Python 指南 v2.6
简体中文版

发布版本： 2.6
发布日期： 2010/04/28

译者：Moonist 邮件：wisherwinfe@126.com 博客：moonist.susan.blog.163.com

目录

前言

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 是一门简单易学、功能强大的编程语言。它拥有高效的高级数据结构，以及简单而有效的面向对象编程方法。 Python 优雅的语法和动态输入，以及其解释语言特性，使之在很多场合下，它都是绝佳的脚本和快速应用开发工具。

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 may 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 支持多种主流平台， Python 解释器和大量的标准库都可以直接从官方网站 http://www.python.org/ 获得，并且可以免费发布。官方网站还包含了许多第三方的 Python 模块、应用程序、工具以及文档。

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 解释器能够方便的扩展到 C/C++（其他可以从 C 语言调用的语言）编写的新的数据结构和函数。Python 亦非常适合作为传统应用的扩展语言。

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.

该指南简单的介绍了 Python 的语言特性和系统特性，它能够让读者快速上手，不过该指南中所有的例子都是独立的，你可以离线阅读。（该文档来源于 Python 官方网站。译者注）

For a description of standard objects and modules, see *The Python Standard Library*. *The Python Language Reference* 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 Manual*. There are also several books covering Python in depth.

如果您想了解标准对象和模块，请参阅 *The Python Standard Library*. *The Python Language Reference* 对该语言进行了更加正式的定义。 如果您想编写 C/C++扩展，请参阅 *Extending* *and Embedding the Python Interpreter* 以及 *Python/C API Reference Manual*。还有其他的一些书籍也对 Python 进行了深入的讲解。

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 Standard Library*.

该指南并不试图全面介绍 Python 语言，也无意覆盖 Python 语言的每个细节，甚至一些常用的特性也未予以介绍。不过，该文档介绍了 Python 最为显著的特征，并且还会给出该语言的喜好与风格。读完该文档后，您就可以自己动手写模块和程序，你也可以在 *The Python Standard Library* 学习更多关于 Python 的库模块。

The *Glossary* is also worth going through.

*Python 词汇表*也非常值得一看。

译者说明

Python v2.6 是 Python v3.0 与 v2.0 之间的一个过渡版本，推出于 2010 年 4 月。它在 v2.5 的基础上略微有一些改动，但是改动幅度并不大。

该《Python 指南 v2.6 简体中文版》的翻译源为 Python 官方网站的《The Python Tutorial》，并参考了网上的一个《The Python Tutorial v2.5 简体中文版》，事实上，由于本人精力有限，有一些章节的翻译直接借鉴于该文档。不过该文档中并未提及翻译者，在此对该文档作者表示衷心感谢。另外为了防止翻译出的中文内容与原内容有偏差或者不能理解，我将中英文对照起来，以方便对比。

由于翻译实践仓促，以及本人翻译水平的限制，本文的翻译还有许多错误以及可以改进的地方，并且其中还有许多翻译未前后统一的地方，希望大家予以指正并发送到我的邮箱 wisherwinfe@126.com，非常感谢您的参与。另外，由于排版因素，我删除了一些尾注。最后的三章没有进行翻译，一来是翻译工作繁重，已经不堪重负，二来，所参考的文件及其他的翻译版本对这三章也未进行翻译，三来，这三章属于额外的知识，读者可以选择进行阅读，故而放弃翻译，请谅解。

本文亦在我的博客上进行了部分转载（除了最后三章），欢迎查看 http://moonist.susan.blog.163.com。

以下是部分词语翻译对照表：

List	列表
Tuple	元组
Instance	实例
List comprehension	列表推导式
Sequence	队列

Moonist

2010/07/01   于华工

1. Whetting Your Appetite 激起你的兴趣

If you do much work on computers, eventually you find that there’s some task you’d like to automate. For example, you may wish to perform a search-and-replace over a large number of text files, or rename and rearrange a bunch of photo files in a complicated way. Perhaps you’d like to write a small custom database, or a specialized GUI application, or a simple game.

如果你在计算机上做过很多事情，你一定非常希望一些事情可以自动进行。例如，你希望在一大堆文本文件中进行查找替换，或者对很多照片以一种复杂的方式进行命名和排序。或许你还希望写一个小的数据库，或者一个特定的 GUI 程序，或者是一个简单的游戏。

If you’re a professional software developer, you may have to work with several C/C++/Java libraries but find the usual write/compile/test/re-compile cycle is too slow. Perhaps you’re writing a test suite for such a library and find writing the testing code a tedious task. Or maybe you’ve written a program that could use an extension language, and you don’t want to design and implement a whole new language for your application.

如果你是一个专业的软件开发人员，你或许在处理一些 C/C/++/JAVA 库，发现“写-编译-测试-重新编译”循环太慢？也许你为某个库写了一个测试用例，却发现写起来非常单调乏味。或者你写的程序会使用扩展语言，但是你却不希望设计并实现一门全新的语言。

Python is just the language for you.

Python 正好适合你。

You could write a Unix shell script or Windows batch files for some of these tasks, but shell scripts are best at moving around files and changing text data, not well-suited for GUI applications or games. You could write a C/C++/Java program, but it can take a lot of development time to get even a first-draft program. Python is simpler to use, available on Windows, Mac OS X, and Unix operating systems, and will help you get the job done more quickly.

也许你可以写一些 Unix Shell 脚本或者 Windows 批处理来处理这些事情，但是脚本最擅长于文件移动与文件内容修改，却不适合于 GUI 或者游戏。你可以写一个 C/C++/JAVA 程序，耗费大量的开发时间，得到的仅仅只是一个初步的草稿。 Python 使用起来非常简单，并且在 Windows、Mac OS X，以及 Unix 系统上都可以使用，而且会帮助你更快的完成这些任务。

Python is simple to use, but it is a real programming language, offering much more structure and support for large programs than shell scripts or batch files can offer. On the other hand, Python 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. 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.

Python 简单易用，而且它也是一门真正的编程语言，亦能够在大型程序中比脚本或者批处理提供更多的结构与支持。另外，Python 也提供比 C 更多的错误验证，并且作为一种*甚高级语言*，它有高级的内置类型，比如可变队列和字典。因为它拥有更多的普遍的数据类型，Python 比 Awk 甚至 Perl 更适合大规模应用，而且在 Python 中处理这些问题不会比在其他语言中更难。

Python allows you to split your program into 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. Some of these modules provide things like file I/O, system calls, sockets, and even interfaces to graphical user interface toolkits like Tk.

Python 允许将程序写成模块以便于在其他程序中重用。在 Python 中有大量的标准模块，这些可以作为你的程序的基石，也可以作为你开始学习的例子。这些标准模块提供了如文件 I/O、系统调用、套接字甚至是如 Tk 的 GUI 工具。

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 是一门解释性语言，你可以节约相当可观的编译、连接时间，因为在 Python 中你无需编译或者链接。解释器可以交互式的使用，你可以很方便的对语言特性进行实验或者写一些临时程序或者在程序开发过程中的一些测试函数。另外，它也是一个很好用的桌面计算器。

Python enables programs to be written compactly and readably. Programs written in Python are typically much shorter than equivalent C, C++, or Java programs, for several reasons:

Python 能够让程序写的紧凑而且易读，Python 所写的程序相比于 C/C++/JAVA 所写的程序来说会小很多，原因如下：

- the high-level data types allow you to express complex operations in a single statement;
- 高级数据结构使得你可以在单个表达式中实现复杂的操作；
- statement grouping is done by indentation instead of beginning and ending brackets;

- 语句的分组采用缩进方式而不是采用头尾的花括号；
- no variable or argument declarations are necessary.
- 变量或者参数无需定义；

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*：如果你知道怎么样使用 C 语言，为了实现运算的最大速度或者将 Python 程序链接到其他二进制库中（例如一些 vendor-specific 图形库），你可以非常方便的在解释器中加入一个新的内置函数或者模块。一旦成功勾上，你可以将 Python 解释器链接到 C 语言程序中，将其作为扩展语言或者命令语言。

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

顺便提及一下，Python 是以 BBC 的《Monty Python’s Flying Circus》命名的，不过和这个爬虫没有什么关系。在文档中将 Monty Python skits 与 Python 语言联系起来却并不被阻止，反而我们非常乐意您这么做。

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 to use it, the tutorial invites you to play with the Python interpreter as you read.

现在，你应该对 Python 充满热情吧，也许你想要马上试验一下。使用是学习一门语言的最好方法，本指南将带你与 Python 解释器互动起来！

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.

在下一个章节中，我们将详细说明使用解释器的方法。这些只是一些简单的信息，但是对于测试后面的例子至关重要。

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.

接下来的章节，我们将通过例子介绍 Python 的各种特性，从最简单的表达式、语句和数据类型，再到函数、模块然后最后接触一些高级的概念比如异常或者是自定义类。

2. Using the Python Interpreter 使用 Python 解释器

2.1. Invoking the Interpreter 调用解释器

The Python interpreter is usually installed as Python/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。

解释器通常安装在/usr/local/bin/python。将/usr/local/bin 置入您的 Shell 环境变量路径，你可以再 Shell 中调用该解释器。

```
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.)

由于 Python 解释器可以根据需要安装在其他文件夹，所以对于您的系统，也许你需要确认本地安装路径或者系统管理员。（比如，/usr/local/python 是一个大家经常使用的地方。）

On Windows machines, the Python installation is usually placed in C:\Python26, though you can change this when you’re running the installer. To add this directory to your path, you can type the following command into the command prompt in a DOS box:

在 Windows 操作系统的机器上，Python 通常被安装在 C:\Python26，不过你可以在安装的时候选择不同的目录进行安装。为了将该路径加入到环境变量，你需要在 DOS 窗口中键入如下命令：

```
set path=%path%;C:\python26
```

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 command: quit().

在解释器中的提示符后输入文件结束符（在 Unix 下是 Control-D，在 Windows 下是 Control-Z）

将使解释器以零状态退出。如果这样不管用，可以键入如下命令退出：quit()。

The interpreter’s line-editing features usually aren’t very sophisticated. On Unix, whoever installed the interpreter may have enabled support for the GNU readline library, which adds more elaborate interactive editing and history features. Perhaps the quickest check to see whether command line editing is supported is typing Control-P to the first Python prompt you get. If it beeps, you have command line editing; see Appendix *Interactive Input Editing and History Substitution* 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.

Python 解释器的命令行编辑特性并不那么高端。在 Unix 上，在安装解释器的同时，也会是 GNU 的 readline 库生效，readline 库能够让命令行编辑更加具有互动性，编辑历史的特性也更为强大。检查命令行编辑特性是否支持的最快方式是在 Python 解释器的第一个提示符后输入 Control-P。如果发出响声，说明你已经拥有了命令行编辑器；参阅附录 *Interactive Input Editing and History Substitution* 获得更多关于此的信息。如果任何事情也没有发生或者仅仅是一个^P 字符被打出，你的解释器并不支持命令行编辑特性，你只能够用 Backspace 键删除当前行。

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.

解释器工作起来和 Unix 的 Shell 类似：当调用由标准输入连接到一个 tty 设备，它交互式的读取和执行命令；当以文件加参数的形式调用作为标准输入，它从文件中以脚本的方式进行执行。

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 usually advised to quote *command* in its entirety with single quotes.

第二种启动解释器的方式是 python -c command [arg] ...，这种方式下，*command* 将被执行，与 shell 的 *-c* 选项类似。由于 Python 语句中有很多 shell 环境中如空格等特殊的字符，所以你最好将 *command* 用单引号括起来。

Some Python modules are also useful as scripts. These can be invoked using python -m module [arg] ..., which executes the source file for *module* as if you had spelled out its full name on the command line.

一些 Python 模块和脚本一样方便。这些模块可以用 python -m module [arg] ...的方式调用，这种方式执行了模块的源代码，就像在命令行上敲出了模块的全名一些样。

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.

值得注意的是，python file 和 python <file 有所不同。后者中的输入如 input() and raw_input() 都要在文件中满足。由于程序将一直读取文件，然后解析并开始执行程序，这时程序将立刻碰到文件结尾符。而在前者中（这正是大部分时候你想要的方式），输入可以从任意的标准输入设备进行输入。

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.)

当你使用一个脚本文件，有时候你希望运行完脚本后进入交互模式，你可以再脚本前写入*-i* 命令。（但是正如前段所说，如果该脚本是从标准输入获得的，这种方式将不起作用。）

2.1.1. Argument Passing 参数传递

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'. When *-m module* is used, sys.argv[0] is set to the full name of the located module. Options found after *-c command* or *-m module* are not consumed by the Python interpreter's option processing but left in sys.argv for the command or module to handle.

命令输入后，脚本名称和额外的参数将通过字符串列表变量 sys.argv 传入。它的长度最少为 1；当没有脚本或者参数给出，sys.argv[0]是一个空字符串。当脚本名称以'-'（标准输入）给出，sys.argv[0] 被设置为'-'。当 *-c command* 被使用，sys.argv[0]被设置为'-c'。当 *-m module* 被使用，sys.argv[0] 被设置为模块的全名。 在*-c command* 或 *-m module* 后的选项并不被 Python 解释器进行选项处理，它会把它们置入 sys.argv 中以便于命令或者模块对它们进行处理。

2.1.2. Interactive Mode 交互模式

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:

当命令从 tty 中读入，我们称解释器处于*交互模式*。在这种模式下，解释器在*提示符*后等待命令输入，一般情况下，由三个大于号(>>>)组成的组成的提示符等待命令输入；对于连续行的输入，解释器将显示*第二提示符*，默认情况下，它是三个点号(...)。在解释器输入第一个命令提示符之前，它将打印一段欢迎信息，包括版本号和版权说明。

```
python
Python 2.6 (#1, Feb 28 2007, 00:02:06)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

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

当我们需要键入一个多行的结构时，我们需要使用连续行。比如，看看这个 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. The Interpreter and Its Environment 解释器及环境

2.2.1. Error Handling 错误处理

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 executed commands is written to standard output.

当错误发生时，解释器打印出错误信息和堆栈信息。在交互模式下，它会返回到主提示符;如果输入来自一个文件，它会在打印堆栈信息后以非零状态退出。(被 except 从句 try 语句处理的异常不在此范围之列。)一些错误使一些非条件致命错误，会导致解释器以非零状态退出；比如外部不协调或者内存访问错误错误。所有的错误信息都被写入到保准错误流，其

他一些正常的信息被输出到标准输出。

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.

