

e 时代网络学科推荐教程

E-time Network Discipline Recommend Textbooks



Python

程序员指南

杨昆 汪兴东 / 编著

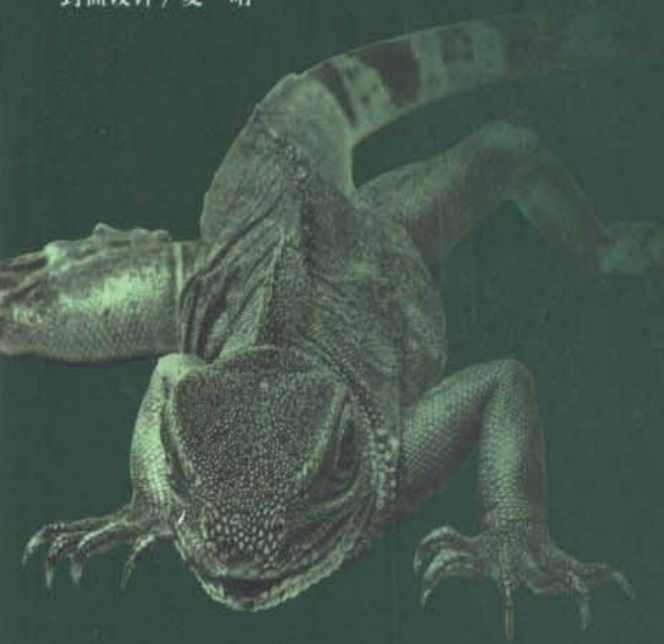
- ◇ 本书主要讲解 Python 语言的基础知识、编程及应用
- ◇ 本书的概念和思路清晰，并对相应的概念提供了大量的编程实例，具有很高的参考和应用价值
- ◇ 本书的内容深入浅出、通俗易懂，从最简单的例子着眼逐步进阶，不仅适合初学者，对使用 Python 编写各种大型、复杂应用程序的程序员来说也是一本实用的参考书

 随书附赠光盘，内含红旗中文 2000 办公平台、Python 工具箱和实例等精彩内容



中国青年出版社

责任编辑/江颖
/朱新媛
特约编辑/江帆
封面设计/夏晴



Python

程序员指南

e 时代网络学科推荐教程
E-time Network Discipline Recommend Textbooks

网络专家从这里
走出……

◆ 精通 XML 与网页设计高级教程(1CD)	定价: 48 元
◆ 精通 XHTML 程序设计高级教程(1CD)	定价: 48 元
◆ 精通 Dynamic HTML 高级教程(1CD)	定价: 48 元
◆ 精通从 HTML 到 XML 实务经典(1CD)	定价: 49 元
◆ HTML 标准教程(1CD)	定价: 48 元
◆ 精通 C++ Builder 5 程序设计高级教程(1CD)	定价: 79 元
◆ 精通 C++ 范例教程(1CD)	定价: 69 元
◆ Java 程序设计入门教程(1CD)	定价: 58 元
◆ JavaScript 网页特效应用与开发手册(1CD)	定价: 29 元
◆ JSP 交互网站实务经典(1CD)	定价: 49 元
◆ 精通从 JavaScript 到 JSP 范例程序设计	定价: 69 元
◆ ASP 入门与实例演练(1CD)	定价: 48 元
◆ 精通 ASP 建站技巧(1CD)	定价: 48 元
◆ 精通 IIS 5 系统规划与管理(1CD)	定价: 47 元
◆ Python 程序员指南(1CD)	定价: 29 元

◆ Python 程序员指南

Python 语言的优越性

- ◇ Python 易于使用，但它是真正的程序语言，能比 shell 提供更多的结构和对大程序的支持
- ◇ Python 提供比 C 更强大的错误检查功能，作为高水平的语言，它拥有极高水平的内建数据类型
- ◇ Python 比 Perl 和 awk 处理的问题更广、更大，至少它不会比其他语言更加复杂
- ◇ Python 允许将程序分割为一些模块，以便与其他的 Python 程序共享
- ◇ Python 还有一些内建的模块提供文件 I/O、系统调用、插座，甚至像 TK 那样的 GUI 工具界面
- ◇ Python 是一种公共域的面向对象的动态语言

ISBN 7-5006-4438-8



9 787500 644385 >

ISBN 7-5006-4438-8/TP · 207 定价: 29.00 元(附 1CD)

e 时代网络学科推荐教程
E-time Network Discipline Recommend Textbooks



Python

程序员指南

杨昆 汪兴东 / 编著

python



中国青年出版社
CHINA YOUTH PRESS

(京)新登字 083 号

本书由中国青年出版社独家出版。未经出版者书面许可,任何单位和个人不得以任何形式复制或传播本书的部分或全部。

策 划: 胡守文

王修文

郭 光

责任编辑: 江 颖

朱新媛

责任校对: 肖新民

书 名: 《Python 程序员指南》

编 著: 杨昆 汪兴东

出版发行: 中国青年出版社

地址: 北京市东四十二条 21 号 邮政编码: 100708

电话: (010) 84015588 传真: (010) 64053266

印 刷: 北京新丰印刷厂

开 本: 16 开

版 次: 2001 年 8 月北京第 1 版

印 次: 2001 年 8 月第 1 次印刷

印 数: 1-5000

定 价: 29.00 元 (附 1CD)

