so on. However, below, we'll use the term method exclusively to mean methods of class instance objects, unless explicitly stated otherwise.)

第二种为实例对象所接受的引用属性是方法。方法是属于一个对象的函数。（在Python 中，方法不止是类实例所独有：其它类型的对象也可有方法。例如，链表对象有append，insert，remove，sort 等等方法。然而，在这里，除非特别说明，我们提到的方法特指类方法）

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are (user-defined) function objects define corresponding methods of its instances. So in our example, `x.f` is a valid method reference, since `MyClass.f` is a function, but `x.i` is not, since `MyClass.i` is not. But `x.f` is not the same thing as `MyClass.f` — it is a *method object*, not a function object.

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

9.3.4 **方法对象**Method Objects

Usually, a method is called immediately:

通常方法是直接调用的：

```
x.f()
```

In our example, this will return the string `'hello world'`. However, it is not necessary to call a method right away: `x.f` is a method object, and can be stored away and called at a later time. For example:

在我们的例子中，这会返回字符串`'hello world'`。然而，也不是一定要直接调用方法。`x.f` 是一个方法对象，它可以存储起来以后调用。例如：

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

will continue to print '`hello world`' until the end of time.

会不断的打印'`hello world`'。

What exactly happens when a method is called? You may have noticed that `x.f()` was called without an argument above, even though the function definition for `f` specified an argument. What happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any — even if the argument isn't actually used...

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

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.

实际上，你可能已经猜到了答案：方法的特别之处在于实例对象作为函数的第一个参数传给了函数。在我们的例子中，调用`x.f()` 相当于`MyClass.f(x)` 。通常，以*n* 个参数的列表去调用一个方法就相当于将方法的对象插入到参数列表的最前面后，以这个列表去调用相应的函数。

If you still don't understand how methods work, a look at the implementation can perhaps clarify matters. When an instance attribute is referenced that isn't a data attribute, its class is searched. If the name denotes a valid class attribute that is a function object, a method object is created by packing (pointers to) the instance object and the function object just found together in an abstract object: this is the method object. When the method object is called with an argument list, it is unpacked again, a new argument list is constructed from the instance object and the original argument list, and the function object is called with this new argument list.

如果你还是不理解方法的工作原理，了解一下它的实现也许有帮助。引用非数据属性的实例属性时，会搜索它的类。如果这个命名确认为一个有效的函数对象类属性，就会将实例对象和函数对象封装进一个抽象对象：这就是方法对象。以一个参数列表调用方法对象时，它被重新拆封，用实例对象和原始的参数列表构造一个新的参数列表，然后函数对象调用这个新的参数列表。

9.4 一些说明Random Remarks

〔有些内容可能需要明确一下……〕

Data attributes override method attributes with the same name; to avoid accidental name conflicts, which may cause hard-to-find bugs in large programs, it is wise to use some kind of convention that minimizes the chance of conflicts. Possible conventions include capitalizing method names, prefixing data attribute names with a small unique string (perhaps just an underscore), or using verbs for methods and nouns for data attributes.

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

Data attributes may be referenced by methods as well as by ordinary users ("clients") of an object. In other words, classes are not usable to implement pure abstract data types. In fact, nothing in Python makes it possible to enforce data hiding — it is all based upon convention. (On the other hand, the Python implementation, written in C, can completely hide implementation details and control access to an object if necessary; this can be used by extensions to Python written in C.)

数据属性可以由方法引用，也可以由普通用户（客户）调用。换句话说，类不能实现纯的数据类型。事实上 Python **中没有什么办法可以强制隐藏数据－－一切都基本约定的惯例。（另一方法讲，**Python **的实现是用**C **写成的，如果有必要，可以用**C **来编写**Python **扩展，完全隐藏实现的细节，控制对象的访问。）**

Clients should use data attributes with care — clients may mess up invariants maintained by the methods by stamping on their data attributes. Note that clients may add data attributes of their own to an instance object without affecting the validity of the methods, as long as name conflicts are avoided — again, a naming convention can save a lot of headaches here.

客户应该小心使用数据属性－－客户可能会因为随意修改数据属性而破坏了本来由方法维护的数据一致性。需要注意的是，客户只要注意避免命名冲突，就可以随意向实例中添加数据属性而不会影响方法的有效性－－再次强调，命名约定可以省去很多麻烦。

There is no shorthand for referencing data attributes (or other methods!) from within methods. I find that this actually increases the readability of methods: there is no chance of confusing local variables and instance variables when glancing through a method.

