Python 3.1

入门指南

羲庭父 / 整理

(并非我翻译，仅作整理，以便传看)

Blog:www.u1u5.cc

Msn:ct100@live.cn

E-mail:ct100@live.cn

目　录

1. 开胃菜.....4
2. 使用 Python 解释器.....5
   2.1. 调用 Python 解释器.....5
   2.1.1. 参数传递.....6
   2.1.2. 交互模式.....6
   2.2. 解释器及其环境.....6
   2.2.1. 错误处理.....6
   2.2.2. 执行 Python 脚本.....7
   2.2.3. 源代码编码.....7
   2.2.4. 交互执行文件.....7
3. Python 简介.....8
   3.1. 将 Python 用作计算器.....8
   3.1.1. 数字.....9
   3.1.2. 字符串.....11
   3.1.3. 关于 Unicode.....15
   3.1.4. 列表.....15
   3.2. 迈向编程的第一步.....17
4. 深入 Python 流程控制.....18
   4.1. if Statements if 语句.....18
   4.2. for 语句.....18
   4.3. range() 函数.....19
   4.4. break 和 continue 语句，以及循环的 else 子句.....20
   4.5. pass 语句.....21
   4.6. 定义函数.....21
   4.7. 深入 Python 函数定义.....23
   4.7.1. 默认参数值.....23
   4.7.2. 关键字参数.....24
   4.7.3. 可变参数列表.....26
   4.7.4. 拆分参数列表.....26
   4.7.5. Lambda 方式.....27
   4.7.6. 文档字符串.....27
   4.8. PS：编码风格.....27
5. 数据结构.....28
   5.1. 深入列表.....28
   5.1.1. 用列表实现堆栈.....29
   5.1.2. 用列表实现队列.....30
   5.1.3. 列表推导式.....30
   5.1.4. 嵌套列表推导式.....31
   5.2. del 语句.....32

5.3. 元组和序列.....32
5.4. 集合.....33
5.5. 字典.....34
5.6. 遍历技巧.....35
5.7. 深入条件控制.....37
5.8. 比较序列和其他类型.....37
6. 模块.....38
6.1. 深入 Python 模块.....39
6.1.1. Executing modules as scripts 像脚本一样执行模块.....39
6.1.2. The Module Search Path 模块的搜索路径.....40
6.1.3. “Compiled” Python files “编译的”Python 文件.....40
6.2. Standard Modules 标准模块.....41
6.3. dir() 函数.....41
6.4. 包.....43
6.4.1. 从一个包中导入 *.....44
6.4.2. 包内引用.....45
6.4.3. 跨目录的包.....45
7. 输入输出.....45
7.1. 格式化输出.....45
7.1.1. 旧式字符窜格式化.....49
7.2. 文件读写.....49
7.2.1. 文件对象的方法.....49
7.2.2. `pickle`模块.....51
8. 错误和异常.....52
8.1. 语法错误.....52
8.2. 异常.....52
8.3. 异常处理.....53
8.4. 抛出异常.....55
8.5. 用户自定义异常.....55
8.6. 定义清理动作.....57
8.7. 预定义的清理动作.....58
9. 类.....58
9.1. 术语相关.....59
9.2. 作用域和命名空间.....59
9.2.1. 作用域和命名空间示例.....60
9.3. 初识类.....61
9.3.1. 类定义语法.....61
9.3.2. 类对象.....61
9.3.3. 实例对象.....62
9.4. 一些说明.....63
9.5. 继承.....64
9.5.1. 多重继承.....65
9.6. 私有变量.....66
9.7. 备注.....66

9.8. 异常也是类 67
9.9. 迭代器 67
9.10. 生成器 69
9.11. 生成器表达式 69
10. Python 标准库概览 70
10.1. 操作系统接口 70
10.2. 文件通配符 71
10.3. 命令行参数 71
10.4. 错误输出重定向和程序终止 71
10.5. 字符串模式匹配 71
10.6. 数学 72
10.7. 互联网访问 72
10.8. 日期和时间 73
10.9. 数据压缩 73
10.10. 性能评测 73
10.11. 质量控制 74
10.12. “瑞士军刀” 74
11. 标准库概览 — 第二部分 75
11.1. 输出格式化 75
11.2. 模板 76
11.3. 使用二进制数据记录布局 77
11.4. 多线程 77
11.5. 日志 78
11.6. 弱引用 79
11.7. 列表工具 79
11.8. 十进制浮点数计算 80
12. 现在做什么呢？ 81
13. 交互的输入编辑和历史记录 82
13.1. 行编辑 82
13.2. 历史记录 82
13.3. 键绑定 83
13.4. 注释 84

入门指南

Release: 3.1
Date: September 05, 2009

Python 是一门简单易学且功能强大的编程语言。它拥有高效的高级数据结构，并且能够用简单而又高效的方式进行面向对象编程。 Python 优雅的语法和动态类型，再结合它的解释性，使其在大多数平台的许多领域成为编写脚本或开发应用程序的理想语言。

你可以自由的从 Python 官方点，http://www.python.org，以源代码或二进制形式获取 Python 解释器及其标准扩展库，并可以自由的分发。此站点同时也提供了大量的第三方 Python 模块、程序和工具，及其附加文档。

你可以很容易的使用 C 或 C++（其他可以通过 C 调用的语言）为 Python 解释器扩展新函数和数据类型。 Python 还可以被用作定制应用程式的一门扩展语言。

本手册非正式的向读者介绍了 Python 语言及其体系相关的基本知识与概念。在学习实践中结合使用 Python 解释器是很有帮助的，不过所有的例子都是完整的，所以本手册亦可离线阅读。

如果需要了解相关标准库或对象的详细介绍，请查阅 Python 库参考文档。 Python 参考手册提供了更多语言相关的正式说明。如果想要使用 C 或 C++编写扩展，请查阅 Python 解释器扩展和集成章节或 Python/C API 参考手册。当然也可阅读一些深入介绍 Python 知识的图书。

本手册不会尝试涵盖 Python 的全部知识和每个特性，甚至不会涵盖所有常用的特性。相反的，它介绍了 Python 中许多最引人瞩目的特性，并且会给你一个关于语言特色和风格的认识。读完之后，你将能够阅读和编写 Python 模块或程序，并为以后使用 Python 库参考手册继续学习诸多 Python 模块库做好准备。

1. 开胃菜

如果你要用计算机做很多工作，最后你会发现有一些任务你更希望用自动化的方式进行处理。比如，你想要在大量的文本文件中执行查找/替换，或者以复杂的方式对大量的图片进行重命名和整理。也许你想要编写一个小型的自定义数据库、一个特殊的 GUI 应用程式或一个简单的小游戏。

如果你是一名专业的软件开发者，可能你必须使用几种 C/C++/JAVA 类库，并且发现通常编写/编译/测试/重新编译的周期是如此漫长。也许你正在为这些类库编写测试用例，但是发现这是一个让人烦躁的工作。又或者你已经完成了一个可以使用扩展语言的程式，但你并不想为此重新设计并实现一套全新的语言。

那么 Python 正是你所需要的语言。

虽然你能够通过编写 Unix shell 脚本或 Windows 批处理文件来处理其中的某些任务，但 Shell 脚本更适合移动文件或修改文本数据，并不适合编写 GUI 应用程序或游戏；虽然你能够使用 C/C++/JAVA 编写程序，但即使编写一个简单的 *first-draft* 程序也有可能耗费大量的开发时间。相比之下，Python 更易于使用，无论在 Windows、Mac OS X 或 Unix 操作系统上它都会帮助你更快的完成任务。

虽然 Python 易于使用，但它却是一门完整的编程语言；与 Shell 脚本或批处理文件相比，它为编写大型程序提供了更多的结构和支持。另一方面，Python 提供了比 C 更多的错

误检查，并且作为一门 *高级语言* ，它内置支持高级的数据结构类型，例如：灵活的数组和字典。因其更多的通用数据类型，Python 比 Awk 甚至 Perl 都适用于更的多问题领域，至少大多数事情在 Python 中与其他语言同样简单。

Python 允许你将程序分割为不同的模块，以便在其他的 Python 程序中重用。 Python 内置提供了大量的标准模块，你可以将其用作程序的基础，或者作为学习 Python 编程的示例。这些模块提供了诸如文件 I/O、系统调用、sockets 支持，甚至类似 Tk 的用户图形界面（GUI）工具包接口。

Python 是一门解释型语言，因为无需编译和链接，你可以在程式开发中节省宝贵的时间。 Python 解释器可以交互的使用，这使得试验语言的特性、编写临时程序或在自底向上的程序开发中测试方法非常容易。你甚至还可以把它当做一个桌面计算器。

Python 让程序编写的紧凑和可读。用 Python 编写的程式通常比同样的 C、C++或 Java 程式更短小，这是因为以下几个原因：

高级数据结构使你可以在一条语句中表达复杂的操作；

语句组使用缩进代替开始和结束大括号来组织；

变量或参数无需声明。

Python 是“可扩展”的：如果你会 C 语言编程便可以轻易的为解释器添加内置函数或模块，或者为了对性能瓶颈作优化，或者将 Python 程序与只有二进制形式的库（比如某个专业的商业图形库）连接起来。一旦你真正掌握了它，你可以将 Python 解释器集成进某个 C 应用程序，并把它当做那个程序的扩展或命令行语言。

顺便说一句，这个语言的名字来自于 BBC 的“Monty Python's Flying Circus”节目，和爬行类动物没有任何关系。在文档中引用 Monty Python 的典故不仅可行，而且值得鼓励！

现在你已经为 Python 兴奋不已了吧，大概想要领略一些更多的细节！学习一门语言最好的方法就是使用它，本指南推荐你边读边使用 Python 解释器练习。

下一节中，我们将解释 Python 解释器的用法。这是很简单的一件事情，但它有助于试验后面的例子。

本手册剩下的部分将通过示例介绍 Python 语言及系统的诸多特性，开始是简单的语法、数据类型和表达式，接着介绍函数与模块，最后涉及异常和自定义类这样的高级内容。

2. 使用 Python 解释器

2.1. 调用 Python 解释器

Python 解释器通常被安装在目标机器的 `/usr/local/bin/python3.1` 目录下。将 `/usr/local/bin` 目录包含进 Unix shell 的搜索路径里，以确保可以通过输入：

```
python3.1
```

命令来启动他。 [2] 由于 Python 解释器的安装路径是可选的，这也可能是其他路径，你可以联系安装 Python 的用户或系统管理员确认。（例如， `/usr/local/python` 就是一个常见的选择）

在 Windows 机器上，Python 通常安装在 `C:\Python31` 位置，当然你可以在运行安装向导时修改此值。要想把此目录添加到你的 PATH 环境变量中，你可以在 DOS 窗口中输入以下命令：

```
set path=%path%;C:\python31
```

通常你可以在主窗口输入一个文件结束符（Unix 系统是 Control-D ，Windows 系统是 Control-Z ）让解释器以0状态码退出。如果那没有作用，你可以通过输入 import sys; sys.exit() 命令退出解释器。

Python 解释器具有简单的行编辑功能。在 Unix 系统上，任何 Python 解释器都可能已经添加了 GNU readline 库支持，这样就具备了精巧的交互编辑和历史记录等功能。在 Python 主窗口中输入 Control-P 可能是检查是否支持命令行编辑的最简单的方法。如果发出嘟嘟声（计算机扬声器），则说明你可以使用命令行编辑功能；更多快捷键的介绍请参考 *Interactive Input Editing and History Substitution 交互的输入编辑和历史记录* 。如果没有任何声音，或者显示 ^P 字符，则说明命令行编辑功能不可用；你只能通过退格键（backspace）从当前行删除已键入的字符并重新输入。

Python 解释器有些操作类似 Unix shell：当使用终端设备（tty）作为标准输入调用时，它交互的解释并执行命令；当使用文件名参数或以文件作为标准输入调用时，它读取文件并将文件作为 *脚本* 执行。

第二中启动 Python 解释器的方法是 python -c command [arg] ... ，这种方法可以在 *命令行* 执行 Python 语句，类似于 shell 中的 *-c* 选项。由于 Python 语句通常会包含空格或其他特殊 shell 字符，一般建议将 *命令* 用单引号包裹起来。

有一些 Python 模块也可以当作脚本使用。你可以使用 python -m module [arg] ... 命令调用它们，这类似在命令行中键入完整的路径名执行 *模块* 源文件一样。

注意 python file 和 python <file 是不同的：对于后一种情况，程序期望从 *文件* 中获得输入，类似调用 sys.stdin.read() 。因为在程序执行前解释器已经读取了全部文件内容，所以程序将指向文件尾；前一种情况（通常是你期望的）它们可以是任何连接到 Python 解释器标准输入的文件或设备。

在使用脚本时，有时希望在运行脚本后能进入交互模式。这可以通过在脚本前传递 *-i* 参数来实现。（如果脚本需要从标准输入获取数据这样就没有作用，原因如前所述。）

2.1.1. 参数传递

运行解释器时，脚本名和其后附带的参数通过字符窜列表变量 sys.argv 传递给脚本。如果没有传递脚本和任何参数时，它至少有一个空字符串元素 sys.argv[0] 。当脚本名为 '-'（意指标准输入）时， sys.argv[0] 的值为 '-' 。当使用 *-c 命令* 时， sys.argv[0] 的值为 '-c' 。当使用 *-m 模块* 时， sys.argv[0] 的值为完整的模块地址名。在 *-c 命令* 或 *-m 模块* 之后的命令选项不会被 Python 解释器的选项处理器所截获，而是保存在 sys.argv 中供命令或模块处理。

2.1.2. 交互模式

当从终端（tty）读取命令时，我们称 Python 解释器工作在 *交互模式* 中。在此模式下它通过 *主提示符* 提示下一个命令，通常为三个大于号（ >>> ）；对于连续的行通过 *次提示符* 提示，默认为三个点号（ ... ）。解释器在第一行提示符之前会输出一条欢迎信息说明它的版本号和版权提示.

```
$ python3.1
Python 3.1a1 (py3k, Sep 12 2007, 12:21:02)
[GCC 3.4.6 20060404 (Red Hat 3.4.6-8)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

当输入一个多行结构时，必须保证是连续的行。例如下面这个 if 语句:

```
>>>the_world_is_flat = 1
```

```
>>>if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

2.2. 解释器及其环境

2.2.1. 错误处理

在错误发生时，Python 解释器会打印出错误信息及其栈跟踪信息。在交互模式下，它将返回主提示符；如果是从文件输入，它将在打印出栈跟踪信息后以非零状态退出。（ try 语句中的 except 从句处理的异常不属于此类错误。）一些错误是致命的并且会导致非零状态退出，这通常是由内部冲突或内存溢出所导致。所有的错误信息都会被写进标准错误流中，命令行中的普通输出被写进标准输出中。

在主提示符或从提示符下输入中断字符（通常是 Control-C 或 DEL）就会取消输入并回到主提示符。

[4]在执行命令时输入中断字符将抛出 KeyboardInterrupt 异常，你可以通过 try 语句捕获。

2.2.2. 执行 Python 脚本

```
#! /usr/bin/env python3.1
```

在 BSD 类 Unix 系统中，只要在 Python 脚本首行输入以下文本行（要确认 Python 解释器路径包含在用户 **PATH** 变量中）并赋予文件可执行权限就能使其像 Shell 脚本一样直接运行：

```
#! /usr/bin/env python3.1
```

#! 必须是文件的前两个字符。在某些平台上，首行必须以 Unix 行结束符（ '\n' ）结束，而不能是 Windows 行结束符（ '\r\n' ）。注意 '#' 符号是 Python 注释的起始符。

可以通过 **chmod** 命令给 Python 脚本指定可执行模式或权限：

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

在 Windows 系统上，并没有“执行模式”的概念。 Python 安装程序会用 python.exe 自动关联 .py 文件，当双击某个 Python 文件时就可以将其作为脚本运行了。文件扩展名也可以是 .pyw ，在这种情况下，通常出现的终端窗口将被禁用。