作者序

最近举办的第九届 Python 研讨会上, Guido van Rossum 宣布了 Python 软件基金会 (PSF) 正式成立。Python 是一种公共域的面向对象的动态语言。1990 年由 Guido van Rossum 开发, 用 Monty Python 剧团的名字命名, 作为一种描述性语言和快速的开发工具, Python 很快得到普及。Python 是真正的免费软件, 因为关于软件的拷贝或者发布任何用 Python 开发的应用程序没有规则限制。只要得到一份 Python 的拷贝, 就等于得到了全部源代码、个调试程序, 一个代码浏览器和一套常用的 GUI 界面。它可以在包括 Linux 在内的任何操作系统平台上运行。

Python 已经变成目前使用的最流行的语言之一, 它通常作为编译语言如 C 和描述性语言如 Perl 和 tcl/tk 之间的一种中介语言。为什么 Python 如此受欢迎呢? Python 语言本身是用语描述性的, 但是有几个特征使它不仅仅是一种简单的描述工具。比如, Python 是可展开的, 这就使得它可以适应并扩展以满足用户的需要。Python 代码易于阅读和维护, Python 也是面向对象的, 尽管你不必使用面向对象特性进行开发。

虽然在国内 Python 的脚步声刚刚踏响, 但在国外 Python 早已经成为一种成熟的解析语言并在各个方面得到了广泛的应用。Linux 的老大 RedHat 的安装程序就是用 Python 编写的, 在国内比较著名的 Linux 系统中, 红旗 Linux 和北京红旗中文 2000 软件技术有限公司 (该公司成立于 2000 年 12 月, 是香港文化传信集团之中文 2000 科技有限公司与北京中科红旗软件技术有限公司合作经营的企业, 该公司的骨干技术人员来自中科院软件所开放系统与中文信息处理中心。合作双方在打破国外软件尤其是操作系统领域的垄断地位, 振兴民族软件产业方面, 有着共同的立场和抱负——就是要为普通用户提供一个廉价易用的用户操作环境——中文 2000。) 的中文 2000 Linux 的安装程序也是用 Python 语言开发的。

迪斯尼公司的工程师发现 Python 语言在使用的时候非常方便, 更由于整个迪斯尼公司大量采用了开放源码软件, 最后毫不犹豫地用 Python 取代了 Perl 语言。现在, 迪斯尼公司生产的各种新动画片中许多功能都是由 Python 实现的。

如今, 在网络中已出现了许多研究、探讨、普及 Python 的中文站点, 但是这些站点上的内容比较零散, 很不系统, 并不适合 Python 的学习者。

本书比较完整地介绍了 Python 的知识和应用, 力图做到概念和思路清晰, 并对相应的

概念提供了大量的编程实例。读者可以在本书介绍的知识的基础上开发出更加复杂的应用程序。

本书由汪兴刚和杨昆两人编著。在编写过程中,得到了很多 Python 程序员的支持和帮助,在此深表谢意。

本书基本上覆盖了 Python 语言基础知识和应用的内容,但是限于本书的篇幅,还有一些内容不能完全叙述。比如应用和安全方面,对一种语言而言是比较重要的,但由于本书毕竟不是专著,因此不可能做到面面俱到。限于时间和作者水平,不当之处请读者指正。

杨昆 汪兴刚

2001 年 5 月

前 言

如果你曾经写过大型的 shell script, 大概能了解那种感觉: 想要添加一个功能, 但是这个 script 已经够大够慢够复杂了, 或者说, 你想要加入的新功能需要调用系统功能或是其他函数, 但是这些功能/函数只有 C 才能调用。你要解决的问题好像并没有严重到要重新用 C 来写整个程序, 或者有些问题因为要用到可变长度的字符串或是特别的数据结构(像是用排序过的文件名称组成序列(list)), 用 C 来写实在比 shell 麻烦得多, 又或者你根本不是对 C 很熟。

另外一个情境是这样的: 也许你要使用好几个 C 的连接库, 但是标准开发 C 程序的过程(写/编译/测试/重新编译)实在太费时间, 你需要能快速开发好软件。又或者你已经写好一个应用程序, 这个程序可以使用一个扩展的语言来控制。你不想创造一种语言, 可是还得写好这个语言的编译器, 还得把这个编译器跟你的程序放在一起。