从方法内部引用数据属性（以及其它方法！）没有什么快捷的方式。我认为这事实上增加了方法的可读性：即使粗略的浏览一个方法，也不会有混淆局部变量和实例变量的机会。

Conventionally, the first argument of methods is often called `self`. This is nothing more than a convention: the name `self` has absolutely no special meaning to Python. (Note, however, that by not following the convention your code may be less readable by other Python programmers, and it is also conceivable that a *class browser* program be written which relies upon such a convention.)

习惯上，方法的第一个参数命名为`self` **。这仅仅是一个约定：对**Python **而言，**`self` **绝对没有任何特殊含义。（然而要注意的是，如果不遵守这个约定，别的**Python **程序员阅读你的代码时会有不便，而且有些类浏览程序也是遵循此约定开发的。）**

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also ok. For example:

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

```
# 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
```

Now `f`, `g` and `h` are all attributes of class `C` that refer to function objects, and consequently they are all methods of instances of `C` — `h` being exactly equivalent to `g`. Note that this practice usually only serves to confuse the reader of a program.

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

Methods may call other methods by using method attributes of the `self` argument:

通过`self` 参数的方法属性，方法可以调用其它的方法：

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

Methods may reference global names in the same way as ordinary functions. The global scope associated with a method is the module containing the class definition. (The class itself is never used as a global scope!) While one rarely encounters a good reason for using global data in a method, there are many legitimate uses of the global scope: for one thing, functions and modules imported into the global scope can be used by methods, as well as functions and classes defined in it. Usually, the class containing the method is itself defined in this global scope, and in the next section we'll find some good reasons why a method would want to reference its own class!

方法可以像引用普通的函数那样引用全局命名。与方法关联的全局作用域是包含类定义的模块。（类本身永远不会做为全局作用域使用！）尽管很少有好的理由在方法中使用全局数据，全局作用域确有很多合法的用途：其一是方法可以调用导入全局作用域的函数和方法，也可以调用定义在其中的类和函数。通常，包含此方法的类也会定义在这个全局作用域，在下一节我们会了解为何一个方法要引用自己的类！

9.5 继承Inheritance

Of course, a language feature would not be worthy of the name "class" without supporting inheritance. The syntax for a derived class definition looks as follows:

当然，如果一种语言不支持继承就，" 类 " 就没有什么意义。派生类的定义如下所示：

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

The name `BaseClassName` must be defined in a scope containing the derived class definition. Instead of a base class name, an expression is also allowed. This is useful when the base class is defined in another module,

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

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

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, it is searched in the base class. This rule is applied recursively if the base class itself is derived from some other class.

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

There's nothing special about instantiation of derived classes: `DerivedClassName()` creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.

派生类的实例化没有什么特殊之处：`DerivedClassName()`（示列中的派生类）创建一个新的类实例。方法引用按如下规则解析：搜索对应的类属性，必要时沿基类链逐级搜索，如果找到了函数对象这个方法引用就是合法的。

Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class, may in fact end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively `virtual`.)

派生类可能会覆盖其基类的方法。因为方法调用同一个对象中的其它方法时没有特权，基类的方法调用同一个基类的方法时，可能实际上最终调用了派生类中的覆盖方法。（对于C++ 程序员来说，Python中的所有方法本质上都是虚方法。）

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call '`BaseClassName.methodname(self, arguments)`'. This is occasionally useful to clients as well. (Note that this only works if the base class is defined or imported directly in the global scope.)

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

9.5.1 多继承Multiple Inheritance

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

Python**同样有限的支持多继承形式。多继承的类定义形如下例：**

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

The only rule necessary to explain the semantics is the resolution rule used for class attribute references. This 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.

这里唯一需要解释的语义是解析类属性的规则。顺序是深度优先，从左到右。因此，如果在`DerivedClassName`（示例中的派生类）中没有找到某个属性，就会搜索`Base1`，然后（递归的）搜索其基类，如果最终没有找到，就搜索`Base2`，以此类推。

(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`.)

（有些人认为广度优先－－在搜索`Base1`的基类之前搜索`Base2`和`Base3`－－看起来更为自然。然而，如果`Base1`和`Base2`之间发生了命名冲突，你需要了解这个属性是定义于`Base1`还是`Base1`的基类中。而深度优先不区分属性继承自基类还是直接定义。）

It is clear that indiscriminate use of multiple inheritance is a maintenance nightmare, given the reliance in Python on conventions to avoid accidental name conflicts. A well-known problem with multiple inheritance is a class derived from two classes that happen to have a common base class. While it is easy enough to figure out what happens in this case (the instance will have a single copy of "instance variables" or data attributes used by the common base class), it is not clear that these semantics are in any way useful.