2.2.3. 源代码编码

默认情况下，Python 源文件是 UTF-8编码。在此编码下，全世界大多数语言的字符可以同时用在字符串、标识符和注释中 — 尽管 Python 标准库仅使用 ASCII 字符做为标识符，这只是任何可移植代码应该遵守的约定。如果要正确的显示所有的字符，你的编辑器必须能识别出文件是 UTF-8编码，并且它使用的字体能支持文件中所有的字符。

你也可以为源文件指定不同的字符编码。为此，在 #! 行（首行）后插入至少一行特殊的注释行来定义源文件的编码。

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

通过此声明，源文件中所有的东西都会被当做用 *encoding* 指代的 UTF-8编码对待。在

Python 库参考手册 codecs 一节中你可以找到一张可用的编码列表。

```
# -*- coding: cp-1252 -*-
```

例如，如果你的编辑器不支持 UTF-8编码的文件，但支持像 Windows-1252的其他一些编码，你可以定义：

```
# -*- coding: cp-1252 -*-
```

这样就可以在源文件中使用 Windows-1252字符集中的所有字符了。这个特殊的编码注释必须在文件中的 *第一或第二* 行定义。

2.2.4. 交互执行文件

当你交互使用 Python 时，通常在每次 Python 解释器启动后都需要执行一些标准的命令。你可以设定一个名为 **PYTHONSTARTUP** 的环境变量指向包含以上命令的文件名来做这些工作。这类似于 Unix shell 的 .profile 特性。

这个文件仅在交互会话中被读取，当 Python 从脚本中读取命令或以 /dev/tty 作为命令源时则不会被读取（尽管这些行为很像交互会话）。它与交互命令在同一个命名空间内执行，所有由它定义或引入的对象可以毫无限制的在会话中使用。你同样可以在这个文件中包含修改 sys.ps1 和 sys.ps2 的指令。

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

如果你想在当前工作目录中读取附加的启动文件，你可以在全局的启动文件中通过使用 if os.path.isfile('.pythonrc.py'): exec(open('.pythonrc.py').read()) 代码编程实现。如果你想在某个脚本中使用启动文件，必须在脚本中明确的执行以下代码:

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

Footnotes

[1] 在 Unix 系统上，Python 3.1解释器默认未被安装成名为 python 的命令，所以它不会与同时安装在系统中的 Python 2.x 命令冲突。

[2] GNU Readline 包的一个问题可能禁止此功能。

3. Python 简介

在下面的示例中，我们通过有没有提示符（ >>> 或 ... ）来区分输入和输出：为了试验示例，你必须在提示符出现时输入其后的所有内容；那些不以提示符开头的行是 Python 解释器的输出内容。注意示例中某行内出现从提示符意味着你必须输入一个空白行，这被用来结束多行命令输入。

本手册中的很多示例，甚至那些在交互提示符下的输入，都包含有注释。 Python 中的

注释以特殊字符 `#` 开始，直到物理行的结束。注释可以出现在一行的开始位置，也可以跟在空白字符或代码后面，但是不能包含在字符串中间。在字符串中的特殊字符就是其字面意思。由于注释是用来说明代码的，并且不会被 Python 解释执行，可以在输入示例时将他们省略。

一些简单的例子:

```
# 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. 将 Python 用作计算器

让我们练习一些简单的 Python 命令。启动 Python 解释器并等待主提示符 `>>>` 出现（这并不需要多久）。

3.1.1. 数字

将 Python 解释器作为一个简单的计算器：你可以输入一个表达式，计算器会返回其值。表达式语言很简单： `+`, `-`, `*` 和 `/` （加、减、乘、除）运算符像在大多数其他语言中一样工作（例如，Pascal 或 C）；括号（ `(` 和 `)` ）可以用来区分优先级。例如：

```
>>>2+2
4
>>># This is a comment
...2+2
4
>>>2+2  # and a comment on the same line as code
4
>>>(50-5*6)/4
5.0
>>>8/5 # Fractions aren't lost when dividing integers
1.6000000000000001
```

注意：有时你可能会得到不同的结果；浮点数在不同机器上的运算结果可能是不同的。后面我们将对控制浮点数输出的显示结果做更多说明；这里我们看到的仅是有效的显示，并非我们能得到的可读性更好的结果。

```
>>>print(8/5)
1.6
```

本指南中，为了清晰，除非我们特别的讨论格式化输出，我们将使用更简单的浮点数输出格式，后面再对为何这两种浮点数显示方式会有不同作解释。完整的信息请参照 *Floating Point Arithmetic: Issues and Limitations* *浮点数运算：问题和限制* 章节。

对整数做除法运算并想去除小数部分取得整数结果时，可以使用另外一个运算符， `//` ：

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

```
-3
```

在 Python 中，等号（ '=' ）符号被用于给变量赋值。随后，在下一个交互提示符前不会有任何输出结果：

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

你可以同时给几个变量赋一个值。

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

在 Python 中，变量在使用前必须“定义”，否则就会引发错误。

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

Python 完全支持浮点数，在混合类型操作数运算中整数将被转换为浮点数。

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

同样，Python 也支持复数，虚数以 `j` 或 `J` 结尾。实部为非零的复数写作 `(real+imagj)`，或者通过 `complex(real,imag)` 构造方法创建。

```
>>>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)
```

复数总是被当做两个浮点数表示：实部和虚部。可以使用 `z.real` 和 `z.imag` 从复数 *z* 中获得这两个部分。

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

整数和浮点数转换函数（ int() ， float() ）均不能用于复数 — 实际上并没有一种正确的方法能将复数转换为实数。使用 abs(z) 可以得到复数 z 的模（以浮点数形式表示）或者使用 z.real 得到它的实部。

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

在交互模式下，最后一次表达式输出被赋予变量 _ 。这说明当你把 Python 当做桌面计算器使用时，可以很方便的进行连续计算。例如：

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

对于用户而言此变量应该是只读的。不要明确的对其作赋值操作 — 如此你将创建一个独立的同名局部变量，并屏蔽具有魔术行为的内置变量。

3.1.2. 字符串

除了数值，Python 还可以通过几种方式操作字符串。字符串可以是用单引号（’）或双引号（”）包围。

```
>>>'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.'
```

Python 解释器按照字符串被输入的方式打印字符串结果：为了显示准确的值，字符串包含在成对的引号中，引号和其他特殊字符要用反斜线（ \ ）转译。如果字符串只包含单引号（ ' ）而没有双引号（ " ）就可以用双引号（ " ）包围，反之用单引号（ ' ）包围。再强调一下， print() 函数可以生成可读性更好的输出。

有几种方法可以让字符串文本显示为多行。你可以使用连续行：通过在行尾添加一个反斜线（ \ ）表示下一行是它逻辑上的延续。

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

print(hello)
```

注意：还是要在字符串中插入 \n 来换行，反斜线（ \ ）后面的换行会被忽略。这个例子将会输出以下内容：

```
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.
```

如果我们将字符串文本表示为原始值（即 Python 中字符串的 r 作用符）， \n （换行符）序列不会被转换为换行，并且行尾的反斜线（ \ ）和代码中的换行符都将作为数据包含在字符串中。如下所示：

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

print(hello)
```

将会输出：

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

字符串可以使用 + 符号（加号）连接，并可以使用 * 符号（星号）重复：

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

两个彼此相邻的字符串文本可以自动的连接起来，上面第一行的列子也可以写成 word = 'Help' 'A' 的形式。这只能用于两个字符串文本中，而不能用于两个字符串表达式中。

```
>>>'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
```

字符串可以通过下标引用，和 C 语言一样，第一个字符的下标是 *0* 。 Python 中没有单独的字符类型，单个字符就是只包含一个字符的字符串。类似图像编程语言，子字符串可以使用 *切片标记* 来指定：使用冒号分割的两个下标索引。

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

索引切片具有实用的默认值：如果省略第一个索引，则默认为 *0* ；如果省略第二个索引，则默认为被操作字符串的长度。

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

不同于 C 语言的字符串，在 Python 中字符串是不可改变的。尝试通过索引给字符串赋值将引发错误！

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

然而，通过联合创建字符串即简单又高效。

```
>>>'x' + word[1:]
'xelpA'
```

```
>>>'Splat' + word[4]
'SplatA'
```

切片操作具有一个有用的恒等特性： `s[:i] + s[i:]` 等于 ``s``。

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

Python 能够优雅的处理那些没有意义的切片索引：一个过大的索引值（即下标值大于字符串实际长度）将被字符串实际长度所代替，当上边界比下边界大时（即切片左值大于右值）就返回空字符串。

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

索引也可以是负数，这将导致从右边开始计算。例如：

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

请注意 *-0* 实际上就是 *0* ，所以它不会导致从右边开始计算！

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

超过边界的负数索引将被截断，但不要尝试对单元素（非切片操作）做如此操作。

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

理解切片如何工作的一种方式就是把索引想象成字符 *之间* 的指点，第一个字符左边是数字 *0* 。那么包含 *n* 个字符的字符串最后一个字符的右边索引即是 *n* ，例如：

```
 +---+---+---+---+---+
```

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

第一行数字显示了字符串中索引0...5的位置，第二行数字显示了对应的负数索引。从 *i* 到 *j* 之间的切片由这两个标记之间的所有字符组成。

对于非负索引，如果索引在边界内，切片的长度是两个索引的差。例如， `word[1:3]` 的长度是2。

内置函数 `len()` 返回字符串的实际长度。

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

3.1.3. 关于 Unicode

从 Python 3.0开始所有的字符串都支持 Unicode（参考 http://www.unicode.org ）。

Unicode 为古代或现代每种语言文本中使用的每个字符提供了统一编码。以前，只有256个可用的语言字符编码，文本被绑定到将编码映射到语言字符的代码页上。这使得软件的国际化（通常写作 `i18n` — i +18个字符+ n ）极为困难。 Unicode 通过为所有的语言定义一个代码页解决了这个问题。

如果想在字符串中包含特殊字符，你可以使用 Python 的 *Unicode_Escape* 编码方式。下面的列子展示了如何这样做：

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

转码序列 `\u0020` 表示在指定位置插入编码为0x0020的 Unicode 字符（空格）。

其他字符就像 Unicode 编码一样被直接解释为对应的编码值。如果你有在许多西方国家使用的标准 Latin-1编码的字符串，你会发现编码小于256的 Unicode 字符和在 Latin-1编码中的一样。

除了这些标准编码，Python 还提供了一整套基于其他已知编码创建 Unicode 字符串的方法。

字符串对象提供了一个 `encode()` 方法用以将字符串转换成特定编码的字节序列，它接收一个小写的编码名称作为参数。

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

3.1.4. 列表

Python 支持几种 *复合* 数据类型，用来对其他值进行分组。最常用的是 *列表（list）* ，在中括号中用逗号分隔的一系列值（项）。列表中的项不需要是同一种类型。

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

类似字符串索引，列表索引也是从 *0* 开始，并且列表也可作切片、连接等操作。

```
>>>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!']
```

与字符串的 *不可变性* 不同，列表中每个元素都是可变的。

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

给列表切片赋值也是允许的，甚至可以改变它的大小或清空整个列表。

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

内置函数 `len()` 同样可以作用于列表。

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

你可以将列表嵌套使用（创建包含其他列表的列表），例如：

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

你可以在列表末尾添加内容：

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

注意：在最后的列子里， `p[1]` 和 `q` 实际上指向同一个对象！我们将在后面讨论 *对象语法* 。

3.2. 迈向编程的第一步

当然，我们可以用 Python 作比2加2更复杂的任务。例如，我们可以像下面一样写出 *菲波那契数* 序列的初始子序列：

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

这个例子展示了几个新功能。

第一行包含了一条 *多项赋值* 表达式：变量 `a` 和 `b` 同时得到新值 *0* 和 *1* 。最后一行我们又一次这样使用，这说明等号右边的表达式在任何赋值前被首先计算。等号右边的表

达式遵从从左向右的求值顺序。

只要条件（此处为：`b < 10`）为真 `while` 循环就会一直执行。像 C 语言一样，在 Python 中任何非零的整数值都为 *True* ，0为 *False* 。条件也可以是一个字符串或列表值，事实上可以是任何序列，所有长度非零的序列都是 *True* ，空的序列为 *False* 。例子中使用的测试是一个简单的比较。标准的比较操作符和 C 语言中的写法一样：`<`（小于）、`>`（大于）、`==`（等于）、`<=`（小于等于）、`>=`（大于等于）和 `!=`（不等于）。

循环的 *主题* 是 *缩进* 代码块：缩进是 Python 语法分组的方法。 Python（仍然！）没有提供一种智能的行输入功能，所以你必须为每个缩进行输入制表符（Tab）或空格（Space）。实践中你应该准备一个文本编辑器来应对更复杂的 Python 代码输入，大多数文本编辑器都具备自动缩进功能。当交互式的输入一个复合语句时，必须在其后输入一个空行以表明输入完成（因为解释器无法猜出你什么时候输入最后一行）。注意：代码块中的每行必须具有相同数目的缩进。

输出函数 `print()` 将给定的表达式输出。与简单的以你想要输出的表达式输出方式不同（像前面我们在计算器示例中的做法），它可以处理多个表达式，大浮点数或字符串。字符串被不带引号的打印，并且两项之间会插入一个空格，所以你可以更好的格式化字符串，列如：

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

关键字 *end* （参数）可以用来避免输出后换行，或者用不同的字符串结束输出。

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

4. 深入 Python 流程控制

除了上一节介绍的 `while` 循环语句外，Python 像其他语言一样支持常用的流程控制语句，当然也有一些自己的特性。

4.1. if Statements if 语句

`if` 语句也许是最常用的语句类型。例如：

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

```
...        print('Single')
...else:
...        print('More')
...
More
```

if 语句可以有零个或多个 elif 分支语句，并且 else 分支语句是可选的。 elif 关键字是'else if'的缩写，这可以有效避免过度缩进问题。在 Python 中，使用 if ... elif ... elif 语句组合代替其他语言中的 switch 和 case 语句结构。

4.2. for 语句

Python 的 :keywor:`for` 语句可能与你在 C 或 Pascal 语言中用过的有些不同。与总是依靠一个等差数字进行迭代(类似 Pascal)或者赋予用户定义迭代步进和终止条件的能力(类似 C）不同，Python 的 for 语句是对任何序列（列表或字符串）的项按照它们在序列中的顺序进行迭代。例如（没有歧义）：

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

在循环中修改正在被迭代的序列是不安全的（这仅在不可变序列类型上发生，像 *列表* ）。如果需要修改正在迭代的列表（比如复制选择的项），你必须对它的副本进行迭代。切片操作可以方便的做到这一点。

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

4.3. range() 函数

如果你必须迭代一个数字序列，内置函数 range() 会派上用场。它生成一个等差级数序列。

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

```
3
4
```

给定的终值从不会包含在生成的列表中， `range(10)` 生成10个值，一个长度为10的序列项的合法索引。可以让 `range` 函数从另一个数值开始，或者指定一个不同的步进值（甚至是负数，有时这被称为“步长”）。

```
range(5, 10)
    5 through 9

range(0, 10, 3)
    0, 3, 6, 9

range(-10, -100, -30)
  -10, -40, -70
```

想要迭代序列的索引，你可以结合使用 `range()` 和 `len()` （实现）。如下所示：