在这些情况之下, Python 也许正是你所需要的语言。Python 虽然简单, 却是不折不扣的程序语言。对大型的程序来说, 它比起 shell 能提供更多的结构性及支持。另外一方面, 它也提供了比 C 语言更多的错误检查。由于 Python 是一个非常高级的语言, 所以它有许多内建的数据类型, 像是有弹性的数组及字典(dictionary)等等, 如果用 C 来做的话得花上你大半天的时间。正是因为 Python 有较为一般性的数据类型, 所以它的可应用范围比起 awk 甚至是 Perl 要广很多, 最起码, Python 跟这些语言一样容易开发。

Python 的另外一个特点就是可以将程序切成小模块, 然后这些模块还可以应用在其他程序之中。Python 本身也有一个相当大的标准模块库让你使用, 或者当作学习 Python 程序设计的范例。在 Python 中也有内建的模块可以提供许多功能, 诸如: 文件 I/O、系统调用、sockets, 甚至是与 Tk 之类的 GUI 工具互动的接口。

Python 是一个直译式的语言, 可以省掉你在开发程序时不少编译及连接程序的时间。这个 Python 的直译器甚至可以交互式地使用, 让你在写一些小程序来试验 Python 语言的特性, 或是测试程序时可以节省不少时间。你还可以把 Python 直译器当作计算器呢。

Python 让你可以写出非常精练且可读性高的程序。用 Python 写出的程序通常比用 C 或 C++写的程序要短得多。为什么这么说呢?

因为其高级的数据类型, 使你可以用很简单的语句(statement)表达复杂的运作过程。

Python 使用缩排来代替 C/C++ 中常见的前后括号 {};

Python 不需要变量或是参数的声明;

Python 是扩展性高的语言。如果你知道如何写 C 语言程序的话,你很容易就能在 Python 的直译器中加入新的内建函数(function)或是模块,这样做的好处是你可以让程序中关键的部分速度调到最快,或者是连接 Python 到 binary 的链接库(例如是厂商做好的图形链接库)去。一但你真的需要,也可以把 Python 直译器加入到你用 C 写的应用程序里面去。然后 Python 就变成你的应用程序的扩展或是商业化的语言了。

关于附赠光盘

为了方便读者使用，本书附带配套光盘一张。光盘包含两大块内容：

一、红旗中文2000办公平台（RedOffice beta 0.8版）

此平台是北京红旗中文 2000 软件技术有限公司开发的中文 LINUX 系统。安装时用户只需将光盘放入光驱。以光盘启动系统，即可安装，用户可选择图形安装或者是文本安装。此系统是为熟悉 LINUX 的用户准备的，请慎重使用，如果在使用中有任何问题，请与北京红旗中文 2000 软件技术有限公司技术支持部联系。电话：010-62148118 转技术支持。也可访问红旗中文 2000 公司网站查询。网址：<http://www.ch2000.com.cn> Email:support@ch2000.com.cn

二、Python 工具箱

Python 工具箱内包括最新的 Python 发布版以及丰富的 Python 工具和实例。书里 Python 实例中的例子均可在“\tools\例子”中找到。另外“\tools\例子”这个目录在 Windows 下可能无法访问，需在 Linux 系统下访问。

Python 工具箱所在目录为“\tools”。其内容简要介绍如下：

1. Pthon 2.1

目录：\tools\Python2.1

说明：包括 Linux、Windows 环境下的 Python2.1 版及其他。Linux 版为 rpm 包，直接安装，Windows 版为 exe 的安装文件。

2. Jpython

目录：\tools\Jpython

3. PythonGTK

目录：\tools\pythonGTK

4. PythonXML

目录：\tools\pythonXML

5. Wxpython

目录：\tools\wxpython

6. Zope

目录：\tools\Zope

7. 官方文档

目录：\tools\官方文档

8. 例子

目录：\tools\例子

9. 未归类

目录：\tools\未归类

目 录

第一部分 利用 Python 编程

第 1 章 Python 概述

1.1 Python 的起源	3
1.2 Python 的优越性	4
1.3 了解 Python 语言	7
1.4 Python 的发展	9

第 2 章 安装并启动 Python

2.1 准备运行 Python	11
2.1.1 安装 Python	11
2.1.2 设置 Python 环境变量	13
2.2 使用 Python 的直译器	14
2.2.1 参数的传递	15
2.2.2 互动模式	15
2.2.3 程序错误处理	16
2.2.4 执行 Python 脚本 (script)	16
2.2.5 交互式启动文件 (startup file)	16
2.3 要 Windows 下安装 Python	17
2.4 在 Apache 下设置 Python	18
2.4.1 准备	18
2.4.2 配置	18
2.4.3 测试	19
2.4.4 后话	20
2.5 PyGTK 在 Windows 下的安装	20