显然不加限制的使用多继承会带来维护上的噩梦，因为Python 中只依靠约定来避免命名冲突。多继承一个很有名的问题是派生继承的两个基类都是从同一个基类继承而来。目前还不清楚这在语义上有什么意义，然而很容易想到这会造成什么后果（实例会有一个独立的"实例变量"或数据属性复本作用于公共基类。）

9.6 私有变量Private Variables

There is limited support for class-private identifiers. Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) is now textually replaced with `_classname__spam`, where `classname` is the current class name with leading underscore(s) stripped. This mangling is done without regard of the syntactic position of the identifier, so it can be used to define class-private instance and class variables, methods, as well as globals, and even to store instance variables private to this class on instances of *other* classes. Truncation may occur when the mangled name would be longer than 255 characters. Outside classes, or when the class name consists of only underscores, no mangling occurs.

Python 对类的私有成员提供了有限的支持。任何形如`__spam`（以至少双下划线开头，至多单下划线结尾）随即都被替代为`_classname__spam`，去掉前导下划线的`classname` 即当前的类名。这种混淆不关心标识符的语法位置，所以可用来定义私有类实例和类变量、方法，以及全局变量，甚至于将其它类的实例保存为私有变量。混淆名长度超过255个字符的时候可能会发生截断。在类的外部，或类名只包含下划线时，不会发生截断。

Name mangling is intended to give classes an easy way to define "private" instance variables and methods, without having to worry about instance variables defined by derived classes, or mucking with instance variables by code outside the class. Note that the mangling rules are designed mostly to avoid accidents; it still is possible for a determined soul to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger, and that's one reason why this loophole is not closed. (Buglet: derivation of a class with the same name as the base class makes use of private variables of the base class possible.)

命名混淆意在给出一个在类中定义"私有"实例变量和方法的简单途径，避免派生类的实例变量定义产生问题，或者与外界代码中的变量搞混。要注意的是混淆规则主要目的在于避免意外错误，被认作为私有的变量仍然有可能被访问或修改。在特定的场合它也是有用的，比如调试的时候，这也是一直没有堵上这个漏洞的原因之一（小漏洞：派生类和基类取相同的名字就可以使用基类的私有变量。）

Notice that code passed to `exec`, `eval()` or `evalfile()` 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.

要注意的是传入`exec`，`eval()` 或`evalfile()` 的代码不会将调用它们的类视作当前类，这与`global` 语句的情况类似，`global` 的作用局限于"同一批"进行字节编译的代码。同样的限制也适用于`getattr()`，`setattr()` 和`delattr()`，以及直接引用`__dict__` 的时候。

9.7 补充Odds and Ends

Sometimes it is useful to have a data type similar to the Pascal “record” or C “struct”, bundling together a couple of named data items. An empty class definition will do nicely:

有时类似于Pascal中“ 记录（record）”或C中“ 结构（struct）”的数据类型很有用，它将一组已命名的数据项绑定在一起。一个空的类定义可以很好的实现这它：

```
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
```

A piece of Python code that expects a particular abstract data type can often be passed a class that emulates the methods of that data type instead. For instance, if you have a function that formats some data from a file object, you can define a class with methods `read()` and `readline()` that gets the data from a string buffer instead, and pass it as an argument.

某一段Python 代码需要一个特殊的抽象数据结构的话，通常可以传入一个类，事实上这模仿了该类的方法。例如，如果你有一个用于从文件对象中格式化数据的函数，你可以定义一个带有`read()`和`readline()`方法的类，以此从字符串缓冲读取数据，然后将该类的对象作为参数传入前述的函数。

Instance method objects have attributes, too: `m.im_self` is the object of which the method is an instance, and `m.im_func` is the function object corresponding to the method.

实例方法对象也有属性：`m.im_self`是一个实例方法所属的对象，而`m.im_func`是这个方法对应的函数对象。

9.8 异常也是类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.

用户自定义异常也可以是类。利用这个机制可以创建可扩展的异常体系。

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

以下是两种新的有效（语义上的）异常抛出形式：

```
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:

第一种形式中，`instance`必须是`Class`或其派生类的一个实例。第二种形式是以下形式的简写：

```
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:

发生的异常其类型如果是异常子句中列出的类，或者是其派生类，那么它们就是相符的（反过来说－－发生的异常其类型如果是异常子句中列出的类的基类，它们就不相符）。例如，以下代码会按顺序打印B，C，D：

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

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

要注意的是如果异常子句的顺序颠倒过来（'`execpt B`'在最前），它就会打印B，B，B－－第一个匹配的异常被触发。

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

打印一个异常类的错误信息时，先打印类名，然后是一个空格、一个冒号，然后是用内置函数`str()`将类转换得到的完整字符串。

9.9 迭代器Iterators

By now, you've probably noticed that most container objects can looped over using a `for` statement:

现在你可能注意到大多数容器对象都可以用`for`遍历：

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

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:

这种形式的访问清晰、简洁、方便。这种迭代器的用法在Python中普遍而且统一。在后台，`for`语句在

容器对象中调用`iter()`**。该函数返回一个定义了**`next()`**方法的迭代器对象，它在容器中逐一访问元素。没有后续的元素时，**`next()`**抛出一个**`StopIteration`**异常通知**`for`**语句循环结束。以下是其工作原理的示例：**

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()

Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    it.next()
StopIteration
```

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

了解了迭代器协议的后台机制，就可以很容易的给自己的类添加迭代器行为。定义一个`__iter__()`**方法，使其返回一个带有**`next()`**方法的对象。如果这个类已经定义了**`next()`**，那么**`__iter__()`**只需要返回**self**：**

```
>>> class Reverse:
    "Iterator for looping over a sequence backwards"
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

>>> for char in Reverse('spam'):
    print char

m
a
p
s
```

9.10 发生器Generators

Generators are a simple and powerful tool for creating iterators. They are written like regular functions but use the `yield` statement whenever they want to return data. Each time the `next()` is called, the generator resumes where it left-off (it remembers all the data values and which statement was last executed). An example shows that generators can be trivially easy to create:

发生器是创建迭代器的简单而强大的工具。它们写起来就像是正则函数，需要返回数据的时候使用`yield` **语句。每次**`next()` **被调用时，生成器回复它脱离的位置（它记忆语句最后一次执行的位置和所有的数据值）。以下示例演示了发生器可以很简单的创建出来：**

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

>>> for char in reverse('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.

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

Another key feature is that the local variables and execution state are automatically saved between calls. This made the function easier to write and much more clear than an approach using class variables like `self.index` and `self.data`.

另外一个关键的功能是两次调用之间的局部变量和执行情况都自动保存了下来。这样函数编写起来就比手动调用`self.index` **和**`self.data` **这样的类变量容易的多。**

In addition to automatic method creation and saving program state, when generators terminate, they automatically raise `StopIteration`. In combination, these features make it easy to create iterators with no more effort than writing a regular function.

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

第十章

标准库概览Brief Tour of the Standard Library

10.1 操作系统概览Operating System Interface

The `os` module provides dozens of functions for interacting with the operating system:

`os` 模块提供了不少与操作系统相关联的函数。

```
>>> import os
>>> os.system('time 0:02')
0
>>> os.getcwd()      # Return the current working directory
'C:\\Python24'
>>> os.chdir('/server/accesslogs')
```

Be sure to use the '`import os`' style instead of '`from os import *`'. This will keep `os.open()` from shadowing the builtin `open()` function which operates much differently.

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

The builtin `dir()` and `help()` functions are useful as interactive aids for working with large modules like `os`:

在使用一些像`os` 这样的大型模块时`dir()`和`help()` 函数非常有用。

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

For daily file and directory management tasks, the `shutil` module provides a higher level interface that is easier to use:

针对日常的文件和目录管理任务，`shutil` 模块提供了一个易于使用的高级接口。

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

10.2 文件通配符File Wildcards

The `glob` module provides a function for making file lists from directory wildcard searches:

`glob` 模块提供了一个函数用于从目录通配符搜索中生成文件列表。

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

10.3 命令行参数Command Line Arguments

Common utility scripts often invoke processing command line arguments. These arguments are stored in the `sys` module's *argv* attribute as a list. For instance the following output results from running '`python demo.py one two three`' at the command line:

通用工具脚本经常调用命令行参数。这些命令行参数以链表形式存储于`sys` 模块的*argv* 变量。例如在命令行中执行'`python demo.py one two three`' 后可以得到以下输出结果：

```
>>> 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 `optparse` module.

`getopt` 模块使用UNIX `getopt()` 函处理*sys.argv*。更多的复杂命令行处理由`optparse` 模块提供。

10.4 错误输出重定向和程序终止Error Output Redirection and Program Termination

The `sys` module also has attributes for *stdin*, *stdout*, and *stderr*. The latter is useful for emitting warnings and error messages to make them visible even when *stdout* has been redirected:

`sys` 还有*stdin*，*stdout* 和*stderr* 属性，即使在*stdout* 被重定向时，后者也可以用于显示警告和错误信息。

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

The most direct way to terminate a script is to use '`sys.exit()`'.