在主提示符后输入中断符（通常是 Control-C 或者 DEL）将取消输入并返回到主提示符。[1] 在命令正在执行时输入中断符将导致程序抛出 KeyboardInterrupt 异常，该异常可以被 try 语句进行处理。

2.2.2. Executable Python Scripts 可执行 Python 脚本

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

在类 BSD 或者 Unix 操作系统上，Python 脚本可以像 shell 脚本一样直接成为可执行的，只需要在脚本文件的开始写下

```
#! /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 Windows ('\r\n') line ending. Note that the hash, or pound, character, '#', is used to start a comment in Python.

（假设该路径在 **PATH** 中。）并且将脚本修改为可执行模式。#!必须是文件的头两个字符。在某些平台上，该行必须以 Unix 形式的换行符结束('\n')，而不是 Windows 的('\r\n')。需要注意的是，hash、pound 或者字符'#'将表示一条注释。

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

可以使用 **chmod** 命令将脚本设置为可执行模式或者权限。

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

On Windows systems, there is no notion of an "executable mode". The Python installer automatically associates .py files with python.exe so that a double-click on a Python file will run it as a script. The extension can also be .pyw, in that case, the console window that normally appears is suppressed.

在 Windows 操作系统上，没有可执行模式。Python 安装程序自动将.py 与 python.exe 关联起

来，你可以通过双击一个 Python 脚本文件运行。扩展名也可以是.pyw，在这种情况下，命令行窗口将不显示。

2.2.3. Source Code Encoding 源代码编码

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:

可以使用不同于 ASCII 编码的编码来编辑 Python 源代码。最好的方法是在文件中的#!行的后面写入如下的注释行：

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

With that declaration, all characters in the source file will be treated as having the encoding *encoding*, 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.

这样声明以后，源文件中的所有字符都将以 *encoding* 的编码对待，这样就可以直接在源代码中以选择的编码方式写入 Unicode 字符了。所有可能的编码列表能够在 Python Library Reference 的 codecs 章节获得。

For example, to write Unicode literals including the Euro currency symbol, the ISO-8859-15 encoding can be used, with the Euro symbol having the ordinal value 164. This script will print the value 8364 (the Unicode codepoint corresponding to the Euro symbol) and then exit:

比如，为了书写包括欧元符在内的 Unicode，可以使用 ISO-8859-15编码，欧元符在其中的值为164。下面的脚本将输出8364（欧洲符号的 Unicode）然后退出：

```
# -*- coding: iso-8859-15 -*-

currency = u"€"
print ord(currency)
```

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 script files with #! lines (only used on Unix systems).

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-ASCII characters 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.

如果你的编辑器支持将文件使用 UTF-8标志（aka BOM）保存为 UTF-8编码，你可以使用它作为编码声明。如果 Options/General/Default Source Encoding/UTF-8被设置，IDLE 将拥有这种功能。不过值得注意的是，这种签名并不被较老的 Python 版本（2.2或者更早）支持，也不被以脚本以#!开头的操作系统支持（仅仅只是在 Unix 操作系统上）。如果使用 UTF-8编码，世界上几乎所有的文字都可以同时在字符串或者注释行中使用。 Python 不支持非 ASCII 字符的标识符。为了正确的显示这些字符，你的编辑器必须支持 UTF-8，而且还必须选择一种支持所有字符的字体。

2.2.4. The Interactive Startup File 交互式启动文件

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.

当你交互式的使用 Python，将一些标准的命令在解释器启动的时候就执行非常频繁且方便。你可以设置环境变量 **PYTHONSTARTUP** 为一个含有这些启动命令的文件。这和 Unix Shell 中的.profile 特性非常类似。

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.

该文件只有在交互式会话中才会被调用，在 Python 从文件中读取脚本甚至是/dev/tty 被显式的指定为命令源（另一种交互式会话）时都不会起作用。它和交互式命令执行的名字空间一致，所以在该文件中定义的或者导入的对象都可以再交互式会话中直接使用。你也可以通过文件 sys.ps1和 sys.ps2修改提示符。

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:

如果你想从当前文件夹读取额外的启动文件，你可以在全局启动文件中写入 if

os.path.isfile('.pythonrc.py'): execfile('.pythonrc.py')的代码。如果你想在脚本中使用启动文件，你必须显式的在脚本中指出：

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

3. An Informal Introduction to Python Python 概要介绍

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.

在下面的例子中，输入和输出以显示提示符(>>>和...)和不显示提示符进行区分：如果你想重复这些例子，你必须在提示符出现后键入提示符后的所有内容；没有以提示符开始的行是解释器的输出。需要注意的是，第二提示符中自动出现的行意味着你必须输入一个空行，这种方式用来结束一个多行的命令。

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. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

该指南中的许多例子，甚至是那些在交互模式下输入的命令，都包括注释行。Python 的注释行以#开始，直到该行的结束都是注释行。注释可能出现在一行代码之前或者空格之后或者是代码的后面，但是不会出现在一个字符串中。在字符串中的#字符只是一个字符。由于注释只是用于帮助理解代码，而且它也不会被 Python 解释，所以当你试验这些例子的时候可以省略它们。

Some examples:

一些例子：

```
# 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. Using Python as a Calculator 将 Python 作为一个计算器

Let’s try some simple Python commands. Start the interpreter and wait for the primary prompt,

```
>>>. (It shouldn’t take long.)
```

让我们先来试一些简单的 Python 命令。启动解释器并等待提示符的出现，>>>。（这不会花很长时间。）

3.1.1. Numbers 数字

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:

这个解释器可以当一个简单的计算器使用：你可以输入一个表达式，它会输出表达式的值。表达式的语法很简单：操作符+, -, *和/与其他语言中的功能一样（比如 Pascal 或者 C）；小括号可以作为分组。比如：

```
>>>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
```

The equal sign ('=') is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

等号('=')用于给一个变量赋值。直到下个交互式提示符出现之前，没有结果会打印出来。

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

A value can be assigned to several variables simultaneously:

一个值可以同时赋值给多个变量：

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

Variables must be “defined” (assigned a value) before they can be used, or an error will occur:

变量在使用前必须被定义（赋值），否则将出现错误：

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

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

对浮点数的支持也很全面；混合运算中整型数值将被转化为浮点数值：

```
>>>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+imagj), or can be created with the complex(real, imag) function.

Python 也支持复数，虚部表示为带有后缀 j 或者 J。实部不为零的复数表示为(real+imagj)，或者可以使用函数 complex(real, imag)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.

复数通常被表示为两个浮点数值，实部和虚部。要从复数 *z* 中提取实部和虚部，使用 z.real 和 z.imag。

```
>>>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.

浮点数转换和整型数值转换函数(float(), int() and long())在复数上不起作用。没有什么方法可以从复数转化为实数。使用 abs(z)获取它的模，或者用 z.real 获取它的实数部分。

```
>>>a=3.0+4.0j
>>>float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
```

```
TypeError: can't convert complex to float; use abs(z)
>>>a.real
3.0
>>>a.imag
4.0
>>>abs(a)  # sqrt(a.real**2 + a.imag**2)
5.0
```

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:

在交互模式中，最后一条表达式的值被赋值给_。这意味着如果你将 Python 作为一个桌面计算器使用，很容易就可以继续计算，比如：

```
>>>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.

这个变量对用户是只读的。不要显式的对它赋值，那样只会建立一个与它同名的局部变量，并发生奇怪的行为。

3.1.2. Strings 字符串

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 也可对字符串进行操作，它可以通过多种方式表示。它可以使用单引号或者双引号括起来：

```
>>>'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:

字符串能够通过多种方式扩展到多行。可以使用连续行，即在行结尾使用一个反斜线，这样下一行成为上一行的逻辑扩展：

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

值得指出的是，表示新的一行必须使用\n；以反斜线起头的新行将被忽略。下面的例子应该打印如下内容：

```
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.
```

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.

字符串亦可以用一对三引号："""或 '''括起来。不必指定行结尾，行结尾将被包括在字符串中。

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

produces the following output:
```
产生如下的输出：

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```
If we make the string literal a "raw" string, \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:

如果字符串为"原始"字符串，\n 序列不会被转化为换行符。看下面的例子：

```
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:
```
将输出：

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

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.)

解释器以输入的方式打印字符串操作的结果：引号里面的内容、引号和其他一些有趣的以反划线标识的字符，都会显示原始内容。如果一个字符串里面包括单引号而且没有双引号，那么字符串最好用双引号括起来，否则用单引号。(稍后将解释的 print 语句，能够无需引号或

者标识符来写入字符串。)

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

字符串能够使用+号连接起来，也可以使用*进行重复：

```
>>>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:

两个连续的字符串将被自动的连接起来；上面的第一行也可以被写成 word = 'Help' 'A'；这仅仅在两个字面意义上的串上才有用，对于字符串表达式不起作用：

```
>>>'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.

字符串还能够使用下标引用；和 C 类似，字符串中的第一个字符的下标为0。这里没有字符类型；一个字符是一个大小为1的字符串。和 Icon 中类似，能够用*切片引用*来表示一个子字符串：两个被逗号分开的下标。

```
>>>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.

切片下标有一些相当有用的默认值；如果第一个下标被忽略，那么它默认为0，第二个下标被忽略，则默认为字符串的大小。

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

与 C 字符串不同，Python 的字符串不能够被改变。对字符串的某个下标赋值将会产生错误：

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

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

不过，创建一个与内容绑定的字符串简单而且高效：

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

Here’s a useful invariant of slice operations: s[:i] + s[i:] equals s.

这里有一个很有用的切片操作诀窍：s[:i] + s[i:]等于 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.

不符合规则的切片下标也非常好处理：如果下标太大，它将被替换为字符串的大小，而如果上界小于下界将返回一个空字符串。

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

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

下标也可以为负数，从右边开始计数。如下：

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

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

不过值得指出的是-0与0一模一样，它不从右边开始计数！

```
>>>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:

超出边界的负数下标将被截断，不过不要试着在单下标切片中使用：

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

One 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:

理解切片如何操作的一种方法是将下标想象成字符的指针，最左边的字符指向0.那么右边的第 *n* 个字符指向数字 *n*，如下所示：

```
 +---+---+---+---+---+
 | 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.

第一行数字给出了字符串中0...5下标的位置，第二行给出了相应的负值下标。从 *i* to *j* 的切片分别由 *i* and *j* 所标记的边界中的所有字符组成。

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.

对于非负下标，如果它们都处于字符串边界内，切片的长度是切片下标的差。比如，word[1:3]的长度为2。

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

内置的 len()函数返回一个字符串的长度：

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

See also

还可以参阅

Sequence Types — str, unicode, list, tuple, buffer, xrange

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

字符串以及下章节讲述的 Unicode 字符串都是*序列*对象，都支持共同的操作。

String Methods

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

String Formatting

Information about string formatting with str.format() is described here.

String Formatting Operations

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

3.1.3. Unicode Strings Unicode 字符串

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，一种新的用于存储字符数据的数据类型出现在编程者面前：Unicode 对象。

它可以用来存储和操作 Unicode 数据(参考 http://www.unicode.org/)，并且和现有的字符串对象集成的很好，它们之间可以在必要的时候进行自动转换。

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

Unicode 有为现代和古老的所有脚本的所有字符提供单一编号的优势。在以前，仅仅有256个字符用于脚本编写。文本被绑定到一个页面代码，而该页面代码指向了脚本字符的序列编号。这样引起很多的混淆，尤其是软件进行国际化（通常被写成 i18n — 'i' + 18个字符 + 'n'）的时候。Unicode 解决了这些由页面代码所造成的问题。

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

在 Python 中创建 Unicode 字符串与创建普通的字符串一样简单：

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

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

字符串引号前小写的'u'表示将创建一个 Unicode 字符串。如果你想要在字符串中包含一些特殊的字符，你可以使用 Python*Unicode-Escape* 编码。下面的例子将示例如何进行：

```
>>>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.

转义序列\u0020表示在给定位置插入序列号为0x0020（空格符）的字符。

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.

其他的字符直接使用它们对应的原始值作为 Unicode 的原始值来进行解释。如果你有西方国家经常使用的标准 Latin-1编码的字符串，你会发现使用起来非常方便，因为最小的256个字符的 Unicode 值与256个最小的 Latin-1的字符值一模一样。

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

对于专家来说，也有一种原始的表示字符串的方法。你需要使用'ur'作为前缀，这样 Python 会使用 *Raw-Unicode-Escape* 编码。如果在'u'前有非偶数个反划线，它会在\uXXXX 的字符串上执行转换。

```
>>>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.

这种原始方法最有用的情况是在正则表达式中当你需要键入很多的反划线的时候。

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

除了这些标准的编码，Python 还提供了一整套创建其他已知编码 Unicode 字符串的方法。

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.

内建函数 unicode()提供了对所有 Unicode 编解码的方法。这些 Unicode 中比较常见的有 *Latin-1*, *ASCII*, *UTF-8*, and *UTF-16*。后两者采用变长编码方式，也就是说每个字符根据不同的情况采取不同的字节进行编码。默认的编码被设置为 ASCII，这种编码只允许0-127的编

码，其他的值都会视为错误。当一个 Unicode 字符串被打印、写入文件或者被 str()转变，都是采用的默认编码进行转换。

```
>>>u"abc"
u'abc'
>>>str(u"abc")
'abc'
>>>u"äöü"
u'\xe4\xf6\xfc'
>>>str(u"äöü")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
```

UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-2: ordinal not in range(128)

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

如果要将一个 Unicode 字符串采用特定的编码转换为一个8比特的字符串，Unicode 对象提供了带一个参数的 encode()函数，即编码名称。最好是提供小写的编码名称。

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

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

如果你有一串在特定编码下的数据需要转化为 Unicode 字符串，你可以查看 unicode() ，该函数的第二个参数为编码名称。

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

3.1.4. Lists 列表

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 拥有许多复合的数据结构，可以作为其他一些值的集合。功能最多的莫过于列表（*list*），它能够写成以方括号括起来的以逗号分开的值的集合。列表的元素可以拥有不同类型。

```
>>>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:

和字符串的下标类似，列表的下标起于0，也可以被切片，连接等等：

```
>>>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] + ['Boo!']
['spam','eggs',100,'spam','eggs',100,'spam','eggs',100,'Boo!']
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a shallow copy of the list *a*:

所有的切片都返回包含所请求元素的新列表。这意味着如下的切片返回 *a* 的浅切片：

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

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

与字符串可能改变不同，列表中的元素可以被修改：

```
>>>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 or clear it entirely:

对切片进行赋值也变为可能，甚至你还可以改变列表的大小或者完全清除其内容：

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

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

内置函数 len()也可以使用在列表上：

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

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

列表还可以被嵌套（在列表中嵌套其他的列表），如下：

```
>>>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.