2.5.1 安装准备	20
2.5.2 安装	21
2.5.3 测试 “Hello, world!” 程序	21

第 3 章 Python 语法

3.1 把 Python 当作计算器来用	24
3.1.1 数字	25
3.1.2 字符串	27
3.1.3 Unicode 字符串	33
3.1.4 列 (List)	34
3.2 迈向程序设计的第一步	36

第 4 章 变量、运算符和表达式

4.1 Python 语言的基本数据类型	39
4.2 标识符和关键字	40
4.3 声明变量	42
4.4 字符、字符串变量	42
4.5 数值类型	45

第 5 章 Python 数据结构

5.1 列表	51
函数程序设计工具	52
5.2 del 语句	53
5.3 序表和序列	54
5.4 字典	55
5.5 条件的进一步讨论	56
5.6 序列与其他类型的比较	57

第6章 控制流

6.1 if 语句.....	59
6.2 while 循环.....	61
6.3 for 循环.....	62
6.4 try 语句.....	63
6.5 range()函数.....	65
6.6 break 及 continue 及循环中的 else 子句.....	66
6.7 pass 语句.....	66
6.8 定义函数.....	67
6.8.1 预设内定参数值.....	69
6.8.2 关键词参数.....	71
6.8.3 随意的参数串.....	73
6.8.4 Lambda 形式.....	73
6.8.5 批注字符串.....	73

第7章 函数

7.1 定义函数.....	75
7.2 使用参数.....	77
7.2.1 预设内定参数值.....	77
7.2.2 关键词参数.....	79
7.2.3 随意的参数串.....	81
7.2.4 Lambda 形式.....	81
7.2.5 批注字符串.....	81

第8章 类与对象

8.1 Class (类).....	83
8.2 术语的使用说明.....	83
8.3 Python 的可用范围 (Scopes) 及 命名空间 (Naming Spaces).....	84
8.4 Class (类别) 初探.....	86

8.4.1 定义 Class (类别) 的语法.....	86
8.4.2 类别对象 (Class Objects).....	87
8.4.3 特例对象 (instance objects).....	88
8.4.4 Method Objects (方法对象).....	89
8.5 一些随意的想法.....	90
8.6 继承 (Inheritance).....	92
多重继承.....	93
8.7 Private 变量.....	94
8.8 其他.....	95
例外 (Exceptions) 也可以是类别.....	95

第9章 Python 语言调试

9.1 句法错.....	97
9.2 例外.....	97
9.3 例外处理.....	98
9.4 产生例外.....	100
9.5 用户自定义例外.....	101
9.6 定义清理动作.....	101

第10章 Python 的杀手程序 Zope

10.1 Zope 简介.....	103
10.2 Zope 动态网页发展及管理系 统简介.....	105
10.2.1 Zope 的内容管理器 (content manager).....	105
10.2.2 新增一个对象.....	105
10.2.3 编辑一个 DTML 文件对象.....	107
10.2.4 文件的属性.....	108
10.2.5 Zope Document Template Markup Language.....	108
10.2.6 特殊 TAG 的格式.....	109

10.2.7 变量与运算式	109
10.2.8 条件式	111
10.2.9 循环	111
10.2.10 Zope 的安全机制	112
10.2.11 Zope 如何决定用户	113
10.2.12 结语	114

10.3 Zope 与 Python 的关系	114
------------------------------	-----

第 11 章 Python 实例

11.1 Hello World 程序	119
11.2 变量和控制流	120
11.3 基本数据类型	121
11.4 基本数据类型 II: 次序和字典	122
11.5 函数和模块	126
11.6 有用的混合运算	129
11.7 对象休止	131
11.8 定义对象	133
11.9 面向对象的概念	136
11.10 更多的面向对象的概念	140
11.11 特殊类程序	146
11.12 Python GUI 编程简介	151
11.13 TK 小部件	151
11.14 TK 部件 2	154
11.15 TK 图形	157
11.16 TK 图形 2	163
11.17 TK 图形 3	170
11.18 CGI 编程	177

第二部分 wxPython 程序设计

第 12 章 wxPython 在 Win32 下编程

12.1 wxPython 简介	185
12.1.1 wxWindows	185
12.1.2 wxWindows + Python = wxPython	186
12.2 初识 wxPython	186
12.2.1 哪里可以得到 wxPython	186
12.2.2 一个简单的例子	186
12.2.3 在 wxPython 中的事件	189
12.3 用 Python 创建一个 Doubletalk 浏览器	191
12.3.1 MDI 框架	192
12.3.2 图标	194
12.3.3 时间	194
12.3.4 主菜单	194
12.3.5 wxFileDialog	197
12.3.6 wxListCtrl	198
12.4 xPython 窗口布局	201
12.4.1 约束	201
12.4.2 布局算法	202
12.5 大小管理器 (sizer)	202
12.5.1 资源	203
12.5.2 强制力	203
12.6 wxDialog and friends	203