大多脚本的定向终止都使用'`sys.exit()`'。

10.5 字符串正则匹配String Pattern Matching

The `re` module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions:

`re` 模块为高级字符串处理提供了正则表达式工具。对于复杂的匹配和处理，正则表达式提供了简洁、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'
```

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

如果只需要简单的功能，应该首先考虑字符串方法，因为它们非常简单，易于阅读和调试。

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

10.6 数学Mathematics

The `math` module gives access to the underlying C library functions for floating point math:

`math` **模块为浮点运算提供了对底层C函数库的访问。**

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

The `random` module provides tools for making random selections:

`random` **提供了生成随机数的工具。**

```
>>> 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Internet Access

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:

有几个模块用于访问互联网以及处理网络通信协议。其中最简单的两个是用于处理从urls 接收的数据的`urllib2`以及用于发送电子邮件的`smtplib`。

```
>>> import urllib2
>>> for line in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
... if 'EST' in line:      # look for Eastern Standard Time
...     print line

<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@tmp.org', 'jceasar@tmp.org',
"""To: jceasar@tmp.org
From: soothsayer@tmp.org

Beware the Ides of March.
""")
>>> server.quit()
```

10.8 日期和时间Dates and Times

The `datetime` module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. The module also supports objects that are time zone aware.

`datetime` 模块为日期和时间处理同时提供了简单和复杂的方法。支持日期和时间算法的同时，实现的重点放在更有效的处理和格式化输出。该模块还支持时区处理。

```
# dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y or %d%b %Y is a %A on the %d day of %B")
'12-02-03 or 02Dec 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Data Compression

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

以下模块直接支持通用的数据打包和压缩格式：

`zlib`**，**`gzip`**，**`bz2`**，**`zipfile`**，以及**`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(t)
-1438085031
```

10.10 性能度量Performance Measurement

Some Python users develop a deep interest in knowing the relative performance between different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

有些用户对了解解决同一问题的不同方法之间的性能差异很感兴趣。Python **提供了一个度量工具，为这些问题提供了直接答案。**

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The `timeit` module quickly demonstrates that the traditional approach is faster:

例如，使用元组封装和拆封来交换元素看起来要比使用传统的方法要诱人的多。`timeit` **证明了传统的方法更快一些。**

```
>>> from timeit import Timer
>>> Timer(’t=a; a=b; b=t’, ’a=1; b=2’).timeit()
0.60864915603680925
>>> Timer(’a,b = b,a’, ’a=1; b=2’).timeit()
0.8625194857439773
```

In contrast to `timeit`’s fine level of granularity, the `profile` and `pstats` modules provide tools for identifying time critical sections in larger blocks of code.

相对于`timeit` **的细粒度，**`profile` **和**`pstats` **模块提供了针对更大代码块的时间度量工具。**

10.11 质量控制Quality Control

One approach for developing high quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.

开发高质量软件的方法之一是为每一个函数开发测试代码，并且在开发过程中经常进行测试。

The `doctest` module provides a tool for scanning a module and validating tests embedded in a program’s docstrings. Test construction is as simple as cutting-and-pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the doctest module to make sure the code remains true to the documentation:

`doctest` **模块提供了一个工具，扫描模块并根据程序中内嵌的文档字符串执行测试。测试构造如同简单的将它的输出结果剪切并粘贴到文档字符串中。通过用户提供的例子，它发展了文档，允许**doctest **模块确认代码的结果是否与文档一致。**

```
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
```

The `unittest` module is not as effortless as the `doctest` module, but it allows a more comprehensive set of tests to be maintained in a separate file:

`unittest` 模块不像`doctest` 模块那么容易使用，不过它可以在一个独立的文件里提供一个更全面的测试集。

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

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

10.12 Batteries Included

Python has a "batteries included" philosophy. This is best seen through the sophisticated and robust capabilities of its larger packages. For example:

Python **包含**"batteries included"**哲学。**Python**可以通过更大的包的来得到应付各种复杂情况的强大能力，从这一点我们可以看出该思想的应用。例如：**

* The `xmlrpclib` and `SimpleXMLRPCServer` modules make implementing remote procedure calls into an almost trivial task. Despite the names, no direct knowledge or handling of XML is needed.

* `xmlrpclib` **和**`SimpleXMLRPCServer` **模块实现了在琐碎的任务中调用远程过程。尽管有这样的名字，其实用户不需要直接处理**XML**，也不需要这方面的知识。**

* 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.

* `email` **包是一个邮件消息管理库，可以处理**MIME **或其它基于**RFC 2822 **的消息文档。不同于实际发送和接收消息的**`smtplib` **和**`poplib` **模块，**email **包有一个用于构建或解析复杂消息结构（包括附件）以及实现互联网编码和头协议的完整工具集。**

* 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.

* `xml.dom` **和**`xml.sax` **包为流行的信息交换格式提供了强大的支持。同样，**`csv` **模块支持在通用数据库格式中直接读写。综合起来，这些模块和包大大简化了**Python **应用程序和其它工具之间的数据交换。**

* Internationalization is supported by a number of modules including `gettext`, `locale`, and the `codecs` package.