```
>>>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
```

然而在大多数此类情况下，使用 `enumerate()` 函数更方便。参考 *Looping Techniques 遍历技巧* 。

如果你只是打印一个 *range* （方法调用）会发生奇怪的事情。

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

在不同方面 `range()` 函数返回的对象表现为它是一个列表，但事实上它并不是。当你迭代它时，它是一个能够像期望的序列返回连续项的对象；但为了节省空间，它并不真正构造列表。

我们称此类对象是 *可迭代的* ，即适合作为那些期望从某些东西中获得连续项直到结束的函数或结构的一个目标（参数）。我们已经见过的 `for` 语句就是这样一个 *迭代器* 。 `list()` 函数是另外一个（迭代器），它从可迭代（对象）中创建列表。

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

稍后我们会看到更多返回可迭代（对象）和以可迭代（对象）作为参数的函数。

4.4. break 和 continue 语句，以及循环的 else 子句

像在 C 语言中一样， break 语句用于跳出最近的 for 循环或 while 循环。

continue 语句，同样借鉴自 C 语言，继续循环的下一次迭代过程。

循环语句可以包含一个 else 子句，当循环因穷尽列表项结束（ for 循环）或循环条件变为逻辑假结束（ while 循环）时执行它，而通过 break 语句结束循环时并不执行。下面这个搜索素数的循环是一个间的示例：

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

4.5. pass 语句

pass 语句不做任何事。它被用于语法结构上需要一条语句但并不做任何事时。例如：

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

这通常被用在创建最小类上。

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

另外一个可以用 pass 的地方是当你新编写一个函数结构或条件体代码时，这允许你在更多的抽象层保持思维。 pass 被默默的忽略。

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

4.6. 定义函数

我们可以创建一个用以生成指定边界的斐波那契数列的函数。

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

关键字 def 声明一个函数 *定义* 。其后必须跟着函数名和以括号标明的参数列表。必须在下一行开始构成函数体的语句，并且必须缩进。

函数体的第一行语句可以是可选的字符串文本，这个字符串是函数的文档字符串，或者称为 *docstring* 。（更多关于 *docstrings* 的信息请参考 *Documentation Strings 文档字符串* 。）有些工具通过 *docstrings* 自动生成在线的或可打印的文档，或者让用户通过代码交互浏览；在你的代码中包含 *docstrings* 是一个好的实践，让它成为习惯吧。

函数 *调用* 会为函数局部变量生成一个新的符号表。确切的说，所有函数中的变量赋值都是将值存储在局部符号表。变量引用首先在局部符号表中查找，然后是包含函数的局部符号表，然后是全局符号表，最后是内置名字表。因此，全局变量不能在函数中直接赋值（除非用 global 语句命名），尽管他们可以被引用。

当函数调用时，实参被引入调用函数的局部符号表；因此，参数是通过 *传值* 方式传递的（这里 *值* 总是指代对象的 *引用* ，而非对象的值）。 [2] 当一个函数调用其他函数时，将为此调用创建一个新的字符表。

一个函数定义会在当前符号表内引入函数名。函数名指代的值（即函数体）有一个被 Python 解释器认定为 *用户自定义函数* 的类型。这个值可以赋予其他的名字（即变量名），然后它也可以被当做函数使用。这可以作为通用的重命名机制。

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

如果你使用过其他语言，你可能会反对说： fib 不是一个函数，而是一个方法，因为它并不返回任何值。事实上，没有 return 语句的函数确实会返回一个值，虽然是一个相当令人厌烦的值（指 None ）。这个值被称为 None （这是一个内建名称）。如果 None 值是唯一被书写的值，那么在写的时候通常会被解释器忽略（即不输出任何内容）。如果你确实想看到这个值的输出内容，请使用 print() 函数。

```
>>>fib(0)
>>>print(fib(0))
```

```
None
```

定义一个返回斐波那契数列数字列表的函数，而不是打印它，是很简单的。

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

照例，这个列子包含了一些 Python 的新特性：

return 语句从函数中返回一个值（可以是多个）。没有表达式参数的 return 语句返回 None 值。函数执行最后（指没有 return 语句的情况）也会返回 None 值。

result.append(b) 语句调用了列表对象 result 的一个 *方法* 。方法是属于对象的函数，并被命名为 obj.methodname ，此处 obj 是某个对象（也可能是个表达式），methodname 是对象类型定义的方法的名字。不同的（对象）类型定义了不同的方法。不同（对象）类型的方法可能拥有相同的名字，但这并不会导致胡乱。（你可以使用 *clasees* 对应自己的对象类型和方法，参考 *Classes 类* ）示例中出现的 append() 方法是由列表对象定义的，它将一个新元素添加到列表的末尾。在这个示例中它等同于 result = result + [b] （表达式运算），但是更高效。

4.7. 深入 Python 函数定义

在 Python 中，你也可以定义包含若干参数的函数。这里有三种可用的形式，也可以混合使用。

4.7.1. 默认参数值

最常用的一种形式是为一个或多个参数指定默认值。这会创建一个可以使用比定义时允许的参数更少的参数调用的函数。

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

```
            print(complaint)
```

这个函数可以通过集中方式调用：

- giving only the mandatory argument: `ask_ok('Do you really want to quit?')` 仅给出必须得参数： `ask_ok('Do you really want to quit?')`
- giving one of the optional arguments: `ask_ok('OK to overwrite the file?', 2)` 给出一个可选的参数： `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!')` 或者给出所有的参数： `ask_ok('Ok to overwite 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.

本示例同样介绍了 `in` 关键字。它检测序列中是否包含某个指定的值。

默认值赋予在函数定义时 *定义* 域的值，例如：

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

将会输出 `5` 。

Important warning: 默认值只被赋值一次。这使得当默认值是可变对象时会有所不同，比如列表、字典或者大多数类的实例。例如，下面的函数在后续调用过程中会累积（前面）传给它的参数。

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

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

这将输出：

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

如果你不想让默认值在后续调用中累积，你可以像下面一样定义函数。

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

4.7.2. 关键字参数

函数同样可以使用 keyword = value 形式通过关键字参数调用。例如，下面的函数：

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

可以通过以下任何方式调用：

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

但是以下调用是错误的：

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

通常，参数列表必须（先书写）位置参数然后才是关键字参数，这里关键字必须来自于形参名字。形参是否有一个默认值并不重要。任何参数都不能被多次赋值——在同一个调用中，与位置参数相同的形参名字不能用作关键字。这里有一个违反此限制而出错的例子：

```
>>>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'
```

当最后一个形参为 **name 形式时，函数接受一个包含所有未出现在形参列表中的关键字参数的字典（参考 *Mapping Types — dict* ）。这可以结合形式为 *name 的形参（在下一节中介绍）一起使用，它接受一个包含所有未出现在形参列表中的位置参数的元组（ *name 必须出现在 **name 之前）。例如，我们定义如下函数：

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

它可以这样调用：

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

他会输出如下内容：

```
-- 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
```

注意：关键字参数名的列表内容在打印前需要对关键字字典的 `keys()` 方法返回的值排序（即在输出列表前需要对列表的键排序），否则参数输出的循序是未知的。

4.7.3. 可变参数列表

最后，最常用的选择是指明一个函数可以使用任意数目的参数调用。这些参数将被包装进一个元组（参考 *Tuples and Sequences 元组和序列* ）。在可变数目的参数前，可以有零或多个普通的参数。

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

通常，这些 `可变化的` 参数在形参列表的最后定义，因为他们会收集传递给函数的所有剩下的输入参数。任何出现在 `*args` 参数之后的形参只能是“关键字参数”，即他们只能通过关键字而非位置参数使用。

```
>>>def concat(*args, sep="/"):
...     return sep.join(args)
...
>>>concat("earth", "mars", "venus")
'earth/mars/venus'
>>>concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

4.7.4. 拆分参数列表

一种相反的情况是：当参数是一个列表或元组但函数需要分开的位置参数时，就需要拆分参数。比如，内建函数 `range()` 期望独立的 *start* 和 *stop* 参数。如果他们不是独立的有效（参数），调用函数时使用 `*-` 操作符将参数从列表或元组中拆分出来即可。

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

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

以此类推，字典可以使用 **- 操作符拆分成关键字参数。

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

4.7.5. Lambda 方式

出于普遍需求，一些通常在函数式编程语言中出现的特性被加入到 Python 中，如 Lisp。使用 lambda 关键字，可以创建短小的匿名函数。这里有一个返回两个参数之和的（Lambda）函数： lambda a, b: a+b 。 Lambda 方式可是使用在任何需要函数对象的地方。出于语法的限制，他们只能是一个独立的表达式。从语义上讲，它们只是普通函数定义的语法糖。类似于嵌套函数定义，lambda 方式可以从包含域内引用变量。

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

4.7.6. 文档字符串

这里是一些关于文档字符串内容和格式的约定。

第一行应该总是关于对象用途的摘要。为了简便起见，无需明确说明对象的名称或类型，因为这些都可以通过其他方法获取（除非名字碰巧是描述一个函数操作的动词）。这一行应该以大写字母开头，并且以句号结束。

如果文档字符串包含多行，第二行应该是空行，在视觉上将摘要（指第一行）与区域的描述内容分割。接下来应该是一段或多段（内容），用来描述对象调用约定和边界效应等。

在 Python 中，语法分析器不会从多行字符串中去除缩进，因此如果需要处理文档的工具必须自己去除缩进。这是通过以下的约定实现的：第一行 *之后* 的第一个非空行决定了整个文档字符串缩进的值。（我们不用第一行是因为它通常紧接在字符串起始引号，以致它在字符串中的缩进并不明显。）字符串中所有行开始的“相当于”缩进的空白都会被去除掉。不应出现缩进不足的行（指有的行缩进空格数少于整体），如果出现此情况所有的前导空白都应该被去除。在扩展制表符后需要对相等的空白做测试（通常为8个空格）。

这里是一个多行文档字符串的示例：

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

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

4.8. PS：编码风格

现在你即将编写更长的，更复杂的 Python 代码，是时候讨论一下 *编码风格* 了。大多数语言可以用不同的风格编写（或更简单的 *格式化* ），其中有一些比其他具有更好的可读性。让其他人容易阅读你的代码总是一个好主意，而采用一个好的编码风格对此尤为有益。

对于 Python 而言， **PEP 8** 已经成为大多数项目使用的风格，它提供了一个非常可读的并且诱人的编码风格。每一位 Python 开发者都应该找个时间读一下它，这里是为你概括的最重要的几点：

使用4个空格的缩进，并杜绝制表符。

4个空格是一个介于小缩进（允许更深层的嵌套）和大缩进（更利于阅读）好的折中（选择）。制表符会导致混乱（因为不同编辑器上的制表符表示的空格可能不同），最好让他滚蛋。

换行，避免他们超过79个字符。

这可以帮助使用小显示器的用户，并且使得在大显示器上并排显示几个代码文件成为可能。

使用空行分割类和函数以及函数内部较大的代码块。

如果可以，将注释独立成行。

使用文档字符串

在操作符两侧和逗号之后使用空格，但不要直接在括号内部使用：`a = f(1, 2) + g(3, 4)` 。

使用统一风格命名你的类和函数：约定使用 `CamelCase` 方式命名类，使用 `lower_case_with_underscores` 方式命名函数和方法。

总是用 `self` 作为方法的第一个参数名（参考 *A First Look at Classes* *初识类* 深入类和方法）。

如果你希望你的代码被用在国际环境中，不要使用新奇的编码。 Python 默认的 UTF-8 编码甚至 ASCII 编码在任何情况下都将工作的更好。

同样的，如果仅有极少数说其他语言的人将要阅读或维护代码，不要使用非 ASCII 字符的标识符。

Footnotes

[1] 事实上， *对象引用调用* 可能是个更好的表述，因为如果传递一个可变对象（参数），调用者将会观察到被调用者对其所做的任何改变（向列表插入元素）。

5. 数据结构

本章详细讨论了你已经学过的一些知识，同样也添加了一些新内容。

5.1. 深入列表

Python 的列表数据类型包含更多的方法。这里是所有的列表对象方法：

list.append(*x*)

在列表末尾添加一个元素，等同于 a[len(a):] = [x] 。

list.extend(*L*)

通过添加指定列表的所有元素扩展列表，等同于 a[len(a):] = L 。

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

在指定位置插入一个元素。第一个参数是将要插入在前面的元素的索引，因此 a.insert(0, x) 会插入在列表头部，同样 a.insert(len(a), x) 就等同于 a.append(x) 。

list.remove(*x*)

删除列表中值为 *x* 的第一个元素。如果不存在这样的元素则引发错误。

list.pop([*i*])

删除列表中指定位置元素并返回它（指元素值）。如果省略索引， a.pop() 会删除并返回列表中的最后一个元素。（方法中围绕 *i* 的方括号表示此参数是可选的，而不是你也要那个位置输入方括号。你会在 Python 库参考手册中经常看到此用法。）

list.index(*x*)

返回列表中值为 *x* 的第一个元素的索引。如果不存在这样的元素则引发错误。

list.count(*x*)

返回列表中元素 *x* 出现的次数。

list.sort()

对列表中的元素进行排序。

list.reverse()

反转列表中的元素。

一个演示列表大多数方法的列子：

```
>>>a = [66.25, 333, 333, 1, 1234.5]
>>>print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>>a.insert(2, -1)
>>>a.append(333)
>>>a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>>a.index(333)
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. 用列表实现堆栈

列表方法使其可以很容易的实现堆栈功能，即最后添加的元素是第一个被返回的元素（LIFO）。要想在堆栈顶部添加一个元素，可以使用 append() 方法。要想返回堆栈顶部的元素，可以使用不指定索引参数的 pop() 方法。例如：

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

5.1.2. 用列表实现队列

你也可以方便的使用列表实现一个队列，即第一个被添加的元素第一个被返回（FIFO）。想要在队列尾部添加一个元素，可以使用 append() 方法。想要返回队列首部的元素，可以使用指定参数为0的 pop() 方法（即 list.pop(0) ）。例如：

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

5.1.3. 列表推导式

列表推导式为从序列中创建列表提供了一个简单的方法。普通的应用程式通过将一些操作应用于序列的每个成员并通过返回的元素创建列表，或者通过满足特定条件的元素创建子序列。

列表推导式由包含一个表达式的括号组成，表达式后面跟随一个 for 子句，之后可以有零或多个 for 或 if 子句。结果是一个列表，由表达式依据其后面的 for 和 if 子句上下文计算而来的结果构成。如果希望表达式产生一个元组，则必须用括号包裹。

这里我们通过将列表中的每个元素乘3返回一个新列表：

```
>>>vec = [2, 4, 6]
>>>[3*x for x in vec]
[6, 12, 18]
```

现在我们玩点小花样：

```
>>>[[x, x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
```

下面我们对序列中的每个元素应用方法：

```
>>>freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>>[weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
```

我们还可以使用 if 子句进行过滤：

```
>>>[3*x for x in vec if x > 3]
[12, 18]
>>>[3*x for x in vec if x < 2]
[]
```

经常可以不适用括号创建元组，但这里不行：

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

这里有一个嵌套循环和其他小技巧的演示：

```
>>>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]
```

列表推导式亦可以使用复杂的表达式和嵌套函数：

```
>>>[str(round(355/113, i)) for i in range(1, 6)]
```

```
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.1.4. 嵌套列表推导式

如果你已对此抱有兴趣，是的，列表推导式可以嵌套。他们是一种功能强大的工具，但和所有功能强大的工具一样，如果确实需要使用，你需要格外谨慎。

考虑以下由包含三个列表的列表构成的3x3矩阵，每行一个列表。

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

现在，如果想要交换行和列你应该使用列表推导式。

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

嵌套 列表推导式需要特别注意：

当嵌套列表推导式时，想要避免麻烦就从右到左读取。

下面是一个关于这个片段（指上文中的嵌套列表推导式）更冗长的清晰版本：

```
for i in [0, 1, 2]:
    for row in mat:
        print(row[i], end="")
    print()
```

在实际中，你应该更喜欢使用内置函数组成复杂流程语句。对此种情况 `zip()` 函数将会做的更好：

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

更多关于本行中使用的星号（*）的说明，参考 *Unpacking Argument Lists* *拆分参数列表* 章节。

5.2. del 语句

有一个方法可以使用索引而不是值从列表中删除一个元素： `del` 语句。这个具有返回值的 `pop()` 方法不同。 `del` 语句也可以用来删除列表的一个片段或者清空整个列表（前面我们是通过给切片赋一个空列表）。例如：

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

```
>>>del a[:]
>>>a
[]
```

del 也可以用来删除整个变量：

```
>>>del a
```

此后，任何对 a 名字的引用就会出错（至少在对它赋其它值前）。稍后，我们会发现 del 的其他用法。

5.3. 元组和序列

我们知道列表和字符串具有很多通用的属性，例如索引和切片操作。他们是 *序列* 数据类型（参考 *Sequence Types — str, bytes, bytearray, list, tuple, range* ）的两个例子。因为 Python 是一中不断进化的语言，也可能添加其他序列数据类型（支持）。这里是另一种标准序列数据类型： *元组（tuple）* 。