注意在最后一个例子中，p[1]和 q 指向同一个对象，我们将稍后解释*对象语义学*。

3.2. First Steps Towards Programming 编程的第一步

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

当然，我们我们可以让 Python 做比2+2更复杂的任务。比如，我们可以写下如下的 *Fibonacci* 数列初始序列：

```
>>># 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.

该例子引入了一些新的特性。

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.

第一行包含了一个*多变量赋值*：变量 a 和 b 同时获得值0和1。在最后一行，这种特征又被使用了一次，说明右边的表达式总是比赋值要先进行计算。右边的表达式从左到右进行计算。

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 循环在条件为 true 的时候总是执行（这里指 b < 10）。在 Python 中，与 C 类似，任何的非0值都为真，而0则为假。条件也可以是一个字符串或者是列表，实际上，任意的非零序列都可以作为条件真，而空序列则表示假。在例子中的条件测试只是一个简单的比较。标准的比较操作符和 C 中一样：< (less than), > (大于), == (等于), <= (小于或等于), >= (大于或等于) and != (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.

循环的部分进行了*缩进*：在 Python 中，缩进是一种将语句分块的方法。Python 到目前为止还没有提供一个智能的编辑器，所以你需要在每行自己键入 tab 或者空格键。在实际中，你会在文本编辑器中进行更复杂的编辑，大部分的文本编辑器都可以自动缩进。当一个复合语

句被键入，必须键入一个空白行以表示该语句的结束（由于解释器无法猜测你什么时候输入了最后一行）。值得注意的是，同一块的内容都必须拥有同样的缩进。

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 语句打印出给定表达式的值。写下你所需要的表达式与 Python 处理多重表达式和字符串不同。字符串打印出来不含有引号，元素之间会被插入空格，所以你可以非常方便的格式化，比如：

```
>>>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:

尾部的逗号防止在新的一行输出。

```
>>>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.

注意，如果上一行未完成，解释器将在打印下一个提示符之前插入新的一行。

4. More Control Flow Tools 深入流程控制

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

除了刚刚介绍的 while 语句，Python 也有与其他语言中一样的流程控制语句，不过稍微有一些变化。

4.1. if Statements if 语句

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

也许最出名的语句就是 if 了，例如：

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

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

在 if 从句中，可以有多个 elif 语句，而 else 语句是可选的。关键字'elif'是'else if'的简写，可以用来处理 if 以外的情况。if ... elif ... elif ...序列是作为其他语言中 switch 或 case 的替换。

4.2. for Statements for 语句

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 语句与 C 或者 Pascal 中的有一些区别。与总是对一个算数运算或者数值（像 Pascal 中一样）进行迭代，或者是让用户可以同时定义迭代步长与终止条件（与 C 中类似），Python 的 for 语句在任意序列（list 或者 string）中以每项在序列中的顺序进行迭代。比如（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:

在被进行循环迭代的序列上修改序列（这种情况只有可能在可修改的序列类型上进行，比如列表）是危险的。如果你的确需要进行修改（比如，重复被选中的项），你最好在一个拷贝上进行迭代。切片标记使得这种修改格外的方便：

```
>>>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. The range() Function range()函数

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

如果你需要在一个数值序列上进行迭代，内置的 range()函数就会用得非常顺手。它将生成算数数列的列表：

```
>>>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, 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'):

所给出的结束点不会出现在列表中；range(10)生成一个拥有10个数值的列表，也就是一个长度为10的序列的合法索引值组。也可以使起始点为另外一个值，或者指定递增长度（负值也是合法的；一些情况下我们也称之为步长）：

```
>>>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, you can combine range() and len() as follows:

为了遍历序列的所有下标，你可以将 range()和 len()像下面的样子组合起来：

```
>>>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
```

In most such cases, however, it is convenient to use the enumerate() function, see *Looping Techniques*.

然而在大多数这样的情况下，用 enumerate()会显得更方便，请参阅 *Looping Techniques*。

4.4. break and continue Statements, else Clauses on Loops 循环中的 break 和 continue 语句以及 else 从句

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

break 语句与 C 类似，跳出当前最小的循环域 for 或者 while。

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

而亦是借鉴于 C 的 continue 语句终止当前的一个循环而开始下一次循环。

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

循环语句可以有一个 else 从句；当循环因为列表遍历完（for）或者判定条件为假（while），而不是由 break 语句终止时，该从句会被执行。下面的寻找质数的循环中将证实这种情况：

```
>>>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 Statements pass 语句

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

pass 语句不做任何事情。在语法上需要而程序上不需要的情况下可以使用，比如：

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

This is commonly used for creating minimal classes:

在创建类的时候也经常用到：

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

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

pass 的另外一个用途是作为书写新代码时函数或者条件块的占位符，这样可以使你思考更多抽象层次的东西。pass 被忽略。

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

4.6. Defining Functions 定义函数

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

我们可以创建一个任意边界的 Fibonacci 数列：

```
>>>def fib(n):    # write Fibonacci series up to n
```

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

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.

def 标明正在进行函数*定义*。在其后必须跟上函数名和括号括起来的形参列表。函数的语句从第二行开始书写，而且必须缩进。

The first statement of the function body can optionally be a string literal; this string literal is the function’s documentation string, or *docstring*. (More about docstrings can be found in the section *Documentation Strings*.) 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 make a habit of it.

函数的第一行语句可以是一个可选的字符串；这个字符串是该程序的说明文档字符串，或者称为 *docstring*。（更多的关于 docstring 的信息请参见 *Documentation Strings*。）一些工具可以使用 docstring 创建在线的或者打印的文档，或者让用户交互的浏览代码；在你写的代码中加入 docstring 是一种好的实践，所以要将其当做一种习惯。

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

函数的*执行*引入了一个新的局部符号表，用于存放函数的局部变量。更加准确的说，在函数中所有的变量赋值都将存储在局部符号表中；对变量进行引用的时候，首先会在局部符号表中查找，然后在包含此函数的函数局部符号表中查找，最后到全局符号表查找，最后到内建名字表。于是，全局的变量不能够在函数中直接被赋值（除非名字使用了 global 语句），尽管他们可能被引用到。

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.

函数调用的实参在函数调用时被引入函数的局部符号表中；这样，参数都以 *call by value* 的形式调用（这里的 *value* 总是表示一个对象的引用，而非对象本身）。[1]当一个函数调用另外一个函数，一个新的局部符号表将被创建。

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:

函数定义将函数名引入当前符号表，函数名的值有一个能够被解释器视为用户定义的函数类型。该值可以赋值给其他名字，这样，这个名字亦能够当函数使用。这将表现为一个通常的重命名机制：

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

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

如果你从其他语言转入 Python，也许你会说 fib 顶多是一个过程而不是函数，因为它没有返回值。实际上，尽管非常令人讨厌，即便函数中没有 return 语句，它也会返回一个值。该值称为 None（它是一个内建的名字）。如果只有 None 可写出，解释器会阻止。如果你想用 print，你可以看到：

```
>>>fib(0)
>>>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:

写一个返回而不是打印出 Fibonacci 数列的函数非常简单：

```
>>>def fib2(n): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
>>>f100 = fib2(100)    # call it
>>>f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

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

和通常一样，这个例子也展现出一些 Python 的新特征：

- The return statement returns with a value from a function. return without an expression argument returns None. Falling off the end of a function also returns None.

- return 语句从一个函数中返回一个值。return 后面没有表达式时函数返回 None。函数的结尾也会返回 None。

- The statement result.append(a) 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*, see *Classes*) 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 + [a], but more efficient.

- result.append(a)语句调用了一个列表对象 result 的方法。方法是属于某个对象并且被命名为 obj.methodname 的函数，其中 obj 是某个对象（也许是一个表达式），而 methodname 则是对象类型中定义的函数。不同的类型定义不同的方法。不同类型的同名方法不会引起歧义。（使用 class 可以定义我们自己的类型和方法，查阅 *Classes*）append()方法为列表对象定义；它在列表的尾部添加对象。在该例中等价于 result = result + [a]，不过它效

率更高。

4.7. More on Defining Functions 深入函数定义

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

也可以使用很多不同的变量定义一个函数。有三种模式可以进行组合。

4.7.1. Default Argument Values 默认参数

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 to allow. For example:

最有用的模式是为一个或者多个参数指定默认值。这可以使用少于函数定义的参数个数的方式进行调用。比如：

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

This function can be called in several ways:

该函数可以采用多种方式进行调用：

- giving only the mandatory argument: ask_ok('Do you really want to quit?')
- giving one of the optional arguments: ask_ok('OK to overwrite the file?', 2)
- or even giving all arguments: ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')

This example also introduces the in keyword. This tests whether or not a sequence contains a certain value.

该例子引入了 in 关键字。它测试在某个序列中是否存在某个特定的值。

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

默认参数在函数定义时在定义域中进行指定，因此

```
i = 5

def f(arg=i):
    print arg

i = 6
f()
```

will print 5.

将打印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：默认参数只指定一次。这样在该值为一个可变对象，如 list,dictionary 或者绝大多数类的时候发生不同。比如，下面的函数计算前后传入参数的和：

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

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

This will print

它将打印

```
[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:

如果你不希望默认参数在前后的调用中共享，你可以按照如下方式写：

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

4.7.2. Keyword Arguments 关键字参数

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

函数亦可以采用 keyword = value 的形式进行调用，比如：

```
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:

可以采用如下的方式将进行调用：

```
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:

不过以下的调用并不合法：

```
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，实际参数不能够一次赋多个值。形式参数在调用时不能同时使用位置参数和关键字参数绑定值，这里有一个例子演示了在这种约束情况下失败的例子：

```
>>>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 (see *Mapping Types — dict*) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form *name (described in the next subsection) which receives a tuple containing the positional arguments beyond the formal parameter list. (*name must occur before **name.) For example, if we define a function like this:

当最后一个形参以**name 的形式出现的时候，它接收一个字典（参见 *Mapping Types — dict*），该字典接收除了形参之外的任何关键字参数。它可以和形参以*name（下一节中将进行描述）的形式结合起来，即接收一个除形参以外的参数列表。（*name 必须在**name 之前出现。）比如，如果我们定义了如下函数：

```
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:

可以进行如下调用：

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

and of course it would print:

其打印结果为：

```
-- 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，关键字名字列表的 sort()方法在打印 keywords 字典的内容之前就被调用；如果不这样进行，打印参数的顺序是未定义的。

4.7.3. Arbitrary Argument Lists 任意参数列表

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 (see *Tuples and Sequences*).

Before the variable number of arguments, zero or more normal arguments may occur.

最后，在函数中最少用的特征就是指定函数可以以任意参数进行调用。这些参数将被包装成一个元组(参阅 *Tuples and Sequences*)

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

4.7.4. Unpacking Argument Lists 取出 List 中的值作为参数

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:

一个相反的情况经常发生，即当参数已经在一个列表或者元组中的时候，需要将他们取出来形成函数的参数列表。比如，对于内置的 range()接收一个单独的 *start* 和 *stop* 参数。如果他们刚好没有被分开，使用*符号将它们拆分出来：

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

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

以同样的方式，可以使用**操作符将字典中的参数提取出来：

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

4.7.5. Lambda Forms Lambda 模式

By popular demand, a few features commonly found in functional programming languages like 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:

出于广泛的需求，一些在功能性编程语言如 Lisp 中发现的一些特性也被加入到 Python 中。使用关键字 lambda 能够创建一些小的匿名函数。这里有一个返回参数和的小函数：lambda a, b: a+b。Lambda 模式能够在任意需要函数对象的地方使用。他们在语法上被限制为单一的一个表达式。而语义上，他们只是一个正常函数的句法糖。和内嵌函数定义一样，lambda 模式能够引用局部域中的对象：

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

4.7.6. Documentation Strings 文档字符串

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

文档字符串已经有成熟的内容和格式。

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.

第一行应当是一个对对象目的简单而精确的描述。为了保持简洁，不应该在此说明对象的名字或者类型，因为有其他的方式可以进行表达（除非该名称刚好可以用来表示对象的功能）。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.

如果文档字符串中有多行，第二行应该为空，以便视觉上将概括与其他描述分开。余下的行应该以多章节的方式描述对象的调用，副作用等等。

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 解释器不会省去多行字符串中的缩进信息，所以处理文档的工具需要去按需处理这些缩进。采用如下方式的变化可以达到效果。第一行文字后的第一行非空格行决定整块文字的缩进量。（我们不能够使用第一行，因为第一行与字符串的引号在一起，其缩进效果不明显。）该缩进所需要的空格符将在其他每行进行缩进。少于缩进量的行不应该出现，如若出现，则该行前所有的空白符都应该省略。缩进所代表的空格符需要与空格符（经常是8个空格）进行测试。

Here is an example of a multi-line docstring:

如下是一个多行的文档字符串的例子：

```
>>>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.
```

4.8. Intermezzo: Coding Style 间奏曲：编码风格

Now that you are about to write longer, more complex pieces of Python, it is a good time to talk about *coding style*. Most languages can be written (or more concise, *formatted*) in different styles; some are more readable than others. Making it easy for others to read your code is always a good idea, and adopting a nice coding style helps tremendously for that.

你也许将要写更长更复杂的 Python 片段，那么现在正是说一下*编码风格*的时候了。大部分的语言都可以以不同的风格（更准确一点，是*被格式化为*）进行编写；不过一些比另外一些可读性更强。让别人更容易读懂你的代码总是一个好主意，而采用一个好的编码风格将极大的促进这种交流。

For Python, **PEP 8** has emerged as the style guide that most projects adhere to; it promotes a very readable and eye-pleasing coding style. Every Python developer should read it at some point; here are the most important points extracted for you:

对于 Python 来说，**PEP 8**已经制定了大部分工程都会遵守的编码风格；它致力于创造一种可读性强而且顺眼的编码风格。每个 Python 开发者都应该或多或少了解一些；这里将抽取几点最重要的：

Use 4-space indentation, and no tabs.

4 spaces are a good compromise between small indentation (allows greater nesting depth) and large indentation (easier to read). Tabs introduce confusion, and are best left out.

使用4个空格的缩进，而不要使用 tabs。

4个空格的缩进是小型缩进（允许更多的嵌套深度）和大型缩进（更加易读）的折中。Tabs 将引入混淆，最好不要使用。

Wrap lines so that they don’t exceed 79 characters.

This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.

每一行不要超过79个字符，否则新起一行。

这样可以使得屏幕小的显示器能够正常显示，也是的在大型显示器上能够同时显示几个文件。

Use blank lines to separate functions and classes, and larger blocks of code inside functions.

用空白行将函数、类和函数中大的代码块进行区分。

When possible, put comments on a line of their own.

在条件允许的情况下，将注释放在当前行上。

Use docstrings.