第三部分 Python 的高级应用

第 13 章 Python 和 XML

13.1 XML 的发展历史	207
----------------------	-----

13.2	XML 的优点	208
13.3	XML 的技术实现	209
13.4	XML 的相关技术	210
13.4.1	Xlink 与 Xpointer	210
13.4.2	Xpointer	213
13.4.3	DOM (Document Object Model)	213
13.4.4	Namespaces	214
13.4.5	TML	216
13.5	XML DOM	218
13.6	thon 和 XML	224
13.6.1	主要模块和包	225
13.6.2	文档对象模型	230
13.6.3	将 HTML 转换成 XML	231
13.6.4	将 Python 对象转换成 XML	232
13.6.5	将 XML 文档转换成 Python 对象	234
13.6.6	Python 交互式会话	236
13.6.7	重新安排 DOM 树	236
13.7	Python 和 XML 的结合	237
13.7.1	xml pickle	238
13.7.2	xml pickle 设计特点	241
13.7.3	xml objectify	243
13.7.4	xml_objectify 的设计特点	245

13.7.5	xml_objectify 的前景	247
--------	-------------------------	-----

第 14 章 Python 中的 Curses 编程

14.1	Curses 的历史与版本	249
14.2	认识 Curses 编程的思路	250
14.3	Curses 多视窗处理方式	258
14.4	Python: Curses 编程	264

第 15 章 Python 中的 TK 编程

15.1	TK 简要描述	271
15.2	基本知识	272
15.2.1	最小的 [Tkinter] 程序	272
15.2.2	main() 函数	272
15.2.3	应用几何图形管理器	274
15.2.4	菜单	274
15.2.5	接受用户输入	275

第四部分 附录

附录 A 交互式输入编辑及代换过去的内容

A.1	整行编辑	279
A.2	代换过去的内容	279
A.3	键盘连接	280
A.4	评注	281

附录 B Python 资源

第一部分 利用 Python 编程



第 1 章 Python 概述

语言是人们描述现实世界，表达自己思想观念的工具。而计算机语言是人与计算机交流的工具。一方面人类使用各种计算机语言将所关心的现实世界映射到计算机世界；另一方面，人类又可以通过计算机语言创造现实世界中并不存在的虚拟世界。

计算机的数学理论基础是图灵于 1937 年提出的图灵机模型，而现代电子计算机的体系结构及实际计算模型则是来自冯·诺依曼 1946 年提出的“程序放入内存，顺序执行”的思想，因此，现在的计算机通常被称为冯·诺依曼计算机。计算机语言的发展历程从此正式开始，从而使计算机语言的使用人员开始被称为程序员。

1.1 Python 的起源

在计算机语言的发展过程中，先后出现的语言至少有几千种，但是真正能普及应用的计算机语言却是屈指可数的。一种计算机语言要能流行普及，除了要有独有的特色以外，还要切合当时的应用需求。

早期程序员们使用机器语言来进行编程运算，直接对以数字表示的机器代码进行操作。后来为了便于阅读，就将机器代码以英文字符串来表示，于是出现了汇编语言。1956 年首先在 IBM 公司的计算机上实现的由美国的计算机科学家巴科斯设计 FORTRAN 语言，标志着高级语言的到来。早期的这些计算机语言都是面向计算机专业人员，为了普及计算机语言，使计算机更为大众化，出现了入门级的 BASIC 语言，至今 BASIC 语言仍然是绝大多数软件开发人员接触到的第一门计算机语言，同时也最流行的计算机语言。

70 年代初，结构化程序设计的思想孵化出两种结构化程序设计语言，一种是 PASCAL 语言，另一种是 C 语言。这两种语言的语法结构基本上是等价的，它们都是通过函数和过程等语言特性来构成结构化程序设计的基础。但是很主要的区别在于 PASCAL 语言强调的是语言的可读性，因此 PASCAL 语言至今成为学习算法和数据结构等软件基础知识的教学语言；而 C 语言强调的是语言的简洁性以及高效性，因此 C 语言成为之后几十年中主流的开发语言，高效性使 C 语言的地位已相当于一种“高级汇编语言”。

当计算机技术逐渐在各种应用中得到普及时，使得软件开发效率提到日程上来。原有的

高级语言，如 BASIC、PASCAL 等结合可视化的界面编程技术、面向对象思想、数据库技术，产生了所谓的第四代语言，如 Visual Basic，Delphi 等。Visual Basic 的语言基础是 BASIC 语言，Delphi 的语言基础是 PASCAL，这两种语言都是软件开发人员所熟知的语言。

在 Web 技术的发展过程中，得到极大的普及的是 Java 语言。Java 是面向对象的网络语言，它的独特的网络特性包括：平台独立性、动态代码下载、为多媒体功能而设计的多线程、为通过 Internet 快速传送而设计的紧凑的代码格式。