元组由若干逗号分隔的值组成，例如：

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

如你所见，元组输出时总是用括号包裹的，这便于正确的表达嵌套元组。在输入时两边的括号是可选的，但不论如何括号通常是必须得（如果元组是更大的表达式的一部分）。

元组有很多用处。例如：(x, y)坐标点，数据库中员工的记录，等等。像字符串一样，元组是不可变的：不能给元组的独立元素赋值（尽管你可以使用切片和连接来模仿这些功能）。也可以使用包含可变对象创建元组，比如列表。

一个特殊的问题就是构造包含0或1个元素的元组：为了适用此情况，语法上需要一个额外的做法。空元组由一对空括号构成，包含一个元素的元组需要在其后面跟一个逗号来构成（在括号中包含一个值是不够得）。丑陋，但这有效。比如：

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

t = 12345, 54321, 'hello!' 语句是 *元组封装* 的一个示例：值 12345 、54321 和 hello! 被

封装进一个元组。其逆反操作也是可以的：

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

这非常适合称为 *序列拆分* ，并且对所有的右操作序列都可以工作。序列拆分要求等号左边的变量数目和序列中的元素数目要相同。注意：多重赋值其实就是元组封装和序列拆分的结合。

5.4. 集合

Python 同样包含一种 *集合* 数据类型。集合就是一个包含不同元组的无序集。基本功能包括关系测试和剔除重复记录。集合对象同样支持数学操作，像联合（union）、交（intersection）、差（difference）和对称差（symmetric difference）。

大括号或 `set()` 函数可以用来创建集合。注意：想要创建空集合，你必须使用 `set()` 而不是 `{}` 。后者用于创建空字典，我们在下一节中介绍的一种数据结构。

一下是简单的演示：

```
>>>basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>>print(basket)
{'orange', 'banana', 'pear', 'apple'}
>>>fruit = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>>fruit = set(basket)               # create a set without duplicates
>>>fruit
{'orange', 'pear', 'apple', 'banana'}
>>>fruit = {'orange', 'apple'}       # {} syntax is equivalent to [] for lists
>>>fruit
{'orange', 'apple'}
>>>'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
{'a', 'r', 'b', 'c', 'd'}
>>>a - b                             # letters in a but not in b
{'r', 'd', 'b'}
>>>a | b                             # letters in either a or b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>>a & b                             # letters in both a and b
{'a', 'c'}
>>>a ^ b                             # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

类似 *for lists* ，这里有一种集合推导式语法：

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

5.5. 字典

另一种使用的 Python 内建数据类型是 *字典* （参考 *Mapping Types — dict* ）。字典有时在其他语言中称为“关联记忆”（ associative memories ）或“关联数组”（ associative arrays ）。与序列不同，序列是以连续的数字作为索引，而字典是以 *关键字* 作为索引。关键字可以是任意不可变类型，数字和字符串都可以作为关键字。如果元组只包含数字、字符串或元组，那么也可以作为关键字使用。如果元组直接或间接包含可变对象，那么就不可以作为关键字使用。你不能将列表作为关键字（使用），因为列表可以通过索引赋值、切片赋值或 append() 和 extend() 方法改变。

最好将字典理解成一个无序的 *key: value* 对（集合），其中关键字必须是互不相同的（在统一字典中）。可以用一对大括号 {} 会创建一个空字典。在大括号中放置用逗号分隔的 *key: value* 对将给字典添加初始化值，这也是字段输出的方式。

字典主要的操作就是根据关键字来存储或获取值。同样可以使用 del 删除 *key: value* 对。如果你使用一个已存在的关键字存储（新值），旧的值将被覆盖。试图使用一个不存在的关键字获取值将导致错误。

对一个字典执行 list(d.keys()) 将返回一个字典中所有关键字组成的无序列表（如果你想要排序，只需使用 sorted(d.keys()) ）。 [1] 使用 in 关键字（指 Python 语法）可以检查字典中是否存在某个关键字（指字典）。

这里是使用字典的一个小示例：

```
>>>tel = {'jack': 4098, 'sape': 4139}
>>>tel['guido'] = 4127
>>>tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>>tel['jack']
4098
>>>del tel['sape']
>>>tel['irv'] = 4127
>>>tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>>list(tel.keys())
['irv', 'guido', 'jack']
>>>sorted(tel.keys())
['guido', 'irv', 'jack']
>>>'guido' in tel
True
>>>'jack' not in tel
False
```

dict() 构造函数可以直接从 *key-value* 对中创建字典：

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

此外，字典推导式可以从任意的键值表达式中创建字典：

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

如果关键字都是简单的字符串，有时通过关键字参数指定 *key-value* 对更为方便：

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

5.6. 遍历技巧

当遍历字典时，关键字及其对应的值可以使用 items() 方法同时获得。

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

当遍历一个序列时，位置索引及其对应的值可以使用 enumerate() 函数同时获取。

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

想要同时遍历两个或更多序列，可以使用 zip() 函数将属性组合。

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

想要反序遍历一个序列，首先以正序指定这个序列，然后对其调用 reversed() 函数。

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

```
7
5
3
1
```

想要按顺序遍历一个序列，使用 sorted() 函数返回一个新的排序列表，这不会改变原序列。

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

5.7. 深入条件控制

在 while 和 if 语句中使用的条件可以是任何操作符，不仅仅是比较运算符。

T 比较操作符 in 和 not in 检查某个值是否存在（不存在）于某个序列中。操作符 is 和 is not 比较两个对象是否是同一个对象，这只和类似列表这样的可变对象有关。所有的比较操作符具有相同的优先级，比所有的数值操作符低。

比较操作可以串联（使用）。列如： a < b == c 会检查 a 是否小于 b ，并且 b 是否等于 c 。

比较操作可以混合使用逻辑操作符 and 和 or ，并且比较的结果（或者任何其他逻辑表达式）可以使用 not 操作取反。这些操作符（指 and , or 和 not ）的优先级低于比较操作符，在它们之中 not 具有最高的优先级， or 则最低。所以 A and not B or C 与 (A and (not B)) or C 是等价的。当然，可以使用括号表达期望的比较操作。

逻辑操作符 and 和 or 也被称作 *短路* 操作符。从左到右计算它们的参数，一旦结果确定就停止。例如：如果 A 和 C 是真，但 B 是假， A and B and C 就不会计算 C 表达式的值。当作为一个普通的值而非逻辑值时，短路操作的返回值是最后一个被计算的参数。

可以将比较操作或其他逻辑表达式的结果赋值给一个变量。例如：

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

注意：在 Python 中与 C 语言不同，赋值操作不能出现在表达式中。 C 程序员可能会对此抱怨，但是它避免了一个在 C 程序中经常遇到的典型的问题：想要在一个表达式中使用 == 操作时却输入了 = 。

5.8. 比较序列和其他类型

相同的序列对象可以进行比较操作。比较操作按 *字典序* 进行：首先比较前两个元素，

如果它们不同就会决定比较的结果；如果它们相等就会比较下两个元素，依次类推，直到两个序列末尾。如果被比较的两个元素也为相同类型的序列对象，就对其作递归字典序比较。如果两个序列的所有元素都相等，那么这两个序列就被认为是相等的。如果一个序列是另一个序列的初始子序列，较短的序列就小于另外一个。字符串的字典序按照单个字符的 Unicode 区位码顺序。一些同类型序列间比较操作的例子：

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

注意：只要对象含有合适的比较方法，不同类型对象之间通过 `<` 或 `>` 的比较是合法的。例如，不同数字类型会根据他们的值进行比较，因此0等于0.0，等等。此外，如果没有确定的排序，解释器将会抛出 `TypeError` 异常。

Footnotes

(1, 2)

1] 调用 `d.keys()` 会返回一个 *dictionary view* 对象。它支持类似关系测试和迭代的操作，但是它并非原始字典的副本——它仅是一个 *视图* 。

6. 模块

如果你退出 Python 解释器并重新进入，你做的任何定义（变量和方法）都会丢失。因此，如果你想要编写一些更大的程序，为准备解释器输入使用一个文本编辑器会更好，并以那个文件替代作为输入执行。这就是传说中的 *脚本* 。随着你的程序变得越来越长，你可能想要将它分割成几个更易于维护的文件。你也可能想在不同的程序中使用顺手的函数，而不是把代码在它们之间中拷来拷去。

为此，Python 提供了一种将定义保存在一个文件中的方法，然后在脚本中或解释器的交互实例中使用。这个文件被称作 *模块* ，模块中的定义可以被 *导入* 到其他的模块或者 *main* 模块（你在顶层进入执行脚本或计算模式下的变量的集合）。

模块就是一个包含 Python 定义和语句的文件。文件名就是添加了 `.py` 扩展名的模块名。在一个模块中，模块的名字（字符串形式）和全局标量 `__name__` 的值相同。例如，用你喜欢的编辑器在当前目录下建立一个包含以下内容的 `fibo.py` 文件。

```
# Fibonacci numbers module

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

def fib2(n): # return Fibonacci series up to n
    result = []
```

```
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

现在进入 Python 解释器并使用以下命令导入这个模块：

```
>>>import fibo
```

这并不会在当前符号表内直接引入 fibo 中定义的函数名，这里仅引入了 fibo 的模块名。你可以通过模块名来使用那些方法：

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

如果打算频繁使用一个函数，你可以将它赋予一个本地变量：

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

6.1. 深入 Python 模块

除了包含函数定义外，模块也可以包含可执行语句。这些语句一般用来初始化模块。他们仅在 *第一次* 被导入的地方执行一次。 [1]

每个模块都有自己私有的符号表，被模块内所有的函数定义作为全局符号表使用。因此，模块的作者可以在模块内部使用全局变量，而无需担心它与某个用户的全局变量意外冲突。从另一个方面讲，如果你确切的知道自己在做什么，你可以使用引用模块函数的表示法访问模块的全局变量， modname.itemname 。

模块可以导入其他的模块。一个（好的）习惯是将所有的 import 语句放在模块的开始（或者是脚本），这并非强制。被导入的模块名会放入当前模块的全局符号表中。

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

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

这种方式不会把操作的模块名引入当前符号表中（所以在这个列子中， fibo 是未定义的）。

更进一步，还有一种导入模块中定义的所有名称的变体：

```
>>>from fibo import *
>>>fib(500)
```

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

这会导入那些所有除了以下划线（ _ ）开头的名称。大多数情况，Pythong 程序员不会使用这个技巧。因为它会在解释器中引入一个未知的名称集合，很可能会将你已经定义的东西覆盖掉。

出于效率的理由，在一个解释器会话中每个模块仅被导入一次。所以，如果你修改了你的模块，你必须重启你的解释器——或者如果只是一个你想要交互测试的模块，使用 imp.reload() 方法。比如， import imp; imp.reload(modulename) 。

6.1.1. Executing modules as scripts 像脚本一样执行模块

当你使用以下方式运行 Python 模块时，模块中的代码便会被执行。

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

就像你导入它一样，但是 __name__ 会被设置为 '__main__' 。这意味着，在你的模块末尾添加如下的代码：

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

你就可以让文件即是可导入的模块又是可用的脚本，因为这些命令解析的代码仅当模块被当做“主”文件执行时才会被运行。

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

如果只是导入模块，代码将不会运行：

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

这通常被用来为模块提供一个方便的用户接口，或者为了测试目的（将模块当做脚本一样运行，以便执行一组测试）

6.1.2. The Module Search Path 模块的搜索路径

当一个名为 spam 的模块被导入时，解释器首先会在当前目录搜索一个名为 spam.py 的文件，然后是在环境变量 **PYTHONPATH** 中定义的目录列表。这和 shell 变量 **PATH** 具有一样的语法，即一系列目录名的列表。如果没有定义 **PYTHONPATH** ，或者按照里面的路径没有找到文件，解释器会继续在 Python 默认安装路径中搜索。在 Unix 系统上，这通常是 .:/usr/local/lib/python 。

实际上，模块都是在变量 sys.path 定义的目录列表中查找，它是从包含输入脚本的目录（当前目录）、 **PYTHONPATH** 和 Python 默认安装目录初始而来。这允许那些确切知道在做什么的 Pythong 程序可以修改或替换模块搜索路径。注意：因为包含执行脚本的目录也在搜索路径中，所以重要的是脚本与 Python 标准模块不应该具有相同的名字，否则当导入一个模块时 Python 将会尝试把脚本当做模块加载。这通常会导致一个错误。更多信息请参考 *Standard Modules 标准模块* 。

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

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

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

通常，你无需自行创建 spam.pyc 文件。每次 spam.py 成功编译后，都会尝试将编译版本写入 spam.pyc 文件。如果尝试写入失败，也不会引发什么错误。不论什么情况，如果文件没有被正确写入，目标文件 spam.pyc 会被认为是无效的并且会被忽略。 spam.pyc 文件的内容是平台无关的，所以一个 Python 模块目录可以被不同体系架构的机器共享。

一些专业的提示：

当使用 *-O* 标志启动 Python 解释器时，Python 会生成优化过的代码并存储在 .pyo 文件中。当前的优化器只能去除 assert 语句，初次之后别无用处。当使用 *-O* 时， *所有* 的 *bytecode* 都会被优化： .py 文件会被编译成优化的字节码，并且忽略 .pyc 文件。

为 Python 解释器传递两个 *-O* 标志（ *-OO* ）将使字节码编译器最优化执行，这可能在一些罕见的情况下导致程序执行异常。目前只会将 __doc__ 字符串从字节码中移除，并保存为更加紧凑的 .pyo 文件。虽然很多程序可能依赖这些有效的（优化），但你还是应该在确定的情况下使用这个选项。

程序不会因为从 .pyc 文件或 .pyo 文件中读取而比从 .py 文件中读取运行的更快。唯一提升的是他们的加载速度。

当从命令行通过指定文件名执行一个脚本时，脚本的字节码不会被写到 .pyc 或 .pyo 文件中。因此，通过将大多数代码放进一个模块，并使用一个导入此模块的引导脚本可能会减少脚本的启动时间。也可以在命令行中直接命名 .pyc 或 .pyo 文件。

同一模块还可以只有 spam.pyc 文件（或者 spam.pyo ，当使用 *-O* 时）而没有 spam.py 文件。这可以作为发布 Python 代码库的形式，让反向工程者稍微难过点~

compileall 模块可以将同一目录的所有模块编译成 .pyc 文件（或者 .pyo 文件，当使用 *-O* 时）。

6.2. Standard Modules 标准模块

伴随 Python 发布版包含一个标准模块库，在独立的 Python 库参考文档中描述（即后面的“库参考文档”）。有些模块内置在解释器里，这些或者为效率或者为支持类似系统调用等系统级访问控制提供了操作入口，虽然他们不是语言核心的组成部分但仍然是内置的。这些模块会根据底层平台进行不同的选择配置，比如 winreg 模块只在 Windows 系统上提供。有一个特别的模块应该注意：sys ，它内置在每个 Python 解释器中。变量 sys.ps1 和 sys.ps2 分别定义了主提示符和次提示符使用的字符串。

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

这两个变量仅在解释器处于交互模式下定义。(译著：仅在 Python 命令行有效，在 Python IDLE 中无效)