使用文档字符串。

Use spaces around operators and after commas, but not directly inside bracketing constructs: a = f(1, 2) + g(3, 4).

在操作符或者逗号后面使用空格，但是不要直接在括弧后使用：a = f(1, 2) + g(3, 4)。

Name your classes and functions consistently; the convention is to use CamelCase for classes and lower_case_with_underscores for functions and methods. Always use self as the name for the first method argument (see *A First Look at Classes* for more on classes and methods).

对类和函数的命名要一致，习惯是采用 CamelCase 作为类名，而 lower_case_with_underscores 作为函数名。在方法中总是使用 self 作为第一个参数（查看 *A First Look at Classes*）。

Don't use fancy encodings if your code is meant to be used in international environments. Plain ASCII works best in any case.

在国际化环境中不要使用其他的编码方式，ASCII 方式适用于任何方式。

5. Data Structures 数据结构

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

本章将更加详细的介绍一些你已经了解的东西，另外我们还会添加一些新的东西。

5.1. More on Lists 深入理解列表

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

列表数据类型还有其他的一些方法。如下是列表的所有方法：

list.append(*x*)

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

list.extend(*L*)

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

list.insert(*i*, *x*)

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).

list.remove(*x*)

Remove the first item from the list whose value is *x*. It is an error if there is no such item.

list.pop([*i*])

Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in 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.)

list.index(*x*)

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

list.count(*x*)

Return the number of times *x* appears in the list.

list.sort()

Sort the items of the list, in place.

list.reverse()

Reverse the elements of the list, in place.

An example that uses most of the list methods:

如下的例子用到了上述的大部分方法：

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

5.1.1. Using Lists as Stacks 将列表作为堆栈使用

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:

列表的方法可以让列表很轻易的变成一个堆栈来使用，而堆栈的特点就是最后一个被添加的数字最先被返回(“last-in, first-out”)。在堆栈的顶部添加一个元素，使用 append()方法，返回堆栈顶部的元素，使用没有显式指定索引的 pop()方法：

```
>>>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. Using Lists as Queues 将列表当做队列

It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first-out”); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).

同样也可以将列表作为队列使用，队列的特点是最先添加的数据最先返回(“first-in, first-out”)；不过，列表对此来说并不高效。将一个元素从尾部添加或者取出非常快，但是在列表的顶部添加或者取出就会变得较慢了（因为其他的元素都将产生移位）。

To implement a queue, use collections.deque which was designed to have fast appends and pops from both ends. For example:

可以用 collections.deque 去实现一个队列，它恰恰是为了快速从两端添加或者取出数据的。比如：

```
>>>from collections import deque
>>>queue = deque(["Eric", "John", "Michael"])
>>>queue.append("Terry")           # Terry arrives
>>>queue.append("Graham")          # Graham arrives
>>>queue.popleft()                # The first to arrive now leaves
'Eric'
>>>queue.popleft()                # The second to arrive now leaves
'John'
>>>queue                         # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3. Functional Programming Tools 函数化编程工具

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

有三个内置的函数在同列表一起使用的时候非常有用：filter(), map(), and reduce()。

filter(function, sequence) returns a sequence consisting of those items from the sequence for which function(item) is true. If *sequence* is a string or tuple, the result will be of the same type; otherwise, it is always a list. For example, to compute some primes:

filter(function, sequence)返回序列中 function(item)测试为真的所有元素的列表。如果 *sequence* 是一个 string 或者 tuple，会返回和它一样的类型，否则返回一个列表。

```
>>>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)对队列中的每个元素调用 function(item)函数，并且返回函数所有返回值的列表。比如，计算一些立方数：

```
>>>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:

可以传入多个队列；不过函数必须也要有和序列数(or None if some sequence is shorter than another)一样的形参数，每个队列对应一个形参。如下：

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

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

reduce(function, sequence)返回二参数函数 *function* 在元素上的值，其中，函数首先计算前两个元素的值，再将返回值与第三个元素计算，依次计算得到最后结果。比如，如下例子计算1到10的和：

```
>>>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.

如果序列中仅有一个元素，该元素的值被返回；如果序列为空，则会抛出异常。

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,

也可以传入第三个参数用以表示初始值。在这种情况下，如果序列为空，则返回该值，部位空的话，则最开始计算该值与第一个元素，然后他们的结果再和第二个数计算，等等。如下：

```
>>>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.

不要使用例子中 sum() 的定义：因为求和函数很常用，所以内建函数中已经有一个 sum(sequence)。

New in version 2.3.

5.1.4. List Comprehensions 列表推导式

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.

列表推导式提供了一种创建列表的简便方式，该方式无需使用 map()， filter() 和/或者 lambda。列表采用这种结构的定义通常比列表的建立过程要直观得多。每一个列表推导式由一个表达式，以及表达式后的一个 for 从句，然后0个或者多个 for 或者 if 从句组成。如果表达式想要表达为一个元组，必须用括号括起来。

```
>>>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 complex expressions and nested functions:

列表推导式比起 map()来说适应性更好，而且可以应用到复杂的表达式和内嵌函数：

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

5.1.5. Nested List Comprehensions 内嵌列表推导

If you've got the stomach for it, list comprehensions can be nested. They are a powerful tool but – like all powerful tools – they need to be used carefully, if at all.

Consider the following example of a 3x3 matrix held as a list containing three lists, one list per row:

如果这很对你的胃口，（我想说）列表推导式还能够被嵌套。这是非常强大的工具，不过和

许多其他强大工具一样，你也必须小心谨慎的使用。

考虑如下的例子，一个内含3个列表的列表所表示的3x3的矩阵，每个子列表表示一行：

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

Now, if you wanted to swap rows and columns, you could use a list comprehension:

现在，如果你想交换行和列，你可以使用列表推导式：

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

Special care has to be taken for the *nested* list comprehension:

对于*内嵌的*列表推导式你需要特别的留意：

To avoid apprehension when nesting list comprehensions, read from right to left.

为了消除嵌套列表推导式的忧虑，（我们）从右到左阅读。

A more verbose version of this snippet shows the flow explicitly:

下面是一段关于上述代码流程的详细表达：

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

In real world, you should prefer built-in functions to complex flow statements. The zip() function would do a great job for this use case:

在真实世界中，你应该更多的使用内建函数，而不是复杂的流程语句。zip()函数在这种情况下起了很大的作用：

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

See *Unpacking Argument Lists* for details on the asterisk in this line.

参阅 *Unpacking Argument Lists* 来获取更多关于本行中星号的细节。

5.2. The del statement 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 differs from the pop() method which returns a value. The del statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

这里提供了一种给出索引而不是值对列表中元素进行删除的方法：del 语句。与返回一个值的 pop()方法不同。del 语句也可以用于删除列表中的一个切片甚至是整个列表（早些章节里面我们提到的将一个空列表赋值给切片）。比如：

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

del can also be used to delete entire variables:

del 亦可以用来删除整个变量：

```
>>>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.

在此以后（直到它被赋值为另一个值）引用名字 a 将产生错误。我们将在后面探索更多 del 的用法。

5.3. Tuples and Sequences 元组和序列

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

我们可以看到，列表与 string 有诸多相同的地方，比如索引和切片操作。如下有两个*序列*的数据结构（参阅 *Sequence Types — str, unicode, 列表, tuple, buffer, xrange*）。因为 Python 是一门正在进化的语言，其他的序列数据结构也会被加入。一种标准的序列数据结构是 *tuple*。

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

元组由逗号分开的一组值组成，比如：

```
>>>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 always 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，在输出是，元组总是被括号括起来，这样，嵌套的 tuple 也可以被正确的解释；在输入时，可以使用或者不适用括号，不过括号一般情况下都是必须的（比如 tuple 是一个很大的表达式中的一部分）。

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.

元组有多种用途。比如：(x,y)坐标对，数据库中的雇员表等等。元组与 string 一样，也是不可以被修改的：你不可以对元组中的单个元素进行赋值（不过你可以使用切片以及一系列相关的操作来模拟这个效果）。也可以创建内含可变对象的元素，比如列表。

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:

一个比较特殊的问题是在创建一个只有0个或者1个元素的元组：为了符合这种创建，语法上会额外多出一些怪异的东西。空元组使用一堆空的括号来创建；一个元素的元组采用元素后加一个逗号（仅仅在用括号将单个元素括起来时不够的）的形式进行创建。虽然很丑陋，不过可以用起来了。比如：

```
>>>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!'是一个 *tuple packing* 的例子：12345, 54321 和 'hello!'被打包到一个元组中。逆向的操作也支持：

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

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

这被称之为 *sequence unpacking*，右边的序列为任意类型皆可。序列划分需要左边的变量数与右边的序列大小一致。值得一提的是，多重赋值实际上是元组打包和分块的结合。

5.4. Sets 集合

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 也有一种 *sets* 的数据结构。set 是一个未排序且元素唯一的集合。最基本的应用包括关系测试和忽略重复的入口。Set 对象还支持像联合、交叉、对比和对称对比等操作。

Here is a brief demonstration:

以下是一个简单的例子：

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

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

5.5. Dictionaries 字典

Another useful data type built into Python is the *dictionary* (see *Mapping Types — dict*). 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 index assignments, slice assignments, or methods like append() and extend().

另外一个非常有用的内建类型是 *dictionary*（参见 *Mapping Types — dict*）。在其他语言里面，字典通常作为关联内存或者关联数组出现。与序列被一个范围内的数字索引不同，字典使用 *keys* 进行索引，key 可以是任意的不可变类型；字符串和数字通常作为 key。tuple 也可以作为 key，如果它的元素都是字符串、数字或者 tuple；如果一个 tuple 直接或者间接的含有一个可变的元素，则不可以作为 key。你不能使用列表作为 key，因为列表可以使用索引、切片或者类似 append()和 extend()的方法进行修改。

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.

最好将字典想象为一个组未排序的 *key: value* 对，而且在一个字典中，每个 key 值唯一。一对花括号创建一个空的字典：{}。在其中放入一组用逗号分开的 key:value 对将初始化字典;这种方式也是字典打印的方式。

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.

字典中最主要的方法是将一个值以某个键存入或者以某个键取出该值。也可以使用 del 删除某个 key:value 对。如果你用一个已经存在的键存储某个值，该键之前的值将被清除。如果采用一个不存在的键取值将引起错误。

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

keys()方法以任意顺序（如果你想进行排序，只需要使用 sort()对键列表进行排序）返回字典

对象中所有的键列表。确认某个键是否在字典中存在，使用 in 关键字。

Here is a small example using a dictionary:

如下是一个使用字典的小例子：

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

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.

dict()构造函数直接从键-值列表中构件一个字典，其中键-值采用元组的形式存放在列表中。当这些来自一个模式，列表推导式可以非常紧凑的指定一个 key-value 列表。

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

Later in the tutorial, we will learn about Generator Expressions which are even better suited for the task of supplying key-values pairs to the dict() constructor.

在本指南的后续章节中，我们讲学习更好的为 dict()构造函数提供 key-value 对的 Generator Expressions。

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

当键的类型为字符串时，可以更容易的指定关键字参数：

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

5.6. Looping Techniques 循环

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

在对字典进行循环访问的时候，使用 iteritems()可以同时返回键和对应的值。

```
>>>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.

在对序列进行循环访问的时候，使用 enumerate()函数可以同时返回位置索引以及与之对应的值。

```
>>>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.

在对两个或者多个序列进行循环访问的时候，可以使用 zip()进行配对。

```
>>>questions = ['name', 'quest', 'favorite color']
```

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

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

希望对一个序列反向访问，首先指定序列的前进方向，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.

要向按顺序访问序列中的元素，使用 sorted()函数，该函数返回被排列好的序列，而原始序列不变。

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

5.7. More on Conditions 深入理解条件

The conditions used in while and if statements can contain any operators, not just comparisons.

在 while 和 if 中使用的条件语句可以包含任意的操作符，而不仅仅只是比较。

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.

比较操作符 in 和 not in 确认某个值是否存在序列中。is 和 is not 操作符比较两个对象是否为同一对象；只有列表之类的可变对象才会用到。所有的比较操作符有同样的优先级，而他们的优先级比所有的数值操作符有限级要低。

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

比较也可以形成链。比如，a < b == c 测试 a 是否小于 b 以及 b 是否等于 c。

Comparisons may be combined using the Boolean operators and and or, and the outcome of a comparison (or of any other Boolean expression) may be negated with not. These have lower priorities than comparison operators; 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. As always, parentheses can be used to express the desired composition.

比较也能够用布尔操作符 and 和 and or 组合起来，而比较的结果能够使用 not 取反。这些操作符比比较操作符优先级低，而 or 的优先级最低，所以 A and not B or C 相当于(A and (not B)) or C。通常情况下，括号能够用于表达想要的操作组合。

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. When used as a general value and not as a Boolean, the return value of a short-circuit operator is the last evaluated argument.

布尔操作符 and 和 or 被称之为 *short-circuit* 操作符：他们的参数解释顺序从左到右，一旦结果确定，就终止解释。比如，如果 A 和 C 为真，而 B 为假，A and B and C 中不会解释表达式 C 的真假。当它们不是布尔值而是用作一个通常意义的值，一个 short-circuit 的返回值是最后一个被解释的值。

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

可以讲比较或者布尔表达式的值赋值给一个变量。比如：

```
>>>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.

注意，在 python 中，与 C 不同，赋值不能出现在表达式中。C 程序员可能会对此抱怨，不过这阻止了 C 程序中的一个通常错误：在表达式中将==误写为=。

5.8. Comparing Sequences and Other Types 将序列与其他类型对比

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

序列能够和其他同类型的序列进行比较，比较采用*词典*顺序：首先两个序列中的第一个元素进行比较，如果他们不相同，则结果就已经被确定；如果他们相同，则再比较第二个元素，以此类推，知道任何一个序列访问完。如果两个比较的元素本身又是同类型的序列，则对它们进行递归的进行词典顺序比较。如果所有的元素比较都相同，这两个序列被认为相同。如果一个序列是另一个序列的子序列，较短的那个序列小。字典顺序中对于字符串的比较采用对每个字符按照 ASCII 中的顺序进行比较。一些例子如下：

```
(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.