* **国际化由**`gettext`**，**`locale`**和**`codecs` **包支持。**

第十一章

What Now?

Reading this tutorial has probably reinforced your interest in using Python — you should be eager to apply Python to solve your real-world problems. Now what should you do?

You should read, or at least page through, the *Python Library Reference*, which gives complete (though terse) reference material about types, functions, and modules that can save you a lot of time when writing Python programs. The standard Python distribution includes a *lot* of code in both C and Python; there are modules to read UNIX mailboxes, retrieve documents via HTTP, generate random numbers, parse command-line options, write CGI programs, compress data, and a lot more; skimming through the Library Reference will give you an idea of what's available.

The major Python Web site is http://www.python.org/; it contains code, documentation, and pointers to Python-related pages around the Web. This Web site is mirrored in various places around the world, such as Europe, Japan, and Australia; a mirror may be faster than the main site, depending on your geographical location. A more informal site is http://starship.python.net/, which contains a bunch of Python-related personal home pages; many people have downloadable software there. Many more user-created Python modules can be found in the Python Package Index (PyPI).

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), or look for it in the 'Misc/' directory of the Python source distribution. Mailing list archives are available at http://www.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.

附录 A

Interactive Input Editing and History Substitution

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.

A.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.

A.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.

A.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 environments. 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.

```
# 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=/max/home/itamar/.pystartup" in bash.
#
# Note that PYTHONSTARTUP does *not* expand "~", so you have to put in the
# full path to your home directory.

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
```

A.4 Commentary

This facility is an enormous step forward compared to earlier versions of the interpreter; however, some wishes are left: It would be nice if the proper indentation were suggested on continuation lines (the parser knows if an indent token is required next). The completion mechanism might use the interpreter's symbol table. A command to check (or even suggest) matching parentheses, quotes, etc., would also be useful.

附录 B

Floating Point Arithmetic: Issues and Limitations

Floating-point numbers are represented in computer hardware as base 2 (binary) fractions. For example, the decimal fraction

```
0.125
```

has value 1/10 + 2/100 + 5/1000, and in the same way the binary fraction

```
0.001
```

has value 0/2 + 0/4 + 1/8. These two fractions have identical values, the only real difference being that the first is written in base 10 fractional notation, and the second in base 2.

Unfortunately, most decimal fractions cannot be represented exactly as binary fractions. A consequence is that, in general, the decimal floating-point numbers you enter are only approximated by the binary floating-point numbers actually stored in the machine.

The problem is easier to understand at first in base 10. Consider the fraction 1/3. You can approximate that as a base 10 fraction:

```
0.3
```

or, better,

```
0.33
```

or, better,

```
0.333
```

and so on. No matter how many digits you're willing to write down, the result will never be exactly 1/3, but will be an increasingly better approximation to 1/3.

In the same way, no matter how many base 2 digits you're willing to use, the decimal value 0.1 cannot be represented exactly as a base 2 fraction. In base 2, 1/10 is the infinitely repeating fraction

```
0.0001100110011001100110011001100110011001100110011...
```

Stop at any finite number of bits, and you get an approximation. This is why you see things like:

```
>>> 0.1
0.10000000000000001
```

On most machines today, that is what you'll see if you enter 0.1 at a Python prompt. You may not, though, because the number of bits used by the hardware to store floating-point values can vary across machines, and Python only prints a decimal approximation to the true decimal value of the binary approximation stored by the machine. On most machines, 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
```

instead! The Python prompt (implicitly) uses the builtin `repr()` function to obtain a string version of everything it displays. For floats, `repr(`*float*`)` rounds the true decimal value to 17 significant digits, giving

```
0.10000000000000001
```

`repr(`*float*`)` produces 17 significant digits because it turns out that's enough (on most machines) so that `eval(repr(`*x*`)) ==` *x* exactly for all finite floats *x*, but rounding to 16 digits is not enough to make that true.

Note that this is in the very nature of binary floating-point: this is not a bug in Python, it is not a bug in your code either, and you'll see the same kind of thing in all languages that support your hardware's floating-point arithmetic (although some languages may not *display* the difference by default, or in all output modes).

Python's builtin `str()` function produces only 12 significant digits, and you may wish to use that instead. It's unusual for `eval(str(`*x*`))` to reproduce *x*, but the output may be more pleasant to look at:

```
>>> print str(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.

Other surprises follow from this one. For example, after seeing

```
>>> 0.1
0.10000000000000001
```

you may be tempted to use the `round()` function to chop it back to the single digit you expect. But that makes no difference:

```
>>> round(0.1, 1)
0.10000000000000001
```

The problem is that the binary floating-point value stored for "0.1" was already the best possible binary approximation to 1/10, so trying to round it again can't make it better: it was already as good as it gets.