变量 sys.path 是一个字符串列表，决定了解释器搜索模块的路径。如果定义了环境变

量 **PYTHONPATH** ，它将以此初始化默认路径，否则就使用一个内置的默认值。你可以使用标准的列表操作符来修改这个值。

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

6.3. dir() 函数

内置函数 dir() 用来查找模块中定义的名称。它返回一个排序后的字符串列表。

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

如果没有指定参数，函数 dir() 会列出当前你已经定义的名称。

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

注意：它会把所有类型的名称都列出来：变量，模块，函数等。

dir() 函数并不会列出内置变量和函数的名称。如果你想这么做，它们被定义在 builtins 标准模块。

```
>>>import builtins
>>>dir(builtins)

['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'Buffer
Error', 'BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'Environme
ntError', 'Exception', 'False', 'FloatingPointError', 'FutureWarning', 'Generato
rExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexErr
or', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError',
 'None', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'P
endingDeprecationWarning', 'ReferenceError', 'RuntimeError', 'RuntimeWarning', '
StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'Ta
```

```
bError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'Unicod
eEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserW
arning', 'ValueError', 'Warning', 'ZeroDivisionError', '__build_class__', '__deb
ug__', '__doc__', '__import__', '__name__', '__package__', 'abs', 'all', 'any',
'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'chr', 'classmethod', 'compile', '
complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate
', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr',
'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance',
 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memory
view', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property'
, 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sort
ed', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

6.4. 包

包是通过使用“点模块名称”创建 Python 模块命名空间的一种方法。列如，模块名称 `A.B` 表示一个在名为 `A` 的包下的名为 `B` 的子模块。就像使用模块让不同模块的作者无需担心彼此全局变量名称（冲突）一样，点模块名称让多模块包的作者无需担心彼此的模块名称（冲突），像 NumPy 或 Python 图像库。

假设，你为了统一处理声音文件和数据想要设计一套模块（一个“包”）。现在有很多不同的声音文件格式，例如： `.wav` ， `.aiff` ， `.au` ，通常以他们的扩展名区分。所以为了在不同文件格式间转换，你可能需要创建并管理一个不断增长的模块集。还可能有很多你想用来处理声音数据的不同操作，比如：混音，添加回声，应用均衡器，创建人造立体声等。因此，为了进行这些操作你还需要另外编写一个永远也完不成的模块。这里是你的包的一种可能的结构（用层级文件系统表示）。

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

```
                equalizer.py
                vocoder.py
                karaoke.py
                ...
```

当导入这个包时，Python 通过 sys.path 搜索路径查找包含这个包的子目录。

为了让 Python 将目录当做内容包，目录中必须包含 __init__.py 文件。这是为了避免一个含有烂俗名字的目录无意中隐藏了稍后在模块搜索路径中出现的有效模块，比如 string 。最简单的情况下，只需要一个空的 __init__.py 文件即可。当然它也可以执行包的初始化代码，或者定义稍后介绍的 __all__ 变量。

用户可以每次只导入包里的特定模块，例如：

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

这回加载 sound.effects.echo 子模块。你必须使用它的全名来引用。

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

导入子模块的一种可替换的方法：

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

这同样会加载 echo 子模块，并使它无需使用包前缀即可访问，如下所示：

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

还有另一种变化就是可以直接导入想要的变量或函数：

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

再次，这会加载 echo 子模块，但这使它的函数 echofilter() 可以直接使用。

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

注意：当使用 from package import item 时， *item* 既可以是包的子模块（或子包），也可以是包中定义的其他名称，比如函数、类或变量。 import 语句首先检查包中是否定义了 *item* ，如果没有它就假设这是一个模块并尝试加载它。如果没有找到这个模块，就会抛出一个 ImportError 异常。

相反的，当使用 import item.subitem.subsubitem 语法时，除了最后一项外所有的项必须是一个包。最后一项可以是一个模块或包，但不能是其前面项定义的一个类、函数或变量。

6.4.1. 从一个包中导入 *

那么，当用户写下 from sound.effects import * 时会发生什么呢？理论上，人们期望这会以某种方式友好的对待文件系统，找出包内出现的所有子模块并把它们全部导入。不幸的是，这个操作在 Windows 平台上不能很好的工作，因为它的文件系统总是没有关于文件名大小写的精确信息！（即不区分文件名大小写）在这类平台上，没有一种明确的途径知道在导入 ECHO.PY 时是否应该被当做一个 echo ， Echo 或者 ECHO 模块导入。（比如，Windows 95就有把所有显示的文件名第一个字母大写烦人习惯。） DOS 系统的8+3文件名约束给长模块名带来了另一个有趣的问题。

唯一的解决办法就是由包作者为包提供一个精确的索引。导入语句会使用如下的转换规则：如果包文件 __init__.py 代码中定义了一个名为 __all__ 的列表，当使用 from package import * 时它被当做应该被导入的模块名称的列表。当发布包的一个新版本时，应该由包作

者负责保持这个列表是最新的。如果包作者在包中没有发现导入*的用法，那么他们也可以决定不去做这些。例如， sounds/effects/__init__.py 文件可以包含如下代码：

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

这意味着 from sound.effects import * 将会从 sound 包中导入三个指定的子模块。

如果没有定义 __all__ ， from sound.effects import * 语句 不会 从 sound.effects 包中将其所有的子模块导入到当前命名空间。它只会确定 sound.effects 包已经被导入（可能是通过执行 __init__.py 文件中的某段初始化代码），然后导入包中定义的任何名称。这会引入 __init__.py 文件中定义的所有名称（及其明确加载的模块）。它同样会引入通过之前导入语句明确加载的包中的所有模块。思考以下代码：

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

在这个示例中， echo 和 surround 模块被导入到当前命名空间，因为当 from...import 语句执行时它们就通过 sound.effects` 包被定义。（同样可以通过定义 __all__ 来做到这一点。）

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

记住，使用 from package import specific_submodule 永远不会有错！事实上，这是被推荐使用的方法，除非在不同包中导入的模块使用了相同的名字。

6.4.2. 包内引用

当包被组织成子包时（就像示例中的 sound 包），你可以使用绝对导入来引用兄弟包的子模块。例如，如果 sound.filter.vocoder 模块需要使用 sound.effects 包中的 echo 模块，可以使用 from sound.effects import echo 。

你仍然可以通过 from module import name 导入语句方式使用相对导入。在相对导入中，通过使用前导点号来表示是从当前包或是父包中导入。列如，从 surround 包中你可以使用：

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

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

6.4.3. 跨目录的包

包还支持一个特殊的属性： __path__ 。这会在那个文件中的代码执行前被初始化为一个包含包文件 __init__.py 的目录名字的列表。你可以修改这个变量，用来影响对包中包含的子包和模块的搜索。

尽管这个特性并不常用，但它可以用来扩展一个包中的模块集合。

Footnotes

[1] 实际上，函数定义也是 *可执行* 的 *语句* ，执行环境从模块的全局符号表查找函数名。

7. 输入输出

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.

一个程序可以有几种输出方式：以人类可读的方式打印数据，或者写入一个文件供以后使用。本章将讨论几种可能性。

7.1. 格式化输出

到目前为止，我们已经解除了两种输出值的方法： *表达式语句* 和 print() 函数。（第三种方式是使用文件对象的 write() 方法，标准文件输出可以参考 sys.stdout 库手册。）

通常，你想要对输出做更多的格式控制，而不是简单的打印使用空格分隔的值。有两种方法可以格式化你的输出：第一种方法是由你自己处理整个字符串，通过使用字符串切割和连接操作可以创建任何你想要的输出形式。标准模块 string 包含一些将字符串填充到指定列宽度的有用操作，随后就会讨论这些。第二种方法是使用 str.format() 方法。

string 模块包含一个模版类，还支持另一种将值替换为字符串的方法。

当然，还有一个问题：你如何将值转换为字符串？幸运的是，Python 有几种方法可以将任何值转换为一个字符串：将它传递给 repr() 函数或 str() 函数。

str() 函数返回值的人类可读的表示形式，而 repr() 函数则生成可以由解释器读取的表现形式（如果没有相等的语法则产生一个 SyntaxError 错误）。对于那些没有一个特别的可供人类使用的表示形式的对象， str() 和 repr() 将返回同样的值。大多数值在使用两个函数时都会有相同的表现形式，像数字或者类似列表和字典的结构。特别的，字符串和浮点数具有两种独特的表现形式。

下面是一些示例：

```
>>>s = 'Hello, world.'
>>>str(s)
'Hello, world.'
>>>repr(s)
"'Hello, world.'"
>>>str(0.1)
'0.1'
>>>repr(0.1)
'0.10000000000000001'
>>>x = 10 * 3.25
>>>y = 200 * 200
>>>s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>>print(s)
The value of x is 32.5, and y is 40000...
>>># The repr() of a string adds string quotes and backslashes:
...hello = 'hello, world\n'
>>>hellos = repr(hello)
>>>print(hellos)
'hello, world\n'
```

```
>>># The argument to repr() may be any Python object:
...repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

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

这里是两种输出平方和立方表格的方法：

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

>>>for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

（注意：在第一个例子中使用 `print()` 函数时在每列之间添加了一个空格——它总是会在参数之间添加空格。）

这个示例演示了字符串对象的 `rjust()` 方法，它通过在字符串左侧填充指定宽度的空格以使其右对齐。还有两个类似的方法： `ljust()` 和 `center()` 。这些方法不输出任何东西，它们仅仅返回一个新字符串。如果输入的字符串过长，它们不会截短它，而是原样返回。这也许会使你的列布局混乱，但总比截短它更好，那样会输出错误的值。（如果你确实想要截短，你总是可以使用切片操作，如 `x.ljust(n)[:n]` 。）

还有另一个方法： `zfill()` ，使用零在数字字符串左侧填充（到指定宽度）。它可以理解

正负号。

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

str.format() 方法基本的用法类似这样：

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

括号及其包含的字符（称作格式化域）会被传递给格式化方法的对象替换。括号中的数字指代传递给格式化方法对象的位置。

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

如果在格式化方法中使用了关键字参数，它们的值可以通过参数名指代。

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

位置参数和关键字参数可以随意混合使用：

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

字段名称后面可以跟一个可选的 ':' 符号和格式化分类符。这也是如何更好的控制格式化值的方法。下面的示例将 PI 小数点后截短为三位。

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

通过在 ':' 后面传递一个整数可以限定那个字段的最小字符宽度。这对美化表格非常有用。

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

如果你有一个不想分割的确实很长的字符串，使用名称替代位置来引用被格式化的变量将会更好。这可以简单的通过传递一个字典并且使用方括号 '[]' 访问所有的键。

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

这同样可以使用“**”表示法将 table 变量当作关键字参数传递做到（即对字典进行拆分）。

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

这在与新的内置函数 vars() 结合时尤为有用，它返回一个包含所有本地变量的字典。

字符串格式化方法 str.format() 的完整介绍请参考 *Format String Syntax* 。

7.1.1. 旧式字符窜格式化

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

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

因为 str.format() 方法十分新颖，大多数 Python 代码仍然使用 % 操作。然后，因为这种旧式风格的格式化最终将从 Python 中废除，一般应该使用 str.format() 方法。

更多信息可以参考 *Old String Formatting Operations* 章节。

7.2. 文件读写

open() 函数返回一个文件对象，并且它通常带有两个参数： open(filename, mode) 。

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

第一个参数是一个包含文件名的字符串。第二个参数是另一个包含几个字符的字符串，描述了文件使用的方式。当文件为只读时 *mode* 取值为 'r' ，只写时为 'w' （如果存在同名的文件则会被覆盖）； 'a' 表示以追加方式打开文件，所有的数据都会被自动的添加到文件末尾。 'r+' 表示以读写方式打开文件。参数 *mode* 是可选的，默认值为 'r' （即只读）。

通常，文件是以 *text mode* （文本模式）方式打开，即你从文件中读写字符串都是以一种特殊编码（默认为 UTF-8）进行编码的。可以通过在常用模式后添加 'b' 选项从而以 *binary mode* （二进制模式）打开文件，现在数据就是以字节码对象形式来读写了。这种模式可以用在所有非文本文件中。

在文本模式中，在读取文件时默认会将特定平台的行结束符（Unix 为 \n ，Windows 为 \r\n ）转换为 \n ，并在写入文件时转换回去。这种对文件数据的后台修改是对文本文件是无害的，但它会破坏像 JPEG 或 EXE 文件中的二进制数据。在读写这类文件时，你应该十分小心的使用二进制模式。

7.2.1. 文件对象的方法

本节中剩下的示例都假设已经创建了一个名为 f 的文件对象。

想要读取一个文件的内容，可以调用 f.read(size) 方法，该方法读取若干数量的数据并以字符串或字节码对象形式返回。 *size* 是一个可选的数值参数。如果没有指定 *size* 或者它为负数，就会读取并返回文件的全部内容。如果文件大小是你机器内容的两倍大时，这会让你面临问题。不过，你应该以尽可能大字节读取文件内容。如果已经达到文件尾， f.read() 将返回一个空字符串（ '' ）。

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

f.readline() 方法从文件中读取单独一行内容，字符串结尾会带有一个换行符（ \n ），并且仅当文件最后一行结尾没有换行符时才会被省略。这样就让返回值清晰明了了，如果 f.readline() 返回一个空字符串就表示到达文件尾了。当用 '\n' 表示一个空行时，将返回一个只含有一个换行符的字符串。

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

f.readlines() 方法返回一个包含文件所有数据行的列表。如果指定了可选的 *sizehint* 参数，它就会从文件中读取至少包含指定字节数的完整行并返回之。这通常被用来从一个大文件中以行的形式高效的读取内容，而不是将整个文件加载到内存。此方法只返回完整的行。

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

一种替代的方法是通过遍历文件对象来读取文件行。这是一种内存高效、快速，并且代码简介的方式：

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

虽然这种替代方法更简单，但并不具备细节控制能力。因为这两种方法处理行缓存的方式不同，千万不能搞混。

f.write(string) 方法将 *string* 的内容写入文件，并返回写入字符的长度。

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

想要写入其他非字符串内容，首先要将它转换为字符串。

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

f.tell() 方法返回一个指代文件对象当前位置的整数，表示从文件开头到当前位置的字节数。想要改变文件对象位置，你可以使用 f.seek(offset, from_what) 方法。新的位置是通过将 *offset* 值与参考点相加计算得来的，参考点是由 *from_what* 参数确定的。如果 *from_what* 值为0则代表从文件头开始计算，值为1时代表从当前文件位置开始计算，值为2时代表从文件尾开始计算。 *from_what* 参数可以省略并且其默认值为0，即使用文件头作为参考点。

```
>>>f = open('/tmp/workfile', 'rb+')
>>>f.write(b'0123456789abcdef')
16
>>>f.seek(5)       # Go to the 6th byte in the file
5
>>>f.read(1)
b'5'
>>>f.seek(-3, 2) # Go to the 3rd byte before the end
13
>>>f.read(1)
b'd'
```

在文本文件中（那些没有使用 b 模式选项打开的文件），只允许从文件头开始计算相对位置（使用 seek(0, 2) 从文件尾计算时就会引发异常）。

当你使用完一个文件时，调用 f.close() 方法就可以关闭它并释放其占用的所有系统资源。在调用 f.close() 方法后，试图再次使用文件对象将会自动失败。

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

当处理文件对象时，使用 with 关键字是个好习惯。这是由好处的：在文件一系列操作后它会被适当的关闭，甚至从某种意义上说异常也会变的可爱。

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

文件对象还有一些其他的方法，像 isatty`和:meth:`truncate() ，但它们都不常用。关于文件对象完整示例请查阅库参考手册。

7.2.2. `pickle`模块

从文件中可以很容易的读写字符串。数字可能需要额外的处理，因为 read() 方法只能

返回字符串。必须将它传递给像 in() 一样的函数，此类函数接收一个字符串并返回它数值，像 '123' 就返回123。然而，当你想保存更复杂的数据类型时事情会变的尤为复杂，像列表、字典或者对象实例等。

好在用户无需经常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。这是一个惊艳的模块，它几乎可以将任何 Python 对象（甚至是一些 Python 代码！）转换为字符串表示形式，这个过程称为 *pickling* （封装）。从这个字符串表示形式中重建 Python 对象被称为 *unpickling* （拆封）。在 *pickling* 和 *unpickling* 之间，字符串表示的对象可以存储在文件或数据中，或者可以通过网络连接发送给远程的机器。

如果你有一个对象 x 对象和一个已经打开并等待写入的文件对象 f ，封装（pickling）对象的最简单方法之需要一行代码：

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

想要拆封（unpickling）这个对象，如果存在一个已经打开并等待读取的文件对象 f ，即可：

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

（当封装许多对象时，或者你不想把封装数据写入一个文件，这可以有其他的变化。请查阅 Python 库文档获取关于 pickle 完整的文档。）

pickle 模块将 Python 对象转换为可以存储并提供给其他程序或其自身以后调用的标准方法，术语称为 *persistent* 对象（持久化对象）。pickle 模块被如此广泛的使用，许多 Python 扩展开发者都非常注意像矩阵这样的新数据类型是否可以被适当的封装和拆封。