注意，比较两个不同类型的对象是合法的。比较结果确定却随意：类型以他们的名字排序。这样，一个列表总是比一个 string 小，而一个 string 总是比一个元组小等等。混合数值类型按照数值的大小进行比较，所以0与0.0是相等的，等等。

6. Modules 模块

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. More on Modules 深入模块

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。

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。被导入的模块名会置入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:

甚至有种方式可以导入模块中的所有定义：

```
>>>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。

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.

值得注意的是，一般情况下，在实践中并不提倡使用 importing *的形式，这样会导致代码的可读性很差。然而，在交互式环境中，这样可以省去许多打字的功夫。

Note

标注

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

考虑到效率原因，每一个模块在每一次解释器会话中只导入一次。因此，如果你改变了你的模块，你必须重新启动解释器，或者如果你仅仅只是要交互测试一下一个模块，使用 reload(), 例如 reload(modulename)。

6.1.1. Executing modules as scripts 将模块作为脚本使用

When you run a Python module with

当你运行一个 Python 模块采用

python fibo.py <arguments>

the code in the module will be executed, just as if you imported it, but with the __name__ set to "__main__". That means that by adding this code at the end of your module:

模块中的代码将被执行，就像你导入了它一样，不过它的__name__被设置为"__main__"。这意味着，在模块的尾部加入以下代码：

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

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the “main” file:

你可以建一个文件作为脚本或者可导入模块，因为仅仅当模块以“main”文件执行时，才会解析命令行：

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

If the module is imported, the code is not run:

如果模块被导入，代码不会被执行：

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

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

这种情况要么是为了给模块提供一个方便的用户接口，要么是为了测试（将一个模块作为脚本执行，即执行一个测试用例）。

6.1.2. The Module Search Path 模块搜索路径

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 *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*标准模块*以了解更多的信息。

6.1.3. “Compiled” Python files "编译"Python 文件

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

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:

对于专家的一些建议：

When the Python interpreter is invoked with the *-O* flag, optimized code is generated and stored in .pyo files. The optimizer currently doesn't help much; it only removes assert statements. When *-O* is used, *all bytecode* is optimized; .pyc files are ignored and .py files are compiled to optimized bytecode.

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* 参数，*所有的 bytecode* 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，然后用一个小的启动脚本导入这个模块，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'一译者)，可以只有 spam.pyc 文件(或者 spam.pyo，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 模块能够创建一个文件夹中所有模块的.pyc 文件（或者是在使用了*-O*时的.pyo 文件）。

6.2. Standard Modules 标准模块

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 winreg module is only provided on Windows systems. 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，winreg 模块只只提供于 Widnows 操作系统。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定义了主提示符和辅助提示符字符串:

```
>>>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.

这两个变量只有在解释器处于交互模式时才被定义。

The variable sys.path is a list of strings that determines 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. The dir() Function dir()函数

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()在找出模块定义的名字而使用的。它返回一个字符串列表：

```
>>>import fibo, sys
>>>dir(fibo)
['__name__', 'fib', 'fib2']
>>>dir(sys)
['__displayhook__',   '__doc__',   '__excepthook__',   '__name__',
'__stderr__',
 '__stdin__', '__stdout__', '_getframe', 'api_version', 'argv',
 'builtin_module_names', 'byteorder', 'callstats', 'copyright',
 'displayhook', 'exc_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:

不带参数的情况下， dir() 列出当前你定义的名字：

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

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

它列出了所有类型的名字：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()不会列出内置函数名和变量名。如果你想要这些名字，他们定义在标准模块__builtin__中：

```
>>>import __builtin__
>>>dir(__builtin__)
['ArithmeticError',       'AssertionError',       'AttributeError',
'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FloatingPointError', 'FutureWarning', 'IOError', 'ImportError',
 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
 'LookupError',       'MemoryError',       'NameError',       'None',
'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError',
 'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError',
 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True',
 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
 'UserWarning', 'ValueError', 'Warning', 'WindowsError',
 'ZeroDivisionError', '_', '__debug__', '__doc__', '__import__',
 '__name__', 'abs', 'apply', 'basestring', 'bool', 'buffer',
 'callable', 'chr', 'classmethod', 'cmp', 'coerce', 'compile',
 'complex',  'copyright',  'credits',  'delattr',  'dict',  'dir',
'divmod',
 'enumerate', 'eval', 'execfile', 'exit', 'file', 'filter', 'float',
 'frozenset', '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', 'reversed', 'round', 'set',
 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super',
 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

6.4. Packages 包

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“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 A 的包中名为 B 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.

当导入这些包的时候，Python 在 sys.path 中的目录下查找包的子目录。

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

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__量，后面会有相关介绍。

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

包的用户可以从包中导入独立的模块，如：

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

This loads the submodule sound.effects.echo. It must be referenced with its full name.

这将加载子模块 sound.effects.echo。必须用全名进行引用。

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

An alternative way of importing the submodule is:

替换方式有：

```
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:

这样也加载了子模块 echo，但是不需要使用包前缀就可使用，故而可以这样使用：

```
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. 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. This could take a long time and importing sub-modules might have unwanted side-effects that should only happen when the sub-module is explicitly imported.

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。这将花费很长时间，而且在导入子模块时会产生一些只有在显式导入时有的副作用。

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 any initialization code in __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)。

Although certain modules are designed to export only names that follow certain patterns when you use import *, it is still considered bad practise in production code.

尽管在使用 import *时一些模块被设计为为了某种模式而只导出名字，在生成代码的时候仍然是一个坏主意。

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. Intra-package References 内置包参考

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

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), you can use absolute imports to refer to submodules of siblings packages. 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 echo 模块，它可以使用 from sound.effects import echo。

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

从 Python2.5开始，除了上述隐式导入的描述，你还可以使用 from module import name 写显式导入的形式。显式导入使用了前置点号来表示当前包以及父包。在使用 surround 模块的例子中，你可以这样使用：

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

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

注意显式导入和隐式导入都是基于当前模块的名字。由于主模块的名字总是"__main__"，在主模块中使用其他模块时要使用绝对导入。

6.4.3. Packages in Multiple Directories 多重路径中的包

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.

包支持一个更为特殊的变量__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。

7. Input and Output 输入与输出

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. Fancier Output Formatting 设计输出格式

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

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 layout 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 str.format() method.

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，使用字符切割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 str.format()方法。

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.

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。

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:

一些例子：

```
>>>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'))"
```

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

有两种方式输出平方和立方表：

```
>>>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 '{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

(Note that in the first example, one space between each column was added by the way print works: 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，可以使用切割操作，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:

另外一个方法 zfill()用于在数值字符串的左边填充0。它能够理解加号与减号：

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

Basic usage of the str.format() method looks like this:

str.format()方法的基本应用如下：

```
>>>print 'We are the {0} who say "{1}!"'.format('knights', 'Ni')
We are the knights who say "Ni!"
```

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

其中的括号和字符（称之为符号域）被传入到 format()方法的对象代替。括号中的一个数字代表传入 format()方法中对象的位置。

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

If keyword arguments are used in the format() method, their values are referred to by using the name of the argument.

如果在 format()中使用了关键字参数，它们的值使用参数的名字。

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

Positional and keyword arguments can be arbitrarily combined:

位置与关键字参数能够任意的组合：

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

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

'!s'（应用 str()）和'!r'（应用 repr()）可以用作格式前的值转换。

```
>>>import math
>>>print 'The value of PI is approximately {0}.'.format(math.pi)
The value of PI is approximately 3.14159265359.
>>>print 'The value of PI is approximately {0!r}.'.format(math.pi)
The value of PI is approximately 3.141592653589793.
```

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

一种可选的':'和格式化分类符可以加在域名字后面。这能够使得格式化更加有控制力。下面的例子将 PI 转化为小数点后三位的字符串。。

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

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

在':'后传入一个整数将会导致该域成为最小字符宽度。这可以使表格看起来很漂亮。

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

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 simply passing the dict and using square brackets '[]' to access the keys

如果你真的有一个不想被拆开但是很长的字符串要格式化，最好是可以引用的变量用名字而不是位置来格式化。可以简单的通过传入字典并通过'[]'来访问键值。

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

This could also be done by passing the table as keyword arguments with the '**' notation.

也可以使用'**'将表格作为关键字参数传入。

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

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

这在于新内置函数 vars()一起使用时特别有用，该函数返回一个包含所有局部变量的字典。

For a complete overview of string formatting with str.format(), see *Format String Syntax*.

请参阅 *Format String Syntax* 来获取用 str.format()格式化字符串的完整描述。

7.1.1. Old string formatting 旧式字符串格式化

The % operator can also be used for string formatting. It 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. For example:

%操作符亦可以用来格式化字符串。它像 sprintf()样式一样解释左参数，并应用右参数，返回经过格式化后的字符串。如下：

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

Since str.format() is quite new, a lot of Python code still uses the % operator. However, because this old style of formatting will eventually be removed from the language, str.format() should generally be used.

str.format()比较新，而许多老式的代码依旧使用%。然而，由于这种旧式的格式化将被淘汰，所以平时最好使用 str.format()。

More information can be found in the *String Formatting Operations* section.

可以在 *String Formatting Operations* 中找到更多信息。

7.2. Reading and Writing Files 读写文件

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

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。在 Unix 系统中，添加'b'并不会损坏文件，所以可以平台独立的使用任何二进制文件。

7.2.1. Methods of File Objects 文件方法对象

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，从中返回多行文本。这个功能通常用于高效读取大型行文件，避免了将整个文件读入内存。这种操作只返回完整的行。

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

An alternative approach to reading lines is to loop over the file object. This is memory efficient, fast, and leads to simpler code:

交换通道可以循环读取文件对象中的行。这是内存操作的效率，快速，代码简单：

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

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

The alternative approach is simpler but does not provide as fine-grained control. Since the two approaches manage line buffering differently, they should not be mixed.

交换通道很简单，但是不提供完整的控制。因为两个通道管理线缓冲不同，它们不能混合。

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')
```

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

如果需要写入字符串以外的数据，就要先把这些数据转换为字符串。

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

f.tell() 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的话，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
```

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

在使用文件对象的时候最好使用 with 关键字。这样的好处是在文件处理完后，文件就会被关闭，即使是有异常产生。比起书写相应的 try-finally 块也简单得多。

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

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. The pickle Module pickle 模块

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 代码段!)表达为字符串，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 in the Python Library Reference。)

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。

8. Errors and Exceptions 错误和异常

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. Syntax Errors 语法错误

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. Exceptions 异常

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 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 provides detail based on the type of exception and what caused it.

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 traceback. In general it contains a stack traceback 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。

Built-in Exceptions lists the built-in exceptions and their meanings.

Built-in Exceptions 列出了内置异常和它们的含义

8.3. Handling Exceptions 处理异常

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 工作如下。

- 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 clause* 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 except clause is

executed, and then execution continues after the try statement.

- 如果在七 ry 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 关健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(*unhandled exception*)，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 tuple, 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 as (errno, strerror):
    print "I/O error({0}): {1}".format(errno, strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

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 clause*，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 tuple). The variable is bound to an exception instance with the arguments stored in instance.args. For convenience, the exception instance defines __str__() so the arguments can be printed directly without having to reference .args.

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

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

程序员也可以在异常抛出前实例化它，并为它添加一些所需的属性。

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

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:

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

```
>>>def this_fails():
...    x = 1/0
...
>>>try:
```

```
...    this_fails()
...except ZeroDivisionError as detail:
...    print 'Handling run-time error:', detail
...
Handling run-time error: integer division or modulo by zero
```

8.4. Raising Exceptions 抛出异常

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

程序员可以用 raise 语句强制指定的异常发生例如：

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

The argument to raise is an exception class or instance to be raised. There is a deprecated alternate syntax that separates class and constructor arguments; the above could be written as raise NameError, 'HiThere'. Since it once was the only one available, the latter form is prevalent in older code.

raise 的参数是一个需要抛出的异常类或者实例。还有一种不建议使用的替代写法是 raise NameError, 'HiThere'。由于这曾经是唯一可用的，所有后者的形式普遍存在于旧式代码中。

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. User-defined Exceptions 用户自定义异常

Programs may name their own exceptions by creating a new exception class (see *Classes* for more about Python classes). 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(在 *Classes* 查看更多关于 Python 类的描述)来命名自己的异常。异常类通常应该直接或间接的从 Exception 类派生，例如：

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

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

在这个例子中，Exception 默认的__init__()被覆盖。新的方式简单的创建 *value* 属性这就替换了原来创建 *args* 属性的方式。

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 that 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:
        expr -- input expression in which the error occurred
        msg  -- explanation of the error
    """

    def __init__(self, expr, msg):
        self.expr = expr
        self.msg = msg

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

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

    def __init__(self, prev, next, msg):
        self.prev = prev
        self.next = next
        self.msg = msg
```

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 *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 *Classes*

8.6. Defining Clean-up Actions 定义清理行为

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!
KeyboardInterrupt
```

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

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

```
>>>def divide(x, y):
...    try:
...        result = x / y
...    except ZeroDivisionError:
...        print "division by zero!"
...    else:
...        print "result is", result
...    finally:
...        print "executing finally clause"
...
>>>divide(2, 1)
result is 2
executing finally clause
>>>divide(2, 0)
division by zero!
executing finally clause
>>>divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

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

如你所见，finally 子句在任何情况下都会执行。TypeError 在两个字符串相除的时候抛出，未被 except 子句捕获，因此在 finally 子句执行完毕后重新抛出。

In real world applications, the finally clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

在实际的应用程序中，finally 子句用于释放外部资源(例如文件或网络连接)，无论资源的使用是否成功。

8.7. Predefined Clean-up Actions 预定义清理行为

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen.

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

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

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

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

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

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

语句执行后，文件 *f* 总会被关闭，即使是在处理文件中的数据时出错也一样。其它对象是否提供了预定义的清理行为要查看它们的文档。

9. Classes 类

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, and a method can call the method of a base class with the same name. Objects can contain an arbitrary amount of data.

Python 的类机制使用了极少量的新语法将类添加到该语言中。它是 C++和 Modula-3中类机制的混合。Python 的类并不将用户和定义分开，而是依赖于用户不会打乱类的定义，对于模块来说是合适的。类的最重要的特性依旧保持全部功能，然而：类继承允许多重继承，一个子类可以重载父类的所有方法，一个方法可以使用同样的名字调用父类的方法。对象可以包含任意数量的数据。

In C++ terminology, all class members (including the data members) are *public*, and all member functions are *virtual*. 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. 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++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

用 C++术语来说，Python 所有类成员（包括数据成员）都是 *public* 的，所有的成员函数都是 *virtual* 的。在 Modula-3中，没有在方法中引用成员的捷径：成员函数的定义中，第一个参数显式的指向当前对象，当调用的时候则会隐式提供。在 Smalltalk 中，类本身就是对象。这样提供了导入与重命名的寓意。与 C++和 Modula-3不同，Python 内建类型可以作为用于扩展的基类。同样，与 C++一样，所有的内建操作符都已用特殊的语法（算数运算符、下标符等等）进行重新定义。

(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.)