Python 语言和其他很多语言一样，在一种特定的环境下成长发展起来的，Python 的创始人是 Guido van Rossum。1989 年圣诞节期间，在阿姆斯特丹，Guido 为了打发圣诞节的无趣，决心开发一个新的脚本解释程序，作为 ABC 语言的一种继承。之所以选中 Python（大蟒蛇的意思）作为程序的名字，是因为他是一个 Monty 大蟒蛇飞行马戏团的爱好者。

ABC 是由 Guido 参加设计的一种教学语言。就 Guido 本人看来，ABC 这种语言非常优美和强大，是专门为非专业程序员设计的。但是 ABC 语言并没有成功，究其原因，Guido 认为是非开放造成的。Guido 决心在 Python 中避免这一错误（的确如此，Python 与其他语言如 C，C++ 和 Java 结合的非常好）。同时，他还想实现在 ABC 中闪现过但未曾实现的东西。就这样，Python 在 Guido 手中诞生了。实际上，第一个实现是在 Mac 机上。可以说，Python 是从 ABC 发展起来，主要受到了 Modula-3 的影响。并且结合了 Unix Shell 和 C 的习惯。

1.2 Python 的优越性

计算机业经过了 PC 革命，又迎来了网络革命，下一个大的革命也许就是智能革命。但在智能时代到来之前，还是有很多有意义的改进工作可做，近年发展的一些半自动开发工具一定程度上降低了劳动强度，对程序语言的改进一直在进行。

然而对于编程来说，程序语言不是关键性的因素，但对工作效率却有不可低估的影响。经过测试证明一些 Script 语言（如 Python，Perl 等）和传统的语言（如 C，C++）相比，开发速度有 5 倍以上的差距。抽象地讲，Python 是一门解释性的、面向对象的、动态语义特征的高层语言。它的高层次的内建数据结构，以及动态类型和动态绑定，这一切使得它非常适合于快速应用开发，也适合于作为胶水语言连接已有的部件。Python 的简单而易于阅读的语法强调了可读性，因此降低了程序维护的费用。Python 支持模块和包，并鼓励程序模块化和代码重用。Python 的解释器和标准扩展库的源码和二进制格式在各个主要平台上都可以免费得到，而且可以免费分发。

通常，程序员爱上 Python 是因为它能增加生产力。由于没有编译过程，编辑—测试—调

试周期相当快。调试 Python 程序很简单：一个错误永远不会导致一个段错误。当解释器发现错误时，它就引发一个异常。当程序没有捕捉到异常，解释器就打印一个堆栈跟踪。一个源码级调试器允许我们检查局部和全局变量、计算表达式、设置断点、单步跟踪等等。调试器是用 Python 写的，这证明了 Python 的能力。另外，最快的调试程序的方法是增加几条打印语句：快捷的编辑—测试—调试周期使得这个简单的办法十分有效。

就像前面讲到的一样，一门计算机语言如果没有其特殊性和优越性是不可能得到迅速的发展和应用的。就简单而言 Python 语言有很多和其他流行语言一样的优越性。

● 容易学习

对于 Python 来说，能够在短期内迅速地得到普及和发展是在于它的易学性。对于一名掌握 C、Perl 或 shell 语言的程序员而言，花 3 至 5 个小时就能基本了解 Python 语言的基本特性。这和 Python 语言本身的简洁的特性是分不开的。

同时 Python 提供了一个交互式环境，这也是它容易学习的主要原因之一，就像以前的 Basic 和一些数据库语言环境（如 Dbase, SQL 等）一样，正如一个资深程序员所说，软件的复杂是因为千万件事堆在一起，交互式环境恰好可以让我们把程序分解开，我们可以试验每一个不清楚的语言成分，同时这也是一个很好的测试平台。

下面我们来看看同一个函数用 Python 和 Perl 来实现以证明其特性：

```
#—— Python version.———
def pairwiseSum (list1, list2) :
    result = []
    for i in range (len (list1)) :
        result.append (list1[i] + list2[i])
    return result

#—— Perl version. ——
sub pairwiseSum {
    my ($arg1, $arg2) = @_ ;
    @list1 = @$arg1; @list2 = @$arg2;
    for ($i=0; $i < length (@list1) ; $i++) {
        push (@result, $list1[$i] + $list2[$i]) ;
    }
    return (\@result) ;
}
```

```
}
```