8. 错误和异常

直到现在，我们仍未提及错误信息，但如果你试验过前面的例子，大概你已经看到过一些。这里有（至少）两种不同类型的错误： *语法错误* 和 *异常* 。

8.1. 语法错误

语法错误，也被称作解析错误，也许是你学习 Python 过程中最常见抱怨。

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

语法分析器指出错误行，并且在检测到错误的位置前面显示一个小“箭头”。错误是由箭头 *前面* 的标记引起的（或者至少是这么检测的）：这个例子中，函数 print() 被发现存在错误，因为它前面少了一个冒号（ ':' ）。错误会输出文件名和行号，所以如果是从脚本输入的你就知道去哪里检查错误了。

8.2. 异常

即使一条语句或表达式在语法上是正确的，当试图执行它时也可能会引发错误。运行期检测到的错误称为 *异常* ，并且程序不会无条件的崩溃：很快，你将学到如何在 Python 程

序中处理它们。然而，大多数异常都不会被程序处理，像这里展示的一样最终会产生一个错误信息：

```
>>>10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: int 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: Can't convert 'int' object to str implicitly
```

错误信息最后一行指出到底发生了什么。异常是以不同的类型出现的，并且类型也被当做信息的一部分打印出来：示例中包含 ZeroDivisionError ， NameError 和 TypeError 类型。异常发生时打印的异常类型字符串就是内置异常的名称。这对内置异常是正确的结果，但对用户自定义异常（尽管这是一个有用的规范）就不可而知了。标准异常的名称都是内置的标识符（不是保留关键字）。

这行其他的部分提供了异常类型和是引发异常相关的详细信息。

异常信息的前面部分以调用堆栈的形式显示了异常发生的上下文。通常，它包含一个调用堆栈的源代码行的清单，但从标准输入读取的行不会被显示。

:ref:`bltin-exceptions`列出了内置异常及其用途。

8.3. 异常处理

通过编程处理选择的异常是可行的。看一下下面的例子：它会一直要求用户输入，直到输入一个合法的整数为止，但允许用户终端这个程序（使用 Control-C 或系统支持的任何方法）。注意：用户产生的终端会引发一个 KeyboardInterrupt 异常。

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

:keyword:`try`语句按照以下方式工作：

首先，执行 *try 子句* （在关键字 try 和 except 之间的语句）。

如果没有发生异常， *异常子句* 就会被跳过， try 语句执行完毕。

如果在执行 *try* 子句时发生异常，剩下的子句就会被跳过。如果异常的类型与 except 后面的异常名称相匹配，那么 *except* 子句就会被执行，然后继续执行 try 语句后的代码。

如果一个异常没有任何 *except* 子句中的异常名称与之匹配，那么它就会被传递到上层的 try 语句。如果仍然没有找到处理这个异常的语句，它就成了一个 *未处理的异常* 并且

中断代码执行，像上面那样显示一条信息。

一个 try 语句可以有多个 *except* 子句，用来明确的处理不同的异常。至多，只有一个异常处理子句会被执行。异常处理子句只处理对应的 *try* 子句中发生的异常，而不是其他的 try 语句发生的异常。一个 *except* 子句可以通过带括号的元组（tuple）定义多个异常类型，例如：

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

最后一个 *except* 子句可以省略异常名称，以作为通配符使用。你需要慎用此法，因为它会轻易隐藏一个实际的程序错误！可以使用这种方法打印一条错误信息，然后重新抛出异常（允许调用者处理这个异常）。

```
import sys

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

try...:keyword:*except* 语句有一个可选的 *else* 子句，如果存在，它必须在所有的 *except* 子句后面。如果 *try* 子句没有抛出异常，这对那些必须执行的代码就非常有用。

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

使用 else 子句比向 try 子句添加额外的代码更好，因为它避免了意外捕获的异常，它们并不是由 try...:keyword:*except* 语句保护的代码所抛出。

当一个异常发生时，它可能含有一个相关值，就是传说中的异常 参数 。是否存在此参数及其类型取决于异常的类型。

except 子句可以在异常名称后面指定一个变量。这个变量以存储在 instance.args 中的参数形式绑定到一个异常实例。方便起见，异常实例定义了 __str__() 方法，所以这个参数可以被直接打印出来而无需通过 .args 引用。你也可以在抛出异常前先实例化它，然后给它添加任何想要的属性。

```
>>>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 be printed directly,
...                           # but may be overridden in exception subclasses
...     x, y = inst.args      # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

对于那些未处理的异常，如果一个它们带有参数，那么就会被作为异常信息的最后部分（“详情”）打印出来。

异常处理器不仅仅处理那些在 *try* 子句中立刻发生的异常，也会处理那些 *try* 子句中调用的函数内部发生的异常。例如：

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

8.4. 抛出异常

`raise` 语句允许程序员强制抛出一个指定的异常。例如：

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

`raise` 唯一的参数指定了要跑出的异常。它必须是一个异常实例，或者是异常类（继承自 `Exception` 的类）。

如果你需要确定是否抛出了一个异常而并不想去处理它，一个简单的 `raise` 语句允许你重新抛出异常。

```
>>>try:
...     raise NameError('HiThere')
...except NameError:
```

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

8.5. 用户自定义异常

通过创建一个新的异常类，程序可以命名它们自己的异常。异常应该是典型的继承自 Exception 类，通过直接或间接的方式。例如：

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

在这个列子中， Exception 的默认方法 __init__() 被覆盖。新的行为只是简单的创建 *value* 属性。这替换了创建 *args* 属性的默认行为。

异常类可以定义成像其他类一样做任何事情，但一般都会保持简洁，通常只提供一些属性以便于异常处理器获取错误相关信息。当创建一个可以产生几种不同错误的模块时，一种通常的做法是为模块中定义的异常创建一个基础类，然后为不同的错误情况创建特定的子异常类。

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

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

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

```
        """

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

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

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

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

大多数异常的名字都是以“Error”结尾，类似标准异常命名。

许多标准模块定义它们自己的异常来报告那些可能发生在函数中错误。更多关于类的信息请参考 *Classes 类* 章节。

8.6. 定义清理动作

try 语句有另外一个可选的子句，可以用来定义那些在所有情况下必须执行的清理动作。例如：

```
>>>try:
...     raise KeyboardInterrupt
...finally:
...     print('Goodbye, world!')
...
Goodbye, world!
KeyboardInterrupt
```

无论是否发生异常，一个 *finally* 子句在离开 try 语句前总是会被执行。当在 try 子句中发生一个异常并且没有被 except 字句处理时（或者发生在 except 或 else 子句中），它将会在 finally 子句执行完后被重新抛出。即使通过 break ， continue 或者 return 等其他任何子句，当要离开 :keywor:`try` 语句时 finally 子句也会被执行。一个稍微复杂的列子：

```
>>>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 2.0
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'
```

如你所见，在任何事件下 finally 子句都会执行。通过将两个字符串相除引发的 TypeError 异常并没有被 except 字句处理，因此在 finally 子句执行后被重新抛出。

在真实的应用程式中，使用 finally 子句释放额外的资源（像文件或网络连接）是很棒的选择，不论对资源的操作是否成功。

8.7. 预定义的清理动作

一些对象定义了标准的清理动作，无论使用对象的操作是否成功，当对象不再需要时它们就在后台被调用执行。下面的例子尝试打开一个文件并将其内容打印在屏幕上。

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

这段代码的问题是当代码执行完毕后不知会过多久它才会关闭文件。这在简单的脚本中还不构成额外难题，但在大的应用程式中问题就严重了。 with 语句保证像文件这样的对象在使用完之后总是可以被立即正确的清理。

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

在这段语句执行后，文件 *f* 总是被关闭，甚至在处理文件行时遇到了问题。诸如文件之类的对象在它们的文档中会指出是否提供了预定义清理动作。

9. 类

Python 的类机制通过最小的新语法和语义在语言中实现了类。它是 C++何 Modula-3 语言中类机制的混合。就像模块一样，Python 的类并没有在用户和定义之间设立绝对的屏障，而是依赖于用户不去“强行闯入定义”的优雅。另一方面，类的大多数重要特性都被完整

的保留下来：类继承机制允许多重继承，派生类可以覆盖（override）基类中的任何方法或类，可以使用相同的方法名称调用基类的方法。对象可以包含任意数量的私有数据。

用在 C++中的术语讲，普通的类成员（包括数据成员）都是 *公有* 的（public）（除了下面提到的 *Private Variables 私有变量* ），并且所有的成员函数都是 *虚* 的（virtual）。类并没有特殊的构造器和析构器。和在 Modula-3中一样，从方法中没有什么简洁的方式可以引用其对象成员，函数方法必须以代表对象的标识符（self）作为第一个明确的参数，在调用时被隐式的提供。和在 Smalltalk 中一样，类本身就是对象，从更广发的意义上讲：在 Python 中，所有的数据类型都是对象。这就为导入和重命名提供了支持。不似 C++和 Modula-3那样，内置类型可以被用户用作基类进行扩展。并且像在 C++中一样，而不是 Modula-3，所有内置带有特殊语法的操作符（算术操作符，下标操作符等）都可以针对类的实例进行重定义。

9.1. 术语相关

关于类因为缺少普遍的可以接受的术语，我暂时借用 Smalltalk 和 C++中的术语。（我更想使用 Modula-3的术语，因为它的面向对象机制比 C++更接近 Python，但我想几乎没人听说过它）。

对象具有特性，并且多个名称（在多个作用于中）可以绑定在同一个对象上。这在其它语言中被称为别名。在对 Python 的第一印象中这通常会被忽略，并且当处理不可变基础类型（数字，字符串，元组）时可以被放心的忽略。但是，在调用列表、字典这类可变对象，或者大多数程序外部类型（文件，窗体等）描述实体时，别名对 Python 代码的语义便具有（有意而为!）影响。这通常有助于程序的优化，因为在某些方面别名表现的就像是指针。例如，你可以轻易的传递一个对象，因为通过继承只是传递一个指针。并且如果一个方法修改了一个作为参数传递的对象，调用者可以接收这一变化——这消除了两种不同的参数传递机制的需要，像 Pascal 语言。

9.2. 作用域和命名空间

在介绍类（class）之前，我必须先告诉你一些 Python 作用域的规则。类定义非常巧妙的运用了命名空间，并且要想探求究竟你需要知道作用域和命名空间是如何工作的。顺便说一句，这个主题的相关知识对任何高级 Python 程序员非常重要。

让我们从一些定义说起。

命名空间 就是一个从名称到对象的映射。大多数命名空间目前都被实现为 Python 字典，但那通常不会被注意（除非为了性能考虑），并且在将来它可能会被改变。命名空间的一些实例：内置名称集（函数，像 `abs()` ，和内置异常名称），一个模块中的全局名称，函数调用时的局部名称。某种意义上讲，一个对象的属性集合也构成了一个命名空间。关于命名空间需要了解的一件重要的事情就是在不同命名空间中的名称之间没有任何的关系。比如，两个不同的模块可能都定义了一个“maximize”函数而不会混淆——用户必须使用模块名为前缀（来使用模块）。

顺便说一下，在 Python 中我习惯将任何跟在一个点（.）后的名称叫作 *属性* 。列如，在 `z.real` 表达式中， `real` 就是对象 `z` 的一个属性。严格来讲，从模块中引用名称就是引用其属性：在 `modname.funcname` 表达式中， `modname` 是一个模块对象，同时 `funcname` 就是它的一个属性。以此而论，在模块属性和模块内部定义的全局名称之间恰好是直接的映射：它们共享相同的命名空间！[#]_

属性可以是只读的或可写的。在后一种情况下（指可写），允许对属性赋值。模块属性

都是可写的：你可以这样使用 `modname.the_answer = 42` 。可写属性也可以使用 `del` 语句删除。例如， `del modname.the_answer` 就会从名为 `modname` 的对象中删除 `the_answer` 属性。

命名空间是在不同的时刻创建的，并且具有不同的生命周期。包含内置名称的命名空间在 Python 解释器启动时即被创建，并且从不会被删除。模块的全局命名空间在模块定义被读取时即被创建，通常模块的命名空间会一直保存到解释器退出。通过解释器顶层调用执行的语句，不论是交互的还是从脚本文件读取的，都被认为是 `__main__` 模块的一部分，因此他们也有自己的全局命名空间。（内置名称实际上也存在于一个模块，称为： `builtins` 。）

当调用函数时，就会为它创建一个局部命名空间，并且在函数返回或抛出一个并没有在函数内部处理的异常时被删除。（实际上，用遗忘来形容到底发生了什么更为贴切。）当然，每个递归调用都有自己的局部命名空间。

作用域 就是一个 Python 程序可以直接访问命名空间的正文区域。这里的 *直接访问* 意思是一个对名称的错误引用会尝试在命名空间内查找。

尽管作用域都是静态定义，但它们都被动态的使用。在执行的任何时刻，至少有三个命名空间可以直接访问的嵌套作用域：最先被查找的最内层作用域，包含局部名称；所有封闭函数的命名空间被从最近的封闭作用域内开始搜索；其次被查找的中间层作用域，包含当前模块的全局名称；最后被查找的最外层作用域，包含内建名称的命名空间。

如果一个名称被声明为全局的，那么所有的赋值和引用都会直接从包含模块全局名称的中层作用域开始。那些在最内层作用域以外的变量，可以使用 `nonlocal` 语句重新绑定，如果没有声明为 *nonlocal* 它们则是只读的。（试图改变这样的变量只会在最内层作用域中简单的创建一个 *新* 的局部变量，而外部那个相同标识符的变量不会改变）

通常，局部作用域引用当前函数（正文的）的局部名称。在函数外部，局部作用域将这一命名空间看做全局作用域：模块命名空间。类定义也会在局部作用域引入另一个命名空间。

重要的是要理解作用域是由正文确定：在模块中定义的函数的全局作用域是该模块的命名空间，而不论从何处或者通过什么别名调用函数。换句话说，对名称的实际搜索是在运行时动态完成的。然而，Python 语言的定义正朝着“编译”时静态名称确定进化，因此不要依赖动态名称确定！（事实上，局部变量已经是静态确定的。）

Python 的一个特别之处就是对名称的赋值总是在最内层作用域内，当然含有 `global` 或 `nonlocal` 语句的除外。赋值操作并不会拷贝数据——他们只是将名称绑定到对象。删除操作亦是如此：语句 `del x` 从局部作用域引用的命名空间中删除名称``x``的绑定。实际上，所有引入新名称的操作都是用局部作用域：特别是 *import* 语句和函数定义语句，它们将模块或函数名称绑定在局部作用域。（可以用 `global` 语句指明某个特定的变量为全局作用域。）

`global` 语句用以指明某个特定的变量为全局作用域，并重新绑定它。 `nonlocal` 语句用以指明某个特定的变量为封闭作用域，并重新绑定它。

9.2.1. 作用域和命名空间示例

以下是一个示例，演示了如何引用不同作用域和命名空间，以及 `global` 和 `nonlocal` 如何影响变量绑定：

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

```
            global spam
            spam = "global spam"

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

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

以上示例代码的输出为：

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

注意： *local* 赋值语句是无法改变 *scope_test* 的 *spam* 绑定。 nonlocal 赋值语句改变了 *scope_test* 的 *spam* 绑定，并且 global 赋值语句从模块级改变了 *spam* 绑定。

你也可以看到在 global 赋值语句之前对 *spam* 是没有预先绑定的。

9.3. 初识类

类引入了一些新语法：三种新的对象类型和一些新的语义。

9.3.1. 类定义语法

类定义最简单的形式如下：

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

类定义，就像函数定义（ def 语句）一样，必须先执行才能生效。（你当然可以在一条 if 语句分支或一个函数内部定义一个类。）

实践中，类定义内部的语句通常都是函数定义，但也允许包含其他语句，有时这非常有用——稍后我们将对此介绍。类内部的函数定义通常有一个特殊形式的参数列表，用于方法调用约定——稍后我们也将对此介绍。