Another consequence is that since 0.1 is not exactly 1/10, adding 0.1 to itself 10 times may not yield exactly 1.0, either:

```
>>> sum = 0.0
>>> for i in range(10):
...     sum += 0.1
...
>>> sum
0.99999999999999989
```

Binary floating-point arithmetic holds many surprises like this. The problem with "0.1" is explained in precise detail below, in the "Representation Error" section. See *The Perils of Floating Point* for a more complete account of other common surprises.

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. `str()` usually suffices, and for finer control see the discussion of Pythons’s `%` format operator: the `%g`, `%f` and `%e` format codes supply flexible and easy ways to round float results for display.

B.1 Representation Error

This section explains the “0.1” example in detail, and shows how you can perform an exact analysis of cases like this yourself. Basic familiarity with binary floating-point representation is assumed.

Representation error refers to 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
0.10000000000000001
```

Why is that? 1/10 is not exactly representable as a binary fraction. Almost all machines today (November 2000) 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)
```

as

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

and recalling that *J* has exactly 53 bits (is `>= 2**52` but `< 2**53`), the best value for *N* is 56:

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

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(2L**56, 10)
>>> r
6L
```

Since the remainder is more than half of 10, the best approximation is obtained by rounding up:

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

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

```
7205759403792794 / 72057594037927936
```

Note that since we rounded up, this is actually a little bit larger than 1/10; if we had not rounded up, the quotient would have been a little bit smaller than 1/10. But in no case can it be *exactly* 1/10!

So the computer never "sees" 1/10: what it sees is the exact fraction given above, the best 754 double approximation it can get:

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

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

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

meaning that the exact number stored in the computer is approximately equal to the decimal value 0.100000000000000005551115123125. Rounding that to 17 significant digits gives the 0.10000000000000001 that Python displays (well, will display on any 754-conforming platform that does best-possible input and output conversions in its C library — yours may not!).

附录 C

History and License

C.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see http://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

Python 是在1990 年初由当时还在荷兰的Stitchting Mathematisch Centrum (CWI, http://www.cwi.nl/) 的Guide van Rossum 创作的。它是作为一种叫做ABC 编程语言的继承者而出现。目前Guide 仍然是Python 的主要作者, 然而许多其他的人士都为它作出了各种贡献。

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see http://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

到了1995 年, Guido 转到了Corporation for National Research Initiatives (CNRI, http://www.cnri.reston.va.us/) 继续Python 的开发工作。公司是位于维珍尼亚的Reston, 在那里他陆续发布了数个新的版本。

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see http://www.zope.com/). In 2001, the Python Software Foundation (PSF, see http://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

2000 年5 月, Guido 及核心开发小组搬到了BeOpen.com 并成立BeOpen PythonLabs 团队。同年的10 月, PythonLabs 团队又移师到了Digital Creations (now Zope Corporation; http://www.zope.com/)。至2001 年, Python Software Foundation (PSF, Python 基金会http://www.python.org/psf/) 成立了。它是一个非牟利的机构, 成立目的专门为了统管和Python 有关的智慧产权。而Zope Corporatoin 则是PSF 的其中一个赞助人。

All Python releases are Open Source (see http://www.opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

所有Python 的发布版本都是开放源码的(有关定义请参考http://www.opensource.org/)。以开发历史来说, 大部份, 但并非所有的Python 版本是兼容于GPL 的; 下表总结了各版本的情况。

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.2	2.1.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2.1	2.2	2002	PSF	yes
2.2.2	2.2.1	2002	PSF	yes
2.2.3	2.2.2	2002-2003	PSF	yes
2.3	2.2.2	2002-2003	PSF	yes
2.3.1	2.3	2002-2003	PSF	yes
2.3.2	2.3.1	2003	PSF	yes

发布版本	来源自	年份	拥有者	GPL 相容?
0.9.0 至1.2	不适用	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
2.2	2.1.1	2001	PSF	是
2.1.2	2.1.1	2002	PSF	是
2.1.3	2.1.2	2002	PSF	是
2.2.1	2.2	2002	PSF	是
2.2.2	2.2.1	2002	PSF	是
2.2.3	2.2.2	2002-2003	PSF	是
2.3	2.2.2	2002-2003	PSF	是
2.3.1	2.3	2002-2003	PSF	是
2.3.2	2.3.1	2003	PSF	是

Note: GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Note: GPL 相容并不是指我们提按GPL 形容来发布Python 的。所有Python 版本的使用证, 和GPL 不一样的是, 它容许你发行你自己修改过的版本而并不强制要求你也公开代码。所谓GPL 相容的意思是允许你把Python 结合到其他以GPL 形式发行的软件中; GPL 不相容的使用证并不容许你这样做。

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

在此向许多外面的志愿工作者致谢, 他们按照Guido 所定的方向, 致力工作, 以使这些版本能够成功实现及发布完成。

C.2 Terms and conditions for accessing or otherwise using Python

PSF LICENSE AGREEMENT FOR PYTHON 2.3.3

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.3.3 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 2.3.3 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2003 Python Software Foundation; All Rights Reserved" are retained in Python 2.3.3 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 2.3.3 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 2.3.3.

4. PSF is making Python 2.3.3 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 2.3.3 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.3.3 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.3.3, 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 2.3.3, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0
BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

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

附录 D

Glossary

>>> The typical Python prompt of the interactive shell. Often seen for code examples that can be tried right away in the interpreter.

`...` The typical Python prompt of the interactive shell when entering code for an indented code block.

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

byte code The internal representation of a Python program in the interpreter. The byte code is also cached in the `.pyc` and `.pyo` files so that executing the same file is faster the second time (compilation from source to byte code can be saved). This "intermediate language" is said to run on a "virtual machine" that calls the subroutines corresponding to each bytecode.

classic class Any class which does not inherit from `object`. See *new-style class*.

coercion Converting data from one type to another. For example, `int(3.15)` coerces the floating point number to the integer, `3`. Most mathematical operations have rules for coercing their arguments to a common type. For instance, adding `3+4.5`, causes the integer `3` to be coerced to be a float `3.0` before adding to `4.5` resulting in the float `7.5`.

descriptor Any *new-style* object that defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, writing *a.b* looks up the object *b* in the class dictionary for *a*, but if *b* is a descriptor, the defined 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.

dictionary An associative array, where arbitrary keys are mapped to values. The use of `dict` much resembles that for `list`, but the keys can be any object with a `__hash__()` function, not just integers starting from zero. Called a hash in Perl.

EAFP Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many `try` and `except` statements. The technique contrasts with the *LBYL* style that is common in many other languages such as C.

__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 actually 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)
```