看完上面的函数对比后相信你自然会对 Python 语言的简洁明了加以赞赏，的确 Python 的风格更接近自然语言，显得更通俗易懂。

● 容易阅读

作为一种开放性语言，其代码设计的易读性更决定了它的生命力，在越来越注重代码重用的今天，Python 的易读性真实地说明了它强大生命力。下面我们来看一段用 Python 语言写的代码。

```
#!c:\python\python.exe
import sys
import string

if len ( sys.argv ) < 2 :
    print "Usage: leap.py year, year, year..."
    sys.exit ( 0 )

for i in sys.argv[ 1 : ] :
    try :
        y = string.atoi ( i )
    except :
        print i, "is not a year."
        continue

    leap = "no"

    if y % 400 == 0 :
        leap = "yes"
    elif y % 100 == 0 :
        leap = "no"
    elif y % 4 == 0 :
        leap = "yes"
    else :
        leap = "no"

    print y, "leap:", leap, "in the Gregorian calendar"
```

```
if y % 4 == 0 :
    leap = "yes"
else :
    leap = "no"
print y, "leap:", leap, "in the Julian calendar"

print "Calculated leapness for", len ( sys.argv ) - 1, "years"
```

该代码是求解古老的年份问题,读者会很惊讶地发现 Python 的语法像 C 和 Shell 的混编,继承了上述两种语言的很多特性,但是它更加容易让人接受。相比较其他的 Script 语言,Python 易读的特性是很明显的。

1.3 了解Python语言

上面我们仅仅从学习的角度来看 Python 优越性,其实使 Python 具有强大生命力的最根本的原因是其作为计算机语言不但拥有自身完美的特性,更重要的是它是一门开发性语言。Python 语言在其开放标准下的不断壮大,在可伸缩性、灵活性和发展方面都为应用者提供了很好的机会。

Python 作为一种计算机语言,它不但拥有其他流行的 Script 语言很多特性外,它还拥有其他 Script 语言所无法比拟的优越性。

● 简单性

Python 语言的简单性可以概括为两个部分:一是语言本身的组成成份较少,结构较小;二是与已有语言类似,用户容易熟悉掌握。如果你用过 C 或 C++ 语言,就会发现 Python 使用了许多与 C 和 C++ 同样的语言结构。当我们编写了上面几个程序片后,我们会惊奇地发现:

- 1) Python 语言的变量不用先定义类型,直接由系统识别。
- 2) 它用行头空格数来标识块的起始,省去了 Perl/C/C++ 的 {;}。
- 3) 更没有了 Perl 中的 \$%& 等复杂的符号。
- 4) 而且类型方面有了很大的改进。

对 Python 语言的简单性我们将会在今后的章节中具体描述。

● 解释性语言

在使用 Python 编写程序之前弄清楚 Python 是如何工作的很重要。这里我们详细地来了解解释性语言和为什么代码可重用。

目前，常用的解释性语言有 Perl, Python, Lisp/Scheme, Ruby 等，在这里我们还是来了解一下到底什么是解释性语言。首先我们可以从其他解释性语言了解到，所谓的解释性语言其主要包括两个方面：一是它们都有自己的解释器，也可以通俗地理解为翻译器；二是它们都是在其他编译语言（通常是 C 语言）的基础上定义和扩充了自己的语法结构。解释性语言的工作原理就是用自己定义的解释器解释并执行由自己定义的语法结构生成的程序代码。所以解释性语言并不编译。这里我们要区分 Java，因为 Java 并不单单是一种解释性语言，它为了提高效率而拥有自己的即时解释器，但实际上 Java 可以算作是一门解释和编译的结合语言。

那么，Python 为什么是一种解释语言，解释语言有什么优越性呢？这个问题我们在前面也提到过，其实任何一种语言的出现都有其特殊的应用背景决定的。随着分布式概念的提出，计算机应用越来越趋向多平台适用性甚至跨平台。而随着 Web 应用的普及，从而使得解释性语言得到空前的发展，因为解释性语言拥有自己的解释器，所以这种语言对平台的依赖性降到了最低程度，而 Python 正是在这种背景下发展壮大起来的。

当然，虽然解释性语言对平台的依赖性相当的小，甚至独立于平台拥有很大的灵活性，但是由于它并不编译，所以用解释性语言编写的程序代码，在执行的过程中相对要比编译性语言效率要低得多。

● 面向对象

Python 是一种公共域的面向对象的动态语言。Python 和 Java 及 C++ 有许多共同的底层原则，它们的差异在于风格和结构。简单地讲面向对象的编程语言（简称 OOP）描述的是对象之间的互相作用。

和其他许多面向对象的语言一样，Python 支持多继承。它甚至支持异常的处理。如果学过 Java，应该对这个不陌生。这使得程序的编写更加清晰，而不需要许多的错误检查了。

● 模块和包

这一点更像是 Java。对于 Java 的支持，大家可以了解 JPython。JPython 是用 Java 写的 Python，它完全支持 Java，在这个环境下使用 Python 可以随意地使用 Java 的类库。

● 语言扩展

可以用 C、C++ 或 Java 为 Python 编写新的新言模块，如函数。或者与 Python 直接编译

在一起，或者采用动态库装入方式实现。也专门有人编写了一个工具，可以实现为 Python 自动实现函数接口封装，这就是 SWIG (Simplified Wrapper and Interface Generator)，或称做简单封装和接口生成器（可以在 <http://www.cs.utah.edu/~beazley/SWIG> 自由获得）。