当进入类定义时，就会创建一个新的命名空间，并且用作局部作用域。因此，所有对局部变量的赋值都会在这个新命名空间内进行。特别的，函数定义就是将新函数的名称绑定在此。

当类定义完成时（正常结束），就创建了一个 *类对象* 。这是一个在类定义创建的命名空间内容周围的基本包装，我们将在下一节中学习更多关于类对象的知识。原来的局部（在进入类定义之前生效的那个）作用域得以恢复，并且类对象在这被绑定到类定义头部指定的类名称（参考 ClassName 示例）。

9.3.2. 类对象

类对象支持两种操作：属性引用和实例化。

属性引用 使用在 Python 中所有属性引用一样的标准语法： obj.name 。当创建类对象时，所有在类命名空间中的名称都是有效的属性名。因此，如果定义一个这样的类：

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

那么， MyClass.i 和 MyClass.f 都是有效的属性引用，分别反馈一个整数和一个函数对象。也可以对类属性进行赋值，所以你可以通过给 MyClass.i 赋值来修改它。 __doc__ 也是一个有效的属性，返回类的 *docstring* （文档字符串）： "A simple example class" 。

类的 *实例化* 使用函数（调用）表示法。只要将类对象看作是一个返回新的类实例的无参函数。例如（假设实例化上面的类）

```
x = MyClass()
```

创建一个类的新 *实例* ，并将其赋值给一个局部变量 x 。

这个实例化操作（“调用”一个类对象）会将建一个空对象。很多类都喜欢创建含有特别的自定义初始化状态的实例对象。因而，你可以定义一个包含特殊方法 __init__() 的类，像下面这样：

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

当一个类定义了 __init__() 方法时，类实例化会为新创建的类实例自动调用 __init__() 方法。所以，在这个例子中，可以获得一个初始化的新实例：

```
x = MyClass()
```

当然，为了更好的灵活性 __init__() 方法可以包含参数。如果那样的话，类实例化操作时给出的参数都会传递给 __init__() 方法。例如：

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

9.3.3. 实例对象

现在，我们可以用实例对象做什么呢？实例对象唯一可用的操作就是属性引用。这里有

两种有效的属性名称：数据属性（字段）和方法。

数据属性 相当于 Smalltalk 中的“实例变量”，或者 C++中的“数据成员”。数据属性无需声明。和局部变量一样，它们在第一次赋值时就会产生。例如：如果 x 是上面创建的类 MyClass 的实例，那么下面的代码片段将会打印出 16 这个值而没有任何错误。

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

另外一种实例属性引用就是 *方法* 。方法即使“属于”某个对象的函数。（在 Python 中，术语“方法”不仅存在于类实例，其他对象类型也包含方法。例如，列表对象含有 append，insert，remove，sort 等方法。然而，在下面的讨论中，除非特别说明，我们所用得术语“方法”专指类实例对象的方法。）

一个实例对象的方法名是否有效取决于它的类。按照定义，一个类中所有函数对象定义与它的实例方法是相对应的。因此在我们的例子中， x.f 是一个有效的方法引用，因为 MyClass.f 是一个方法；但是 x.i 则不是，因为 MyClass.i 不是一个方法。然而 x.f 与 MyClass.f 并不是相同的东西——它是一个 *方法对象* ，而非函数对象。

9.3.4. 方法对象

通常，方法通过右绑定方式调用。

```
x.f()
```

在 MyClass 示例中，这将会返回 'hello world' 字符串。然而，你无需立刻就调用一个方法： x.f 是一个方法对象，并且它可以被保存起来以便稍后调用。比如：

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

会不断打印 hello world 字符串，直到程序终止。

当调用一个方法是具体做了什么呢？你可能已经注意到上面我们调用 x.f() 时并没有使用参数，尽管在 f() 的定义中指定了一个参数。这个参数怎么了呢？当然，当不使用任何参数调用需要一个参数的函数时，Python 就会抛出一个异常——即使这个参数没有实际使用…

实际上，你可能已经猜到了答案：方法有一个特性就是实例对象被当做第一个参数传递给了函数。在我们的例子中， x.f() 调用实际上等价于 MyClass.f(x) 调用。通常，使用包含 *n* 个参数的列表调用一个方法，相当于使用通过将方法的对象插入到参数列表第一个参数前面后创建的参数列表调用相应的函数。

如果你还不理解方法是如何工作的，了解一下它的实现可能会明白真相。当引用一个非数据属性的实例属性时，就会搜索它的类。如果这个名称表示一个有效的函数对象类属性，就会将实例对象和函数对象封装（用指针指向）进一个抽象对象，从而创建一个方法对象：这就是方法对象。当使用一个参数列表调用方法对象时，它会被重新拆封，用实例对象和原始参数列表构造一个新的参数列表，然后用这个新的参数列表调用函数对象。

9.4. 一些说明

数据属性会覆盖同名的方法属性。为了避免意外的名称冲突，这在大型程序中是极难发现的 Bug，使用一些约定来减少冲突的机会是明智的。可能的约定包括：大写方法名称的首字母，使用一个唯一的小字符串（也许只是一个下划线）作为数据属性名称的前缀，或者方法使用动词而数据属性使用名词。

数据属性可以被方法引用，也可以由一个对象的普通用户（客户）使用。换句话说，类不能用来实现纯净的数据类型。事实上，Python 中不可能强制隐藏数据——一切基于约定。（如果需要，使用 C 编写的 Python 实现可以完全隐藏实现细节并控制对象的访问。这可以用来通过 C 语言扩展 Python。）

客户应该谨慎的使用数据属性——客户可能通过践踏他们的数据属性而使那些由方法维护的常量变得混乱。注意：只要能避免冲突，客户可以向一个实例对象添加他们自己的数据属性，而不会影响方法的正确性——再次强调，命名约定可以避免很多麻烦。

从方法内部引用数据属性（或其他方法）并没有快捷方式。我觉得这实际上增加了方法的可读性：当浏览一个方法时，在局部变量和实例变量之间不会出现令人费解的情况。

一般，方法的第一个参数被命名为 `self` 。这仅仅是一个约定：对 Python 而言，名称 `self` 绝对没有任何特殊含义。（但是请注意：如果不遵循这个约定，对其他的 Python 程序员而言你的代码可读性就会变差，而且有些 *类查看器* 程序也可能是遵循此约定编写的。）

类属性的任何函数对象都为那个类的实例定义了一个方法。函数定义代码不一定非得定义在类中：也可以将一个函数对象赋值给类中的一个局部变量。例如：

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'hello world'
    h = g
```

现在， `f` ， `g` 和 `h` 都是指向函数对象的类 `C` 的属性，因此它们都是 `C` 实例的方法——`h` 严格等于 `g` 。注意：这种习惯通常只会让程序的读者迷惑。

通过使用方法属性的 `self` 参数，方法可以调用其他方法。

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

方法可以像普通函数一样引用全局名称。方法可见的全局作用域是包含此类定义的模块。（类自身永远不会作为全局作用域！）虽然在方法中使用全局数据只有极少好的理由，还是有很多合法的用法使用全局作用域：首先，在全局作用域中导入的模块和方法可以被方法

使用，也可以调用其中定义的类和函数。通常，包含此方法的类也会在这个全局作用域中被定义。在下一节中，我们会了解为何一个方法想要引用自己的类！

Python 中，每个值都是一个对象，并具有一个*类*（称作它的*类型*）。这被存储为 ``object.__class__``。

9.5. 继承

当然，如果一种语言不支持继承的特性，那么 *类* 就没有什么意义。派生类的定义语法如下所示：

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

名称 BaseClassName 必须定义在包含派生类定义的作用域内。在基类名称的位置，允许出现任何表达式。当基类在另一个模块中定义时，这种做法非常有用。比如：

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

派生类定义的执行过程和基类是相同的。当构造类对象时，基类就会被记住。这在解析属性引用时被使用：如果一个请求的属性在类中没有找到，就会继续在基类中查找。如果基类自身也是从其他类派生而来，这个规则就会被递归的应用。

派生类的实例化并没有什么特别之处： DerivedClassName() 调用会为类创建一个新的实例。方法引用按照如下决定：搜索相应的类属性，必要时沿着基类链逐层搜索，如果发现一个函数对象这个方法引用就是有效的。

派生类可能会覆盖它们基类的方法。因为方法没有任何特权，所以本意想要调用基类中的另一个方法时，可能会被派生类中覆盖它的方法终止。（对于 C++程序员：Python 中所有的方法实际上都是 virtual 的。）

在派生类中覆盖方法，实际上可能是想要扩展基类中同名的方法，而非简单的替代。有一个简单的方法可以直接调用基类中的方法：只需调用 BaseClassName.methodname(self, arguments) 。偶尔，这也会对客户有所帮助。（注意：只有基类在同一作用域内定义或导入时才可以这样使用。）

Python 有两个和集成相关的内置函数：

使用 isinstance() 函数可以检测一个对象的类型：仅当 obj.__class__ 是 int 或者从 int 中派生的类时， isinstance(obj, int) 才返回 True 。

使用 issubclass() 函数可以检查类的继承关系：因为 bool 是 int 的子类，所以 issubclass(bool, int) 返回 True 。但是， issubclass(float, int) 是 False ，因为 float 不是 int 的子类。

9.5.1. 多重继承

Python 也支持多重继承的形式：使用多个基类进行类定义。如下所示：

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

```
        .
        .
        <statement-N>
```

大多数情况下，最简单来讲，你可以认为对从父类继承的属性搜索遵循深度优先，从左向右，不会对重叠的相同层次的同一个类搜索两次。因此，如果一个属性在 DerivedClassName 中找不到，就会继续在 Base1 中查找，然后（递归的）在 Base1 的基类中查找。如果还没有找到，就会继续从 Base2 类中查找，依次类推。

实际上，它比以上要稍微复杂一些，解决方法就是为支持协同调用 super() 而动态改变排序。这个方法在一些其他多重继承的语言中称作 *call-next-method* ，并且要比单继承语言中的 *super* 调用更强大。

动态排序是必须的，因为多重继承中所有的情况都会呈现为一个或多个菱形关系（从最低层的类开始，至少存在一个父类可以通过多条路径访问）。比如，所有的类继承自 oject ，索引任何情况下的多重继承都存在不止一条路径可以访问 object 。想要避免基类被多次访问，动态算法在每个类中通过维护一个从左向右的特殊顺序的方式将搜索顺序线性化，从而每个父类支部调用一次，并且那是不变的（即继承一个类不会一项它父类的优先级）。总之，这些属性让使用多重继承设计可靠的和可扩展的类成为可能。更多信息请参考：http://www.python.org/download/releases/2.3/mro/ 。

9.6. 私有变量

Python 对类私有标识符提供了有限的支持。形如 __spam （至少两个下划线前缀，至多一个下划线后缀）的任何标识符都会被原文的替换成 _classname__spam 形式，这里 classname 就是当前的类名。这种变换不会关注标识符的语法位置，因此可以用来定义类私有实例和类变量、方法、全局变量，甚至把*其他*类实例变量保存为私有实例变量。当变换的名字超过255个字符时就会被截短，在类外部或者类名只有一个下划线组成时则不会被截短。

名字变换是一种实现类定义“私有”实例变量和方法的简单的途径，无需担心与派生类定义的实例变量混淆，或与类外部代码的实例变量混淆。注意：变换规则的设计主要是用来避免冲突，执意访问或修改被认为是私有的变量仍然是可行的。这在特殊环境下尤为有用，比如调试的时候，这也是为何一直没有堵上这个漏洞的原因之一。（号外：派生类和基类具有相同的名字即可使用基类的私有变量。）

注意：传递给 exec() 或 eval() 函数的代码不会将调用类的类名作为当前类。这和 global 语句的情况相似，它的作用限制于一起进行字节码编译的代码。同样的限制也适用于 getattr() ， setattr() 和 delattr() 函数，以及直接引用 __dict__ 时。

9.7. 备注

有时拥有一种类似 Pascal 的“记录（record）”或 C 的“结构（struct）”的数据类型是非常有用的，可以将一些已命名的数据项绑定在一起。定义一个空的类便可以很好的做到这点：

```
class Employee:
    pass

john = Employee() # Create an empty employee record
```

```
# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

一段需要一个特别的抽象数据类型的 Python 代码通常可以传入一个模仿那种数据类型方法的类来代替。例如，如果你有一个格式化文件对象数据的函数，你可以定义一个包含 read() 和 readline() 方法的类，它可以替代从字符串缓冲中获取数据并作为参数传递。

实例方法对象也包含属性： m.__self__ 就是方法 m() 的实例对象，并且 m.__func__ 就是方法对应的函数对象。

9.8. 异常也是类

用户自定义的异常也被当做类。利用这一原理便可以创建可扩展的异常分类。

以下是两种有效的（语义上）异常抛出形式：

```
raise Class

raise Instance
```

在第一种方式中， Class 必须是一个 type 或其派生类的实例。第一种方式为以下形式的简写：

```
raise Class()
```

在 *except* 从句中的类是与异常相兼容的，这里的异常是指同一个类或者是一个基类（但是不能反过来说—— *except* 从句中列出的派生类与基类是不兼容的）。例如，下面的代码将依次输出 B，C，D：

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

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

注意：如果 *except* 从句被颠倒了（最先使用 except B ），它只会打印 B，B，B——首先被匹配的异常被触发。

当打印一个未处理异常的错误信息时，异常的类名也会被打印出来，然后紧跟一个冒号和一个空格，最后使用内置函数 str() 将实例转换为字符串。

9.9. 迭代器

到现在，你可能已经注意到大多数容器对象都可以使用 for 语句遍历：

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

这种访问风格清晰、简洁并且方便。迭代器的使用在 Python 中非常普遍而且统一。在幕后， for 语句会对容器对象调用 iter() 函数。这个函数返回一个定义了 __next__() 方法的迭代对象，每次访问容器中的一个元素。当没有可以继续访问的元素时， :meth: *__next__* 方法会跑出一个 StopIteration 异常，这将会通知 for 语句结束循环。你可以使用内置 next() 函数调用 __next__() 方法。以下是一个完整的示例：

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

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

了解了迭代器协议背后的机制，你可以轻松的给类添加迭代器行为。定义一个 __iter__() 方法，使其返回一个带有 __next__() 方法的对象。如果这个类定义了 __next__() 方法，那么 __iter__() 方法只需要返回 self 即可。

```
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. 生成器

*Generator*s 是创建迭代器简单而又强大的工具。它们写起来就像正式的函数，但是在需要返回数据时使用 yield 语句。每次对其调用 next() 函数，生成器就会从上次脱离的位置继续（它记忆所有的数据值和最后执行的语句）。下例说明可以辩解的创建一个生成器：

```
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
```

任何可以使用生成器做的事情，也可以使用前一节介绍的基于迭代器的类来完成。生成器之所以如此简单是因为在生成器中 __iter__() 和 __next__() 方法是自动创建的。

另一个关键特性是在调用之间局部变量和执行状态都被自动的保存。这就使得函数编写更容易，并且比像 self.index 和 self.data 形式这样手动调用示例变量更清晰。

除自动创建方法和保存程序状态外，当生成器到达结尾时，它们会自动抛出 StopIteration 异常。综合来说，这些特性使得创建迭代器就像书写一个普通函数一般简单。

9.11. 生成器表达式

有些简单的生成器可以使用类似列表推导式的符号简单编码为表达式，但无需带有中括

号。这些表达式是为某种情景而设计的，在那里生成器被一个封闭函数使用。生成器表达式并完整的生成器定义更为简洁，但没有那么通用，而且比等价的列表推导式更容易记住。

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

>>>xvec = [10, 20, 30]
>>>yvec = [7, 5, 3]
>>>sum(x*y for x,y in zip(xvec, yvec))         # dot product
260

>>>from math import pi, sin
>>>sine_table = {x: sin(x*pi/180) for x in range(0, 91)}

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

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

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

Footnotes

[1] 有个例外：模块对象有一个隐秘的只读属性 __dict__ ，返回一个作为模块命名空间的字典。名称 __dict__ 是一个属性，但不是全局名称。显然，这样使用违法了命名空间实现的抽象概念，并且应该严格限制在希望在测试之后进一步研究的场合。

10. Python 标准库概览

10.1. 操作系统接口

os 模块提供了很多与操作系统交互的函数。