（由于缺乏一种普遍的术语描述类，我将经常使用 Smalltalk 与 C++中的特性。我也会使用 Modula-3的特性，因为它的面向对象语法比起 C++来与 Python 更加接近，不过我预计很少有读者听说过。）

9.1. A Word About Names and Objects 名字与对象的简单描述

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 a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. 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.

对象拥有特性，多个名字（在多个作用域）可以被绑定到一个对象上面。在其他语言中称之为别名。在最初使用 Python 的时候最好不要这么做，在处理不可变的基本类型（数字、字符串、元组）是可以安全的忽略。然而，在可变对象如 list、字典以及大部分其他类型中别名会引起不可思议的效果。这样使用对于编程有好处，因为别名更像是一个指针。比如，传递一个对象代价很低，以为在实现上仅仅只是传递了一个指针；如果某个函数修改了从参数中传入的对象，调用者就会看到变化。这淘汰了 Pascal 中需要传递两个不同参数的做法。

9.2. Python Scopes and Namespaces 作用域和名字空间

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，我首先要向你介绍关于 python 作用域的规则。类的定义对名字空间玩弄了一些小技巧，为了全面了解所发生的事情，你必须了解作用域与名字空间的工作方式。顺便提一句，该话题对于任何一个进阶的 Python 程序员都有用。

Let’s begin with some definitions.

让我们以一些定义开始。

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.

一个*名字空间（namespace）*是对象到名字的映射。当前绝大部分的名字空间使用 Python 的字典实现，不过这在任何情况下（效率除外）都不明显，在将来也许会改变这种方式。名字空间的例子有：内建的名字集（abs()类似的函数，和一些内建的异常名字）；模块的全局名字；在函数内部调用的局部名字。在某种意义上，对象的属性集也构成了一个名字空间。了解名字空间最重要的一点是，在不同名字空间中名字没有任何联系；比如，两个不同的模块可以同时定义一个 maximize 函数而不相互冲突；模块的使用者使用该函数时必须在前面加上模块名称。

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]

顺便说一下，我使用*属性*一词表示所有点号后面的名字；比如，在表达式 z.real 中，real 是对象 z 的属性。严格的说，在模块中，对名字的引用就是属性引用：在表达式 modname.funcname 中，modname 是一个模块对象，funcname 则是该对象的一个属性，在这种情况下，这里恰好有一个模块属性与模块中定义的全局名字的直接映射：它们共享同一个名字空间！[1]

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.

属性可以是只读的或者可写的。在后一种情况下，对属性的赋值是可能的。模块的属性是可写的：你可以写 modname.the_answer = 42。可写的属性也可以使用 del 语句删除。比如，del modname.the_answer 将 the_answer 从名字为 modname 的对象中移除。

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

名字空间在不同的时刻创建，也拥有不同的生命周期。包含内建名字的名字空间是在 Python 解释器启动的时候创建的，至此不会被删除。模块的全局名字空间是当模块的定义被读入的时候创建的；一般情况下，模块名字空间一直持续到解释器退出。被解释器顶层调用的语句——要么来自于一个脚本文件或者来自交互模式，被称之为__main__模块的一部分，所以他们也拥有自己的全局名字空间。（内建的名字也存活于一个模块；被称之为__builtin__模块）。

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.

函数被调用时，函数的局部名字空间被创建，函数返回或者引起一个没有处理的异常时，该名字空间被销毁。（实际上，忘记是描述所发生之事的更好方法。）当然，递归调用每次都会有自己的局部名字空间。

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.

一个*作用域*（*scope*）是指 Python 程序中可以直接使用名字空间的文本区域。直接使用，在这里指对某个名字的引用将在该名字空间中查找改名字。

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:

尽管作用域是静态决定的，它们在使用是时候是动态的。在执行的任意时刻，最少有三个嵌套的作用域中的名字是可以直接使用的：

- the innermost scope, which is searched first, contains the local names

- 最深处的作用域会被最先查找，改作用域含有局部名字

- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains non-local, but also non-global names

- 函数的作用域，查找与它最近的作用域，该作用域包含非局部，但是也非全局的名字

- the next-to-last scope contains the current module’s global names

- 倒数第二个作用域包含当前模块的全局名字

- the outermost scope (searched last) is the namespace containing built-in names

最外面的名字空间（最后一个被查找）包含内建名字

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

如果一个名字声明为全局，那么所有的引用与赋值直接进入包含模块全局名字的中间作用域。否则，在最深处的作用域外发现的变量都是只读的（任何试图对该变量的写入将简单的在最深处名字空间创建一个*新的*局部变量，而外部的变量不改变。）

Usually, the local scope references the local names of the (textually) current function. Outside 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.

通常情况下，局部作用域引用当前函数的局部名字。在函数之外，局部作用域引用全局作用域相同的名字空间：模块的名字空间。类定义在局部作用域内又引入了一个名字空间。

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 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.)

意识到作用域是以文本所在处决定的：在模块中定义的一个函数的全局作用域是模块的名字空间，不论函数从哪里调用或者采用什么别名被调用。另外一方面，真正的运行时名字查找是动态进行的；然而，这门语言正朝着在编译期间的静态名字解决方案演进，所以，它们并不依赖动态名字解决方案！（实际上，局部变量已经是静态决定的。）

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

Python 有一个怪癖，即如果没有 global 语句，对名字的赋值将在最里面的作用域中进行。赋值不会拷贝数据，它们只是将名字绑定到对象。删除也遵从该规则：del x 语句移除名字

空间中对象与 x 名字的绑定。实际上，所有的操作都将引入一个在局部作用域中的新名字:特别的，import 语句和函数定义在局部作用域中绑定模块和函数名字。（global 语句指出某一些特殊的变量在全局作用域中存活。）

9.3. A First Look at Classes 类之初识

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

类引入了一些新的语法，三个新的类型以及一些新的语义。

9.3.1.   Class Definition Syntax 类定义语法

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.)

与函数定义一样，类定义（def 语句）必须在定义的最开始。（你可以想当然的将一个类的定义置入 if 语句的一个分支或者某个函数中。）

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.

在实践中，类定义中的语句通常都是函数定义，不过其他的一些语句也被允许，一些时候还会很有用——我们之后会来说明这些。类中的函数定义通常有独特的参数列表，这样是为了调用方法的需要——同样的，我们之后会解释这些。

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.

当输入了一个类定义，一个新的名字空间被创建，并作为局部域使用——这样，所有对局部变量的赋值都将进入这个新的名字空间。特别的，函数定义为新的函数绑定了名字。

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 definition 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).

当一个类定义完成，一个*类对象*就被创建。它是新创建的名字空间的周围内容的包装；我们将在下一章节中介绍更多的类对象的内容。最初的局部域（在输入类定义之前的局部域）被回复，而且该类对象在此处被绑定为类定义头上指定的名字。

9.3.2. Class Objects 类对象

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

类对象支持两种操作：属性引用与实例化。

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*使用了 Python 中引用的标准语法：obj.name。类对象创建时，类的名字空间中的所有名字都是合法的属性名字。所以，如果类定义如下：

```
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 function 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 都是合法的属性引用，分别返回一个整型和函数对象。类属性也可以被赋值，因此你可以通过赋值改变 MyClass.i 的值。__doc__也是一个合法的属性，该属性返回该类的文档字符串："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*实例化*使用了函数签名方式。只需认为类对象是一个无参的返回一个类实例的函数。比如（使用上面的类）：

```
x = MyClass()
```

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

创建了一个该类的新*实例*，且将该对象赋值给本地变量 x。

The instantiation operation (“calling” a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named __init__(), like this:

该实例化操作（“称之为”一个类对象）创建了一个空的对象。许多类都希望创建能够被初始化的对象实例。因此，一个类可以定义个一个特殊的方法称为__init__()，就像这样：

```
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:

当一个类定义了一个__init__()方法的时候，类实例化自动为新创建的实例调用__init__()方法。所以，在下面的例子中，一个新的，且被初始化的对象被创建：

```
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，__init__()可以有非常灵活的参数。在上例中，置入实例化操作的参数被传递到__init__()方法中。比如，

```
>>>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. Instance Objects 实例对象

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

那么现在我们如何处理该对象实例呢？实例对象能够接受的唯一操作符就是属性引用。有两种合法的参数名字，数据属性和方法。

data attributes 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:

*数据属性*对应于 Smalltalk 中的“实例化变量”或者 C++中的“数据成员”。数据属性无需声明；和局部变量一样，他们在首次被赋值的时候就被创建了。比如，如果 x 是 MyClass 的实例，如下的代码片段将打印16而不会留下痕迹：

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

The other kind of instance attribute reference is a *method*. 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, in the following discussion, we’ll use the term method exclusively to mean methods of class instance objects, unless explicitly stated otherwise.)

另外一种实力属性引用是*方法*。一个方式是指“属于”某个对象的函数。（在 Python 中，方法并不是类实例才有：其他的对象类型也拥有方法。比如，list 对象有 append,insert,remove,sort 等等方法。然而，在下面的讨论中，我们在不特别说明的情况下，笼统的称方法为类对象实

例的方法。）

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are 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.

合法的方法名字来自于类。在定义中，所有的函数属性对应于对象的某个方法。所以，在我们的例子中，x.f 是该实例的一个合法方法引用，因为 MyClass.f 是一个函数，然而 x.i 不是，因为 MyClass.i 不是函数。不过 x.f 与 MyClass.f 并不是一码事——它是一个*方法对象*，而不是一个函数对象。

9.3.4. Method Objects 方法对象

Usually, a method is called right after it is bound:

通常情况下，一旦绑定，该方法就被调用：

```
x.f()
```

In the MyClass 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 MyClass 例子中，将返回字符串'hello world'。然而，没有必要立刻调用该方法：x.f 是一个方法对象，可以被存储起来，以便于以后使用。比如：

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

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

将持续输出 hello world 直到时间终止。

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...

当一个方法被调用的时候，什么事情发生了呢？你也许会注意到 x.f()的调用并没有如上的参数，尽管 f()的定义指定了一个参数。这个参数发生了什么呢？可以肯定的是，当一个需要参数的函数调用时没有传入参数——即便该参数并不会使用时，Python 会触发一个异常...

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.

事实上，你也许已经猜到了答案：关于方法的特殊之处在于对象本身被作为第一个参数传入该函数。在我们的例子中，x.f()的调用与 MyClass.f(x)几乎相同。大部分情况下，调用一个有 *n* 个参数的方法与使用插入对象本身作为第一个参数的参数列表调用对应的函数调用是一致的。

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, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.

如果你仍然不知道方法怎么运作，对实现的观察也许会解除你的疑惑。当一个不是数据属性的实例属性被引用的时候，该类（的实现）被搜索。如果该名字代表一个合法的类属性，且对应一个函数对象，一个方法对象被创建，该对象将类对象与函数对象绑定（指向）起来，从而形成一个抽象的对象，这就是方法对象。当该方法对象使用一个参数列表调用的时候，一个新的参数列表将从该参数列表与对象实例创建出来，然后改函数对象使用新的参数列表。

9.4. Random Remarks 一些说明

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.

数据属性重写了方法属性；为了防止偶尔的名字冲突（这将造成一些在大的程序里很难查找的错误），最好使用某种变换使得冲突的可能性最小。可用的变换包括对方法名字首字母进行大写，在数据属性名字前面独特的字符串（也许就是一个下划线），或者对方法的名字使

用一些动词形式而数据属性使用名词形式。

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.)

数据属性会被方法引用，和普通的用户引用它们一样。也就是说，类不是用来创建纯抽象数据类型。实际上，在 Python 中没有方法实现数据隐藏——这依赖于变换。（另外一方面，如果需要的话，用 C 实现的 Python 能够完全隐藏实现细节和对象的存取控制；这可以作为 Python 的 C 扩展。）

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.

客户应该谨慎使用数据属性——客户可能会陷入方法属性与数据属性的隐藏泥潭。注意，客户可以添加自己的数据属性到对象实例上而不影响方法的正确性，直到名字冲突被阻止，在此处，一个名字转换可以省很多脑筋。

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.

从方法中引用数据属性没有捷径。我发现这样实际上增加了方法的可读性：在查看方法的代码时，没有任何机会搞混局部变量和实例变量。

Often, the first argument of a method is 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 to other Python programmers, and it is also conceivable that a *class browser* program might be written that relies upon such a convention.

通常情况下，方法的第一个参数被称为 self。这里不需要更多的转换：self 对 Python 而言没有任何特别的地方。然而，值得注意的是，不遵循这种形式，你的代码对别的 Python 程序员来说将会变得可读性下降，而且可以想象得到，一个*类浏览*程序可能按照这种形式来查找类。

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:

任何一个类属性的函数对象都为其实例定义了一个方法。并不需要在类定义的代码中定义函数：将一个函数对象赋值给一个类内部的局部变量也是可行的。比如：

```
# 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.

这样，f，g 和 h 都是类的函数对象属性，也就是说，他们都是 C 实例的方法；h 与 g 几乎一样。值得注意的是，在实践中，这种方法只会让读者混淆。

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

方法可以使用 self 的方法属性调用其他方法：

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

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.

方法也可以与普通的函数一样引用全局名字。该方法的全局域是包含类定义的模块。（类本身从来就不会作为全局域。）然而我们很少遇到在方法中使用全局数据的好借口，有很多合理的使用全局域的方法：一则在全局域中导入的函数和模块可以被方法使用，就如在其中定义的函数或者类一样。通常情况下，包含方法的类本身就定义在全局域中，在下一而章节中，我们会找到一些为什么方法会调用类本身的理由。

Each value is an object, and therefore has a *class* (also called its *type*). It is stored as object.__class__.

每一个值都是一个对象，因此他们拥有一个类（也被称之为*类型*）。它以 object.__class__的方式存储。

9.5. Inheritance 继承

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

当然，如果一门语言不支持继承特性，也枉含有"class"这个名字。派生类的语法定义如下：

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

The name BaseClassName must be defined in a scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

名字 BaseClassName 必须定义在一个包含派生类的定义域中。。除了基类名字，其他的一些表达式也可以使用。这非常有用，比如，当基类定义在另外一个模块的时候：

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, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.

派生类定义与基类的执行过程一样。当类对象被构造，基类也被构建出来。这是为了解决属性引用：如果某个请求的属性在类定义中没有找到，那么查找将在基类中继续进行。当该类

的基类本身又是从其他类继承而来，该规则就会递归执行。

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.

派生类的实例化并没有什么特别的地方：DerivedClassName()构造了该类的一个新实例。方法引用的解决方案如下：当前类的属性被查找，并且沿着类继承链向下查找，知道出现一个函数对象的时候，该方法引用就变得有效。

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 end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively virtual.)

派生类可以重载基类的方法。因为调用同一个对象的方法没有优先级，一个基类的方法调用另外一个定义在同一个类中的方法会调用一个在派生类中重载了的方法。（对于 C++程序员来说：Python 中所有的方法都是 virtual 的。）

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 accessible as BaseClassName in the global scope.)

派生类中的某个方法可能实际上是想扩展而不是简单的替代基类中具有相同名字的方法。有一个简单的方法来直接调用基类的方法：只需要调用 BaseClassName.methodname(self, arguments)。这对客户代码来说也是有用的。（值得注意的是这只有当 BaseClassName 在全局作用域中可见的时候。）

Python has two built-in functions that work with inheritance:

Python 有两个关于继承的内建函数：

- Use isinstance() to check an instance’s type: isinstance(obj, int) will be True only if obj.__class__ is int or some class derived from int.

- 使用 isinstance() 来检测一个实例的类型：isinstance(obj, int) 为 True 当且仅当 obj.__class__是 int 或者一些派生自 int 的类。

- Use issubclass() to check class inheritance: issubclass(bool, int) is True since bool is a subclass of int. However, issubclass(unicode, str) is False since unicode is not a subclass of str (they only share a common ancestor, basestring).

- 使用 issubclass()来检查类继承：issubclass(bool, int)为 True 因为 bool 是 int 的子类。然而，issubclass(unicode, str)为假因为 unicode 不是 str 的子类（他们仅仅是共享一个相同的基类——basestring）。

9.5.1. Multiple Inheritance 多重继承

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

Python 也支持有限的多重继承。一个拥有多重继承的类定义如下：

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

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

对于旧式的类，唯一的规则是深度优先，从左到右。这样，如果某个属性没有在 DerivedClassName 中发现，首先在 Base1中查找，然后（递归的）在 Base1的基类中查找，如果没有找到，则以此方式继续在 Base2中查找。

(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.)

（对于某些人来说，广度优先——查找 Base2和 Base3应该先于 Base1的基类——看上去似乎更合理。然而，这要求你在指出与 Base2的属性有名字冲突的情况之前能够知道 Base1的某个特定属性是否真的定义 Base1中或者其基类中。优先查找规则对于 Base1的直接或者继承属性没有区别。）

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

对*新样式类*，该方法超找解决方案顺序动态变化，用以支持联合调用 super()。这种方法在其他的多继承语言中被称为调用下一个方法，而且比在单继承语言中的 super 调用更加有用。

With new-style classes, dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where one at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all new-style classes inherit from object, so any case of multiple inheritance provides more than one path to reach object. To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance. For more detail, see http://www.python.org/download/releases/2.3/mro/.

在新样式的类中，动态顺序非常必要，因为所有的多继承情况都展现一个或者多个钻石关系（在最底层类中至少有一个父类能够通过多种路径被引用）。比如，所有的新样式类从 object 类继承，所以任何一种多继承情况都提供了多余一种的方法触及 object。为了保证基类被多次触及，动态算法对坚持从左至右的查找顺序进行了线性规划，仅仅调用父类一次，而且那是稳定的（意思是说一个类能够被作为子类而不影响父类的优先顺序）。把这些都考虑进来，这些属性能够让我们设计稳定且可扩展的多继承类。更多的细节请查看 http://www.python.org/download/releases/2.3/mro/。

9.6. Private Variables 私有变量

“Private” instance variables that cannot be accessed except from inside an object, don’t exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. _spam) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.

只能够在一个对象内部存取的“私有”实例变量在 Python 中是不存在的。然而，绝大部分的 Python 代码都会遵循这种替代：一个以下划线开头的变量（比如 _spam），应当作为一个非公开的 API（不管它是函数、方法或者数据成员）。它应该被认为是一个实现细节，且倾向于在未通知的情况下改变。

Since there is a valid use-case for class-private members (namely to avoid name clashes of names with names defined by subclasses), there is limited support for such a mechanism, called *name mangling*. Any identifier of the form __spam (at least two leading underscores, at most one trailing underscore) is textually replaced with _classname__spam, where classname is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

因为有一个有效的类私有成员用户用例（即防止与子类中的名字冲突），Python 对这种机制的支持有限，称之为*名字压延*。任何__spam（至少两个前置的下划线，至多一个后置下划线）的形式标识符被文本上置换为_classname__spam，其中 classname 是当前类的名字，而且会前置下划线。名字压延不对标识符的语法位置敏感，只要它出现在类定义中。

Note that the mangling rules are designed mostly to avoid accidents; it still is possible to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger.

值得注意的是名字压延规则是为了防止意外而设计的；仍然可以去修改一个被认为是私有的变量。这在一些特殊的情境下也有用，比如调试。

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

注意到，传入 exec，eval()或者 execfile()的代码并不考虑调用的类的名字作为当前类（名字）；这和 global 语句的效果是一样的，该效果同样限制编译成字节的代码。getattr()，setattr() 和 delattr()也有同样的限制，且与直接引用__dict__一样。

9.7. Odds and Ends 其他事项

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

有时候拥有一个类似 Pascal 中的“record”或者 C 语言中的“struct”是非常有用的，它们都可以绑定一写命名的数据项。一个空的类定义可以很完美的完成这些：

```
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 get the data from a string buffer instead, and pass it as an argument.

如果一段代码需要,可以通过一个特有的抽象数据类型可以通过传入一个模拟该类型的所有的方法的类来得到。比如，如果你有一个将从文件对象中得到的数据进行格式化的函数,你可以定义一个拥有 read() 和 readline()方法的类，它们从一个字符串缓存中获取数据，并将其作为一个参数。

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

实例方法对象也有属性：m.im_self 是方法 m()的实例对象，而且 m.im_func 是该方法对应的函数对象。

9.8. Exceptions Are Classes Too 异常也是类

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

用户自定义的异常也被识别为类。使用这种机制可以创建出可扩展的异常继承。

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

对>raise 语句有两种新且合理的（语义上）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:

在第一种形式中，instance 必须是 Class 或者其派生类的一个实例。第二种形式更加好用：

```
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:

在 except 从句中的类与异常是相符的，如果它是该类或者其派生类（反过来说——发生异常的类型如果是列表中类型的父类则不相符的）。比如，如下的代码将按照顺序打印 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.

注意，如果 except 从句顺序颠倒（except B 在第一位），它将打印 B,B,B——最开始匹配的从句将被触发。

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

当一个未被处理的异常错误信息被打印的时候，该异常类的名字被打印，然后一个分号和空格，最后是将实例使用内置函数 str()将实例转化为字符串。

9.9. Iterators 迭代器

By now you have probably noticed that most container objects can be looped over using a for statement:

到现在为止也许你已经注意到几乎所有的容器对象可以使用一个 for 语句进行遍历：

```
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:

这种读取形式清晰、简单而且方便。迭代器在 Python 中的使用普遍且统一。在后台，for 语句在容易上调用 iter()。该函数返回一个迭代器对象，该迭代器定义了每次获取容器中的一个元素的 next()方法。当没有更多的元素时，next()抛出一个 StopIteration 异常告诉 for 循环终止。下面的例子展现了其工作过程：

```
>>>s = 'abc'
>>>it = iter(s)
>>>it
<iterator object at 0x00A1DB50>
>>>it.next()
'a'
>>>it.next()
'b'
>>>it.next()
'c'
>>>it.next()

Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    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:

在认识了迭代器协议机制后，可以很容易的对你的类添加迭代行为。定义一个__iter__()方法，该方法返回一个有 next()方法的对象。如果该类定义了 next()，那么__iter__()只需返回 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. Generators 生成器

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 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:

Generator 是一个简单而且威力巨大的用于创建迭代器的工具。它们像一般的函数一样，不过任何时候需要返回数据，它们就使用 yield 语句。每次 next()被调用，生成器回复它脱离的位置（它记住了语句最后一次执行的位置和所有的数据值）。下面的例子可以展示生成器

可以非常简单的创建：

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

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

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.

任何生成器可以做的事情也可以用前一节所讲述的基于类的迭代器来做。生成器显得如此简洁，因为它自动生成了__iter__() 和 next()方法。

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 instance variables like self.index and self.data.

另外一个重要的特征是局部变量和执行状态在调用之间会被自动保存。这使得函数更容易写，也比使用类似 self.index 和 self.data 的实例变量更加清楚。

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.

对于自动方法创建和程序状态保存补充一点，当生成器终止，它自动抛出 StopIteration 异常。组合起来，这些特征使得创建一个迭代器更加简单，而不用写一个普通函数。

9.11. Generator Expressions 生成器表达式

Some simple generators can be coded succinctly as expressions using a syntax similar to list comprehensions but with parentheses instead of brackets. These expressions are designed for situations where the generator is used right away by an enclosing function. Generator expressions are more compact but less versatile than full generator definitions and tend to be more memory friendly than equivalent list comprehensions.

一些简单的生成器能够被简单的编码为一个表达式，该表达式使用类似链式推导表的语法，不过用圆括号代替方括号。这些表达式是为了在函数调用生成器所设计的。它更加简洁，但是不如完整的生成器定义那么多变，而且通常比等价的链式推导更加容易记。

Examples:

例子：

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

285



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

>>>yvec = [7, 5, 3]

>>>sum(x*y for x,y in zip(xvec, yvec))         # dot product

260



>>>from math import pi, sin
```

```
>>>sine_table = dict((x, sin(x*pi/180)) for x in range(0, 91))

>>>unique_words = set(word  for line in page  for word in line.split())

>>>valedictorian = max((student.gpa, student.name) for student in
graduates)

>>>data = 'golf'
>>>list(data[i] for i in range(len(data)-1,-1,-1))
['f', 'l', 'o', 'g']
```

10. Brief Tour of the Standard Library 标准库简介

10.1. Operating System Interface OS 接口

The os module provides dozens of functions for interacting with the operating system:

os 模块提供了一系列与操作系统打交道的函数：

```
>>>import os
>>>os.system('time 0:02')
0
>>>os.getcwd()      # Return the current working directory
'C:\\Python26'
>>>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 built-in open() function which operates much differently.

应该使用 import os 而不是 from os import *的样式。这样会保证随操作系统不同而变化的 os.open()不会覆盖内建函数 open()。

The built-in dir() and help() functions are useful as interactive aids for working with large modules like os:

内建函数 dir() 和 help()对于在与像 os 之类巨大模块工作的时候能起到很好的交互帮助作用。

```
>>>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:

对于日常的文件和文件夹处理，shutil 提供了一个高层次的易用接口：

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

10.2. File Wildcards 文件通配符

The glob module provides a function for making file lists from directory wildcard searches:

glob 模块头功了一个从文件夹通配符查找中建立文件列表的函数：

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

10.3. Command Line Arguments 命令行参数

Common utility scripts often need to process 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。这些参数以列表的形式存储在 sys 模块的 *argv* 属性中。例如，在命令行上运行 python demo.py one two three 将产生如下结果：

```
>>>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()函数处理 *sys.argv*。更多强大且适用性好的命令行处理工具在 optparse 有提供。

10.4. Error Output Redirection and Program Termination 错误输出重定向与程序终止

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 还具有属性 *stdin*，*stdout*，和 *stderr*。即使在 *stdout* 被重定向后，后者也能够显示警告和错误信息。

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

The most direct way to terminate a script is to use sys.exit().

最直接的终止程序的方法是使用 sys.exit()。

10.5. String Pattern Matching 字符串正则匹配

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。对于复杂的匹配与操作，正则表达式提供了简单高效的解决方案：

```
>>>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，字符串的方法就足够了，因为它们更容易阅读与调试：

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

```
'tea for two'
```

10.6. Mathematics 数学

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 模块提供了产生随机数的工具：

```
>>>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. Internet Access 互联网访问

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

有许多模块用于访问互联网以及处理互联网协议。两个最简单的是由于返回网络地址数据的 urllib2模块和发送邮件用的 smtplib 模块：

```
>>>import urllib2
```

```
>>>for                            line                           in
urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
...    if 'EST' in line or 'EDT' in line:  # look for Eastern Time
...        print line

<BR>Nov. 25, 09:43:32 PM EST

>>>import smtplib
>>>server = smtplib.SMTP('localhost')
>>>server.sendmail('soothsayer@example.org',
'jcaesar@example.org',
..."""To: jcaesar@example.org
...From: soothsayer@example.org
...
...Beware the Ides of March.
...""")
>>>server.quit()
```

(Note that the second example needs a mailserver running on localhost.)

（注意，第二个例子中需要在本地运行一个邮件服务器。）

10.8. Dates and Times 日期和时间

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 timezone aware.

datetime 模块提供了用于处理复杂或者简单情况下的日期与时间的类。在日期和时间计算被支持的同时，模块重点工作放在高效率的处理格式化输出和操作。模块还支持时区处理的对象。

```
>>># dates are easily constructed and formatted
>>>from datetime import date
>>>now = date.today()
>>>now
datetime.date(2003, 12, 2)
>>>now.strftime("%m-%d-%y.%d %b %Y is a %A on the%d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'
```

```
>>># dates support calendar arithmetic
>>>birthday = date(1964, 7, 31)
>>>age = now - birthday
>>>age.days
14368
```

10.9. Data Compression 数据压缩

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

以下模块支持通用的数据打包和压缩格式：zlib，gzip，bz2，zipfile 和 tarfile。

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

10.10. Performance Measurement 性能度量

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

一些 Python 程序员对于使用不同方法解决同一问题的性能问题有很大的兴趣。Python 提供了一些对这些问题立刻就可以解决的度量工具

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 a modest performance advantage:

比如，可能要尝试使用元组封装和拆封而不是传统的方式来交换元素。timeit 模块很方便的验证了最新的性能优化。

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

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.

与 timeit 的细粒度相比，profile 和 pstats 模块提供了针对更大块代码的时间度量工具。

10.11. Quality Control 质量控制

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是为每个函数书写测试用例，并且在函数完成后用这些测试反复进行测试。

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 模块提供了能够扫描模块以及根据嵌入程序中的字符串文档进行测试的工具。测试构建非常简单，只需要将一个典型调用的结果剪切并粘贴到字符串文档中即可。通过用户提供的例子，它发展了文档字符串，语序 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 模块并不如 doctest 模块好用，不过它可以在一个单一文件中提供完整的测试集。

```
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 有一种“batteries included”的哲学。Python 可以通过使用更大的包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 modules names, no direct knowledge or handling of XML is needed.