generator A function that returns an iterator. It looks like a normal function except that the `yield` keyword is used instead of `return`. Generator functions often contain one or more `for` or `while` loops that `yield` elements back to the caller. The function execution is stopped at the `yield` keyword (returning the result) and is resumed there when the next element is requested by calling the `next()` method of the returned iterator.

GIL See *global interpreter lock*.

global interpreter lock The lock used by Python threads to assure that only one thread can be run at a time. This simplifies Python by assuring that no two processes can access the same memory at the same time. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of some parallelism on multi-processor machines. Efforts have been made in the past to create a "free-threaded" interpreter (one which locks shared data at a much finer granularity), but performance suffered in the common single-processor case.

IDLE An Integrated Development Environment for Python. IDLE is a basic editor and interpreter environment that ships with the standard distribution of Python. Good for beginners, it also serves as clear example code for those wanting to implement a moderately sophisticated, multi-platform GUI application.

immutable A object with fixed value. Immutable objects are numbers, strings or tuples (and more). Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed. For example as a key in a dictionary.

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, a 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__*.

interactive Python has an interactive interpreter which means that you can try out things and directly see its result. Just launch `python` with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`).

interpreted Python is an interpreted language, opposed to a compiled one. This means that the source files can be run right away without first making an executable which is then run. Interpreted languages typically have a shorter development/debug cycle than compiled ones. See also *interactive*.

iterable A container 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 builtin 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 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 is 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 that 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 from the second iteration pass, making it appear like an empty container.

list comprehension A compact way to process all or a subset of 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 hex numbers (0x..) that are even and in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed in that case.

mapping A container object (such as `dict`) that supports arbitrary key lookups using the special method `__getitem__()`.

metaclass The class of a class. Class definitions create a class name, a class dictionary, and a list of base classes. The metaclass is responsible for taking those three arguments and creating the class. Most object oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses can provide powerful, elegant solutions. They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.

LBYL Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the *EAFP* approach and is characterized the presence of many `if` statements.

mutable Mutable objects can change their value but keep their `id()`. See also *immutable*.

namespace The place where a variable is stored. Namespaces are implemented as dictionary. There is the local, global and builtins namespace and the 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 modules implement 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 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 Any class that 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, `__getattribute__()`, class methods, and static methods.

Python3000 A mythical python release, allowed not to be backward compatible, with telepathic interface.

__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 An *iterable* which supports efficient element access using integer indices via the `__getitem__()` and `__len__()` special methods. 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.

Zen of Python Listing of Python design principles and philosophies that are helpful in understanding and using the language. The listing can be found by typing "`import this`" at the interactive prompt.

附录 E

翻译团队

- **刘鑫**: . . .
- glace : **后勤**