```
>>>import os
>>>os.system('time 0:02')
0
>>>os.getcwd()      # Return the current working directory
'C:\\Python31'
>>>os.chdir('/server/accesslogs')
```

一定要使用 import os 风格而不是 from os import * ！这会使 os.open() 函数覆盖内置的 open() 函数，因为它们的操作有很多不同。

内置函数 dir() 和 help() 对交互的使用像 os 这样的大模块非常有用。

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

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

对于日常文件和目录管理任务， shutil 模块提供了易于使用的更高层的交互。

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

10.2. 文件通配符

glob 模块提供了一个函数用来从目录通配符搜索中生产文件列表。

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

10.3. 命令行参数

常用的实用脚本通常需要处理命令行参数。这些参数以一个列表的形式存储在 sys 模块的 *argv* 属性中。例如在命令行中运行 python demo.py one two three 可以得到下面的输出：

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

getopt 模块以 Unix 的 getopt() 函数方式处理 *sys.argv* 。更多灵活有效的命令行参数处理由 optparse 模块提供。

10.4. 错误输出重定向和程序终止

sys 模块还包含 *stdin* ， *stdout* 和 *stderr* 属性。即使在 *stdout* 被重定向时，后者也可以用于显示警告和错误信息。

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

退出一个脚本最直接的方法是使用``sys.exit()``。

10.5. 字符串模式匹配

re 模块为高级字符串处理提供了正则表达式工具。正则表达式为复杂的字符串匹配和处理提供了简洁，优化的方法：

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

```
'cat in the hat'
```

当仅需简单的功能时，应该首先考虑使用字符串方法，因为他们容易阅读和调试。

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

10.6. 数学

`math` 模块为浮点数运算提供了对底层 C 函数库的访问支持。

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

`random` 模块为随机选择功能提供了工具支持。

```
>>>import random
>>>random.choice(['apple', 'pear', 'banana'])
'apple'
>>>random.sample(range(100), 10)   # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>>random.random()    # random float
0.17970987693706186
>>>random.randrange(6)    # random integer chosen from range(6)
4
```

SciPy 项目<http://scipy.org>包括很多其他数字计算模块。

10.7. 互联网访问

Python 包含许多访问互联网和处理互联网协议的模块。其中最简单的两个是通过 URL 地址获取数据的 `urllib.request` 和发送邮件的 `smtplib` 。

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

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

>>>import smtplib
>>>server = smtplib.SMTP('localhost')
>>>server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
..."""To: jcaesar@example.org
...From: soothsayer@example.org
```

```
...
...Beware the Ides of March.
..."""")
>>>server.quit()
```

（注意：第二个示例需要在本机运行邮件服务器。）

10.8. 日期和时间

datetime 模块为日期和时间处理提供了简单和复杂的类支持。由于为日期和时间的算术运算提供了支持，格式化输出和处理实现的重点就是高校的成员提取。这个模块同样支持时区处理。

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

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

10.9. 数据压缩

Python 模块还直接支持常用数数据打包和压缩格式，包括：zlib ， gzip ， bz2 ， zipfile 和 tarfile 等。

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

10.10. 性能评测

有些 Python 开发对处理同一问题的不同方法之间的性能差异抱有浓厚的兴趣。Python 提供了一个测试工具可以立即找到这些问题的答案。

例如，使用元组的封装和拆封特性代替传统的方法交换参数是很诱人的。 timeit 模块快速的演示了这一微小性能优势：

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

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

10.11. 质量控制

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

doctest 模块为模块扫描和验证内嵌在程序文档字符串中的测试提供了一个工具。测试编制就是简单的把一个典型的调用及它的结果剪切并粘贴到文档字符串里。这通过为用户提供一个示例改善了文档，并且它允许 *doctext* 模块确认代码和文档相符。

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

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

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

unittest 模块不像 doctest 那么容易使用，不过它可以在一个独立的文件里提供更全面的测试集。

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

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

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

10.12. “瑞士军刀”

Python 展现了“瑞士军刀”的哲学。这可以通过它更大的包的高级和健壮的功能来得到最好的展现。列如：

xmlrpc.client 和 xmlrpc.server 模块让远程过程调用变得轻而易举。尽管模块有这样的名字，用户无需拥有 XML 的知识或处理 XML。

email 包是一个管理邮件信息的库，包括 MIME 和其它基于 RFC 2822的信息文档。不同于实际发送和接收信息的 smtplib 和 poplib 模块， *email* 包包含一个构造或解析复杂消息结构（包括附件）及实现互联网编码和头协议的完整工具集。

xml.dom 和 xml.sax 包为这些流行的数据交换格式解析提供了强健的支持。同样的，csv 模块支持从一种通用的数据库格式中直接读写。总之，这些模块和包大大简化了 Python 应用程序和其他工具之间的数据交换。

有若干模块可以实现国际化操作，包括： gettext ， locale 和 codecs 包。

11. 标准库概览 — 第二部分

本部分覆盖了支持专业编程需要的更高级的模块。这些模块在小脚本中很少出现。

11.1. 输出格式化

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

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

pprint 模块为内置和用户自定义对象提供更精确的输出控制，从某种程度上更利于解释器阅读。当结果比一行更长时，“美化打印器”就会添加行中断和缩进，一边更清晰的显示数据结构。

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

textwrap 模块格式文本段落，以便适应指定的屏幕宽度。

```
>>>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.
```

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

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

11.2. 模板

string 模块包含一个通用的 Template 类，为最终用户的编辑提供了简化的语法。这允许用户无需改变就可以定制他们的应用程序。

这个格式使用 $ 加有效的 Python 标志符（数字，字母和下划线）形式的占位符名称。通过在占位符两侧使用大括号便可以不用空格分割在其后面跟随更多的字母和数字字符。使用 $$ 来创建一个单独 $ 转码字符。

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

当一个占位符在字典或关键字参数中没有被提供时， substitute() 方法就会抛出一个 KeyError 异常。对于邮件合并风格的应用程序，用户提供的数据可能并不完整，这时使用 safe_substitute() 方法可能更适合 — 如果数据不完整，它就不会改变占位符。

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

```
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

模板子类可以指定一个自定义分隔符。例如，图像查看器的批量重命名工具可能选择使用百分号作为占位符，像当前日期，图片序列号或文件格式。

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

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

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

模板的另一个应用是把多样的输出格式细节从程序逻辑中分类出来。这便使得 XML 文件，纯文本报表和 HTMLWEB 报表定制模板成为可能。

11.3. 使用二进制数据记录布局

`struct` 模块为使用变长的二进制记录格式提供了 `pack()` 和 `unpack()` 函数。下面的示例演示了在不使用 `zipfile` 模块的情况下如何迭代一个 ZIP 文件的头信息。压缩码 `"H"` 和 `"I"` 分别表示2和4字节无符号数字， `"<"` 表明它们都是标准大小并且按照 *little-endian* 字节排序。

```
import struct

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

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

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

11.4. 多线程

线程是一种为了分离那些无顺序依赖关系任务的技术。线程可以用来提高应用程序的响应速度，那些应用程序可以在接收用户输入的同时在后台运行其他任务。一种相关的应用就是在运行 I/O 的同时在另一个线程中执行计算。

下列代码演示了:mod:`threading`高级模块如何在主程序继续执行的同时又在后台运行任务。

```
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.')
```

多线程应用程序的主要挑战是协调线程，诸如线程间共享数据或其它资源。为了达到那个目的，线程模块提供了许多同步化的原生支持，包括：锁，事件，条件变量和信号灯。

尽管这些工具很强大，微小的设计错误也可能造成难以挽回的故障。因此，任务协调的首选方法是把对一个资源的所有访问集中在一个单独的线程中，然后使用 queue 模块用那个线程服务其他线程的请求。为内部线程通信和协调而使用 Queue 对象的应用程序更易于设计，更可读，并且更可靠。

11.5. 日志

logging 模块提供了一个完整功能和灵活的日志系统。最简单的形式就是把日志信息发送到一个文件或 sys.stderr 。

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
```

```
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

输出如下：

默认情况下，提示信息和调试信息都会被捕获并被发送到标注错误输出。其他的输出选项包括通过邮箱路由信息，数据报文，sockets 或者 HTTP 服务器。新的过滤器可以基于信息优先权选择不同的路由： DEBUG ， INFO ， WARNING ， ERROR 和 CRITICAL 。

可以从 Python 中直接配置日志系统，或者为定制日志从一个用户可编辑的的配置文件中加载而无需修改应用程序。

11.6. 弱引用

Python 自动进行内容管理（为大多数对象做引用计数并为消除循环引用作 *garbage collection* ）。在对象最后一个引用消除后，内存就会立即被释放。

这种方法在大多数应用程序中工作良好，但偶尔也需要在对象被其它东西使用时追踪对象。不幸的，仅仅为跟踪它们而创建的引用会使其持久存在。 weakref 模块提供了无需创建引用便可跟踪对象的工具。当对象不再需要时，它会被从一个弱引用表中自动的删除并且会为弱引用对象触发回调。典型的应用程序的创建都是昂贵的，包括的缓存对象。

```
>>> import weakref, gc
>>> class A:
...         def __init__(self, value):
...                 self.value = value
...         def __repr__(self):
...                 return str(self.value)
...
>>> a = A(10)                          # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a                   # does not create a reference
>>> d['primary']                       # fetch the object if it is still alive
10
>>> del a                              # remove the one reference
>>> gc.collect()                       # run garbage collection right away
0
>>> d['primary']                       # entry was automatically removed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                      # entry was automatically removed
  File "C:/python31/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7. 列表工具

很多数据结构要求可能用内置的列表类型就可以满足。然而，有时出于不同性能取舍需要从中选择一种实现。

array 模块提供了一个类似列表的 array() 对象，它存储同一类型的数据并且更为简洁。以下示例演示了一个存储2字节无符号二进制数字（编码类型 "H" ）数字数组，而非常见的每个项为16字节的 Python 整型对象的正规列表。

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

collections 模块提供了一个类似列表的 deque() 对象，它从左边添加（append）和弹出（pop）更快，但在中间查找时更慢。这些对象更适合实现队列和广度优先的搜索树。

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

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

除了替代列表的实现外，该库还提供了其它工具，像操作排序列表的 bisect 模块函数。

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

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

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

11.8. 十进制浮点数计算

decimal 模块为十进制浮点数计算提供了一个 Decimal 数据类型。与内置的 float 二进制浮点数实现相比，新类对商业应用程序和其他诸如需要精确十进制表示、控制精度、为法律或管理的需要控制舍入、确保小数点的有效位数的应用要求，或者那些用户想要控制数学计算结果的应用程序尤为有用。

例如，计算一次70分钟电话费对应的5%的税费，使用十进制浮点数和二进制浮点数的结果是不同的。如果要对最接近的分钟数进行舍入，这种差别就变得很重要。

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

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

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

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

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

Decimal 模块为算术提供了需要的高精度支持。

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

12. 现在做什么呢？

阅读本指南可能已经增加了你使用 Python 的兴趣 — 你应该已经迫不及待的希望使用 Python 来解决实际中的问题了。你应该从哪里学习更多的知识呢？

本指南是 Python 文档集的一部分。还有其他一些文档：

The Python Standard Library:

你应该浏览此手册，它包含关于标准库中类型，函数和模块的完整（虽然简炼）参考资料。标准 Python 发布版包含了 *大量* 的附加示例代码。有读取 Unix mailboxes 的模块，从 HTTP 中获取文档的模块，生成随机数的模块，解析命令行参数的模块，编写 CGI 程序的模块，压缩数据的模块，以及完成许多其他任务的模块。浏览库参考你会了解都有哪些可用的

模块。

Installing Python Modules:

解释如好安装其他 Python 用户编写的扩展模块。

The Python Language Reference:

Python 句法和语法的详细说明。比较难以阅读，但作为一个完全指南对理解语言自身很有帮助。

More Python resources: 更多 Python 资源（PS：我就不翻译了～）

- 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.)
- http://scipy.org: The Scientific Python project includes modules for fast array computations and manipulations plus a host of packages for such things as linear algebra, Fourier transforms, non-linear solvers, random number distributions, statistical analysis and the like.

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. 交互的输入编辑和历史记录

一些版本的 Python 解释器支持对当前输入行和历史记录的编辑，就像 Korn Shell 和 GNU Bash Shell 提供的工具一样。这是使用支持 Emacs 风格和 VI 风格的 *GNU Readline* 库实现的。这个库包含自己的文档，这里我就不列出了，但最基本的用法是很容易理解的。这里描述的交互编辑和历史记录功能在 Unix 和 Cygwin 版本的解释器中都是可选的有效功能。

本章 *不* 包括 Mark Hammond 编写的 PythonWin 包中编辑功能或 Python 携带的基于 Tk 的 IDLE 环境的文档。在 NT 上的 DOS 窗口或其他 DOS 中，以及 Windows 系列系统上，命令行历史记录是糟糕的。

13.1. 行编辑

如果支持的话，输入行编辑功能在解释器中总是可用的。当前行可以使用常见的 Emacs 控制字符进行编辑。最重要的一些包括： C-A （Control-A），将光标移动到左边； C-F ，将光标移动到右边； :kdb:`C-K` ，删除行中光标右边的内容； :kdb:`C-Y` ，删除行中光标左边的内容； :kdb:`C-underscore` ，撤销最后的编辑修改，可以重复撤销。

13.2. 历史记录

历史记录按照如下方式工作：所有非空的输入行都会被存储在一个历史缓冲区内，当一个新提示符出现时，你被置于这个缓冲区底部的一个新行上。使用 C-P 可以在这个历史缓冲区内向上（往后）移动一行，而 C-N 是向下（往前）移动一行。历史缓冲区内的任何行都是可编辑的，当对某行作编辑后就会在提示符前面显示一个星号（*）来标示。按下 Return 键就可以将当前行传递给 Python 解释器。

13.3. 键绑定

关于 Readline 库的键绑定和一些其他参数可以在一个叫做 ~/.inputrc 的初始化文件中通过放置命令自定义。键绑定的书写格式是：

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

或者：

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

并且可以通过如下方式设定选项：

```
set option-name value
```

例如：

```
# 1 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
```

注意：在 Python 中， Tab 默认绑定为插入一个 Tab 字符代替 Readline 的默认文件名补全函数。如果你较真，你可以通过在 ~/.inputrc 文件中加入以下键绑定覆盖它。

```
Tab: complete
```

（当然，如果你已经习惯使用 Tab 来补全文件名，这会让输入缩进延续行变得困难。）对于变量和模块名也可以使用自动补全。想要在解释器的交互模式中使用它，你需要在

启动文件中添加以下代码： [1]

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

这会把 Tab 键绑定到补全函数上，所以当敲击两次 Tab 键时就会提示补全字符。它会查看 Python 声明的名称，当前局部变量和有效的模块名。对于类似 string.a 的点号表示，它将这个表示看作最后的 '.' 并且从结果对象的属性中给出补全建议。注意：如果一个带有 __getattr__() 方法的对象是这个表示的一部分，这可能会执行程序定义代码。

一个更有效的启动文件看上去可能类似这个示例。注意：由这个文件创建的名称一旦不再需要就会被删除；在同一个命名空间中，像交互命令，从启动文件被执行时就会完成这些绑定，并且通过删除名称来避免给交互环境带来副作用。你可能会发现这对维持一些导入的模块很方便，像 os ，它们被证明在大多数解释器会话中是必须的。

```
# 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. 注释

与更早的解释器版本相比，这个功能是一个庞大的步骤。然而，一些要求被舍弃：如果在连续行中使用适当的缩进一切就 OK 了（当下一行要求一个缩进标记时，语法分析器知道如何处理）。完整的结构可能使用解释器的 table 标志。使用命令检查（或提示）匹配的圆

括号，引号等也是很有用的。

Footnotes

当交互解释器启动时，Python 会执行环境变量 **PYTHONSTARTUP** 定义的文件内容。[1]