- xmlrpclib 和 SimpleXMLRPCServer 模块实现了在琐碎的任务重调用远程过程。尽管有这样的名字，其实用户不需要直接处理 XML，亦不需要有相关的知识。

- 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 包是一个用于处理邮件信息的库，它包括 MIME 和其他 RFC2822为基础的消息文档。它不像 smtplib 和 poplib 一样真正的发送和接收消息，email 包有一个用于构建或者解码复杂消息结构（包括附件）以及实现互联网编码和协议头的完整工具集。

- 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 包为流行数据格式解析提供了强大的支持。同样的 csv 模块提供了在公共数据库格式中进行读写的支持。总的来说，这些模块极大的简化了 Python 程序与其他工具之间的数据交换。

- Internationalization is supported by a number of modules including gettext, locale, and the codecs package.

- 国际化也被多个模块支持，包括 gettext，locale， 以及 codecs 包。

11. Brief Tour of the Standard Library – Part II 标准库简介 II

This second tour covers more advanced modules that support professional programming needs. These modules rarely occur in small scripts.

第二部分提供了更多支持专业编程的高级模块。这些模块很少出现在小型脚本中。

11.1. Output Formatting 输出格式化

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

repr 模块提供了一个版本的 repr()用于打印巨大或者深层的容器：

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

The pprint module offers more sophisticated control over printing both built-in and user defined objects in a way that is readable by the interpreter. When the result is longer than one line, the “pretty printer” adds line breaks and indentation to more clearly reveal data structure:

pprint 提供了一种更加老练的对解释器可读的打印方式，该方式适用于内建或者用户自定义的对象。当结果长于一行的时候，“pretty printer”会加入行分割并提供缩进以便更清楚展现数据的结构：

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

The textwrap module formats paragraphs of text to fit a given screen width:

textwrap 模块将给定的文本段落格式化使之适应特定的屏幕宽度：

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

The locale module accesses a database of culture specific data formats. The grouping attribute of locale’s format function provides a direct way of formatting numbers with group separators:

locale 用预定的国家信息数据结构访问数据库。locale 的格式化函数集合提供了对数字进行分组的格式化方式。

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

11.2. Templating 模板

The string module includes a versatile Template class with a simplified syntax suitable for editing by end-users. This allows users to customize their applications without having to alter the application.

string 包含一个灵活多变的 Template 类，而对于终端用户来说，只需要一个简单的语法即可编辑。这允许用户在无需改变应用的情况下自定义它们的应用。

The format uses placeholder names formed by $ with valid Python identifiers (alphanumeric characters and underscores). Surrounding the placeholder with braces allows it to be followed by more alphanumeric letters with no intervening spaces. Writing $$ creates a single escaped $:

格式采用了$和一些合法的 Python 标识符（数字、字母和下划线）组成的占位符。在占位符周围加上括号将使得它后面可以添加更多的字母或者数字二不必在意空格。$$创建一个$符号：

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

The substitute() method raises a KeyError when a placeholder is not supplied in a dictionary or a keyword argument. For mail-merge style applications, user supplied data may be incomplete and the safe_substitute() method may be more appropriate — it will leave placeholders unchanged if data is missing:

substitute()会在字典或者关键字参数上未提供占位符时抛出 KeyError 异常。对于邮件合并风格的应用中，用户提供的数据可能并不完整，则 safe_substitute()方法会更加适合——如果数据丢失，它会保持占位符不变：

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

Template subclasses can specify a custom delimiter. For example, a batch renaming utility for a photo browser may elect to use percent signs for placeholders such as the current date, image sequence number, or file format:

模板子类能够制定一个定制分隔符。比如，一个照片浏览的批量重命名工具会使用百分号作为当前日期、图片编号或者文件格式之类的符号的占位符：

```
>>>import time, os.path
```

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

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

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

Another application for templating is separating program logic from the details of multiple output formats. This makes it possible to substitute custom templates for XML files, plain text reports, and HTML web reports.

另外一个模板的应用是逻辑的将程序从多种输出格式细节中分离出来。这样就可以用定制模板替代 XML 文件、文本文件报告以及 HTML 网络报告。

11.3. Working with Binary Data Record Layouts 使用二进制记录层

The struct module provides pack() and unpack() functions for working with variable length binary record formats. The following example shows how to loop through header information in a ZIP file without using the zipfile module. Pack codes "H" and "I" represent two and four byte unsigned numbers respectively. The "<" indicates that they are standard size and in little-endian byte order:

struct 模块提供了 pack() 和 unpack()函数用于变长二进制记录格式。以下的例子显示了如何在不使用 zipfile 模块的情况下遍历 ZIP 文件中的各个文件的文件头。压缩代码中的"H" 和 "I"分别表示2字节和四字节的无符号数字。"<"表示它们大大小是标准的而且采用小端字节顺序：

```
import struct
```

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

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

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

11.4. Multi-threading 多线程

Threading is a technique for decoupling tasks which are not sequentially dependent. Threads can be used to improve the responsiveness of applications that accept user input while other tasks run in the background. A related use case is running I/O in parallel with computations in another thread.

线程是一种分离无顺序依赖任务的技术。线程能够提高在其他任务在后台运行时当前需要接受用户输入的应用的响应速度。一个相关的例子就是与其他线程一起并行竞争 IO。

The following code shows how the high level threading module can run tasks in background while the main program continues to run:

下面的代码在高层次展示了在主程序继续运行的情况下 threading 能够在后台继续运行其他线程。

```
import threading, zipfile

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

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

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

The principal challenge of multi-threaded applications is coordinating threads that share data or other resources. To that end, the threading module provides a number of synchronization primitives including locks, events, condition variables, and semaphores.

多线程引用最主要的挑战在于协调多个共享数据或者其他资源的线程。为此，threading 模块提供了包括锁、事件、条件变量和信号量之类的同步原语。

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

虽然这些工具功能强大，但是一些很小的设计错误会导致一些不可重复的问题出现。所以，最好的协调任务的方法是在在一个单一的线程中集中进行资源访问，然后使用 Queue 将其他线程的请求传入该线程。使用 Queue.Queue 对象的用于线程间通信与协调的应用将更加容易设计，而且可读性好，可靠性强。

11.5. Logging 日志

The logging module offers a full featured and flexible logging system. At its simplest, log messages are sent to a file or to sys.stderr:

logging 模块提供了一个功能全面而且扩展性良好的日子系统。最简单的情况下，日志消息被发送到文件或者是 sys.stderr：

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file%s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

This produces the following output:

这将产生如下的输出：

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

By default, informational and debugging messages are suppressed and the output is sent to standard error. Other output options include routing messages through email, datagrams, sockets, or to an HTTP Server. New filters can select different routing based on message priority: DEBUG, INFO, WARNING, ERROR, and CRITICAL.

默认情况下，报告和调试信息被捕获并发送到标准错误流。其他输出选项包括在 email、datagrams、sockets 或者一个 HTTP 服务器中传递消息。新的过滤器能够根据消息的级别选择不同的传递方式：DEBUG, INFO， WARNING，ERROR，和 CRITICAL

The logging system can be configured directly from Python or can be loaded from a user editable configuration file for customized logging without altering the application.

日志系统可以直接通过 Python 来配置，或者可以通过一个可编辑的配置文件来配置而无需改变应用程序。

11.6. Weak References 弱引用

Python does automatic memory management (reference counting for most objects and *garbage collection* to eliminate cycles). The memory is freed shortly after the last reference to it has been eliminated.

Python 进行自动内存管理（对大多数对象进行引用计数，用 *garbage collection* 以便循环利用。）内存在最后一次引用后被释放。

This approach works fine for most applications but occasionally there is a need to track objects only as long as they are being used by something else. Unfortunately, just tracking them creates a reference that makes them permanent. The weakref module provides tools for tracking objects without creating a reference. When the object is no longer needed, it is automatically removed from a weakref table and a callback is triggered for weakref objects. Typical applications include caching objects that are expensive to create:

这种方式对于绝大部分的应用来说工作良好,不过在偶然的情况下还需要对某些对象进行跟踪，直到他们被其他一些东西使用。不幸的是，跟踪他们就会对他们形成一个引用从而使得它们不被释放。weakref 模块提供了跟踪对象而不形成引用的工具。当对象不再需要，它自动从 weakref 表中被删除，并且会为之触发一个回调函数。典型的引用包括对一些创建代价很高的对象进行缓存：

```
>>>import weakref, gc
>>>class A:
...    def __init__(self, value):
...            self.value = value
...    def __repr__(self):
...            return str(self.value)
...
>>>a = A(10)                  # create a reference
>>>d = weakref.WeakValueDictionary()
>>>d['primary'] = a           # does not create a reference
>>>d['primary']              # fetch the object if it is still alive
10
>>>del a                     # remove the one reference
>>>gc.collect()              # run garbage collection right away
0
>>>d['primary']              # entry was automatically removed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']             # entry was automatically removed
  File "C:/python26/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7. Tools for Working with Lists 列表工具

Many data structure needs can be met with the built-in list type. However, sometimes there is a need for alternative implementations with different performance trade-offs.

许多的数据结构都能够用内建的列表类型表示。然而，有时需要不同的代价实现。

The array module provides an array() object that is like a list that stores only homogeneous data and stores it more compactly. The following example shows an array of numbers stored as two byte unsigned binary numbers (typecode "H") rather than the usual 16 bytes per entry for regular lists of Python int objects:

array 模块提供了一个 array()对象，该对象和列表类似，不过只能够存储同类型的数据，而且存储更加紧凑。如下的例子展示了一个以两字节的无符号二进制数（类型号"H"）存储的数列，而不是用采用平常列表中用16个字节才能够存储一个 Python 对象的形式：

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

The collections module provides a deque() object that is like a list with faster appends and pops from the left side but slower lookups in the middle. These objects are well suited for implementing queues and breadth first tree searches:

collections 模块提供了一个可以快速从左边插入和弹出但是在中间部分查找比较慢的 deque()对象。这些对象适用于队列及广度优先搜索树的实现：

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

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

In addition to alternative list implementations, the library also offers other tools such as the bisect module with functions for manipulating sorted lists:

补充一点，该库还提供了其他工具，比如 bisect 模块提供了操作已排序数列的函数：

>>>import bisect

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

The heapq module provides functions for implementing heaps based on regular lists. The lowest valued entry is always kept at position zero. This is useful for applications which repeatedly access the smallest element but do not want to run a full list sort:

heapq 模块提供了基于普通序列而实现的堆。最小的元素总是在位置0.这对于一些频繁访问最小元素却不想对整个序列排序的应用非常有用。

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

11.8. Decimal Floating Point Arithmetic 十进制浮点算法

The decimal module offers a Decimal datatype for decimal floating point arithmetic. Compared to the built-in float implementation of binary floating point, the class is especially helpful for

decimal 模块提供了一个 Decimal 用于十进制浮点数算法的数据类型。与内建的 float 二进制浮点实现相比，该类对以下的情况特别有用：

- financial applications and other uses which require exact decimal representation,
- 一些类似财政应用之类的需要精确十进制表达的应用，
- control over precision,

- 精度控制，
- control over rounding to meet legal or regulatory requirements,
- 控制舍入以符合法律或者通常的需要，
- tracking of significant decimal places, or
- 跟踪重要的十进制位，或者
- applications where the user expects the results to match calculations done by hand.
- 用户希望结果与手算一致的应用。

For example, calculating a 5% tax on a 70 cent phone charge gives different results in decimal floating point and binary floating point. The difference becomes significant if the results are rounded to the nearest cent:

比如，70分电话费的5%税计算，十进制浮点数和二进制浮点数的计算就有差异。如果结果要求舍入到最接近的分值，这个差别就显得特别重要了：

```
>>>from decimal import *
>>>Decimal('0.70') * Decimal('1.05')
Decimal('0.7350')
>>>.70 * 1.05
0.73499999999999999
```

The Decimal result keeps a trailing zero, automatically inferring four place significance from multiplicands with two place significance. Decimal reproduces mathematics as done by hand and avoids issues that can arise when binary floating point cannot exactly represent decimal quantities.

Decimal 的结果总是保有结尾的0，自动从两位精度延伸到4位。Decimal 重现了手工数学运算，这就保证了二进制浮点数无法确保的精度。

Exact representation enables the Decimal class to perform modulo calculations and equality tests that are unsuitable for binary floating point:

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

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

The decimal module provides arithmetic with as much precision as
needed:
>>>getcontext().prec = 36
>>>Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```

12. What Now?

Reading this tutorial has probably reinforced your interest in using Python — you should be eager to apply Python to solving your real-world problems. Where should you go to learn more?

This tutorial is part of Python's documentation set. Some other documents in the set are:

The Python Standard Library:

You should browse through this manual, which gives complete (though terse) reference material about types, functions, and the modules in the standard library. The standard Python distribution includes a *lot* of additional code. There are modules to read Unix mailboxes, retrieve documents via HTTP, generate random numbers, parse command-line options, write CGI programs, compress data, and many other tasks. Skimming through the Library Reference will give you an idea of what's available.

Installing Python Modules explains how to install external modules written by other Python users.

The Python Language Reference: A detailed explanation of Python's syntax and semantics. It's heavy reading, but is useful as a complete guide to the language itself.

More Python resources:

- http://www.python.org: The major Python Web site. 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.

- http://docs.python.org: Fast access to Python's documentation.

- http://pypi.python.org: The Python Package Index, previously also nicknamed the Cheese Shop, is an index of user-created Python modules that are available for download. Once you begin releasing code, you can register it here so that others can find it.

- http://aspn.activestate.com/ASPN/Python/Cookbook/: The Python Cookbook is a sizable collection of code examples, larger modules, and useful scripts. Particularly notable contributions are collected in a book also titled Python Cookbook (O'Reilly & Associates, ISBN 0-596-00797-3.)

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

13. 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.

13.1. Line Editing

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

13.2. History Substitution

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

13.3. Key Bindings

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

```
key-name: function-name
or
"string": function-name
and options can be set with
set option-name value
For example:
# I prefer vi-style editing:
set editing-mode vi

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

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

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

```
Tab: complete
```

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

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

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

```

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

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

```
# 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=/home/user/.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
```

13.4. Alternatives to the Interactive Interpreter

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.

One alternative enhanced interactive interpreter that has been around for quite some time is

IPython, which features tab completion, object exploration and advanced history management. It can also be thoroughly customized and embedded into other applications. Another similar enhanced interactive environment is bpython.

14. 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 of 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 uses the built-in 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, and it is not a bug in your code either. 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 built-in 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, summing ten values of 0.1 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 str.format() method's format specifiers in *Format String Syntax*.

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

```
>>>0.1
0.10000000000000001
```

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:

```
>>>2**52
4503599627370496L
>>>2**53
9007199254740992L
>>>2**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(2**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 * 2**56
7205759403792794.0
```

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

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

meaning that the exact number stored in the computer is approximately equal to the decimal value 0.100000000000000005551115123125. 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!).