● 优秀的语法

Guido 认为 Python 的语法是非常优美的。其中一点就是，块语句的表示不是 C 语言常用的 {} 对，或其他符号对，而是采用缩进表示法！有趣吧。就这一点来说，Guido 的解释是：首先，使用缩进表示法减少了视觉上的混乱，并且使程序变短，这样就减少了需要对基本代码单元注意的范围；其次，它减少了程序员的自由度，更有利于统一风格，使得阅读别人的程序更容易。感觉还是不错的，就 C 语言来说，在 if 语句后面大括号的写法就好几种，不同的人喜欢不同的样子，还不如统一起来，都不会看得别扭。

● 运行方式

Python 可以以命令行方式运行，也可以交互式方式运行，还具有图形集成环境，这样开发 Python 就相当方便。现在已经出现了许多用 Python 编写的可视化编程软件，用于实现像 Delphi 一样的功能。

1.4 Python的发展

Python 是网络环境中多种应用程序普遍适用的一种万能语言。BYTE 杂志评价说，它是 Unix shell 编程和 C 编程之间的一座桥。也就是说，用一般 shell 工具编写比较复杂，但是运用的程序项目 C 或 C++ 编写又太简单的程序，用 Python 比较合适。Python 最大的灵活性在于它能够接受 HTTP 请求并嵌入 HTML 页面，和有高驱动能力的 Unix shell 脚本一样具有非常强大的功能。它能处理更复杂的数据结构，例如联合的数组，并且它的语言富有表现力，能寻找数据库或充当 CGI 脚本。

Python 中有大量 hooks，能与不同的操作系统和环境兼容。它像一条变色龙一样，能相当容易从任何本地的文件输入。有些人认为，如果说 Perl 是程序管道，能即时编写不连续的脚本程序或数据，那么 Python 就是技艺熟练的总管，它拥有跨平台的能力，并且具有可升级、可扩展及可嵌入的特性。因此 Python 也是一个很吸引人的网页编程工具，相信在不久的将来 Python 将会得到广泛的应用。

第2章 安装并启动 Python

2.1 准备运行Python

如果你想边学习边使用 Python，就需要在你的系统中安装 Python 编程工具（如果系统尚未安装的话）。同时，必须把环境设置成能够运行 Python 的状态，并设置查找 Python 文件的目录。

由于很多 Linux 系统本身就包含了许多 Python 的工具，以 RedHat 为例，默认设置就安装 Python。通过核对是否存在 `/usr/bin/python` 文件，就可以检查到 Python 是否已经被安装。

2.1.1 安装 Python

本书的配套光盘中有 Python，所以你不必到处购买。如果由于某种原因你没有光盘，或者你想查看最新的版本，最方便的方法是通过 Python 的 FTP 站点 <ftp://ftp.python.org/src>。使用匿名 FTP 就可以得到源代码。

Python 源文件通常作为 C 的源代码提供，需要在系统中使用 C 编译器进行编译和链接。FTP 站点也包含大量对目标硬件和操作系统进行预编译的二进制文件，因此不再必须编译，但是必须确保你的机器支持下载的二进制文件。

注意：为了更方便的获得合适的二进制文件和源代码，对 Python 网点进行了修改，允许人们选择合适的平台和操作系统。这样就能传送合适的二进制文件。当我们有了二进制文件以后，可能还想得到源代码，当我们想在 Python 语言中加入 C 语言进行扩展时就会遇到这种情况。不管什么时候对 Python 进行扩展，都必须对二进制文件重新进行编译和链接，因此必须有源代码。

在编写本书的时候，可以从 FTP 站点和 Python 网站获得 Python 的 1.6 版。并且能够获得 Python 适用于多种平台和操作系统的各种版本，如适用于 UNIX 各种版本、Linux、Windows 和 Macintosh 的 Python 版本。

如果仅仅下载源代码，获得的 Python 文件通常是打包文件，需要用 `gzip -d` 进行解包。打

包文件通常用文件名的一部分来表明版本，如 `python1.3` 表示系统是 1.3 版的。文件的扩展名通常是 `.tar.gz` 或者是 `.tgz`，使用这些扩展名的文件用相同的命令解包。Python 软件通常包括一个或多个 `README` 文件，同时包含编译进程，并建议合适的操作系统。使用如下命令开始安装：

```
gzip -d python1.x.tar.gz
```

用 `gzip -d python1.x.tar.gz` 命令解包被打包的文件。（该命令是举例说明，在实际应用时，要用文件名代替命令中的 `python1.x.tar.gz`）然后，用下面的命令开始安装：

```
tar xvf python1.x.tar
```

同样用相应的文件名代替。需要注意的是该命令对扩展名为 `.tar.gz` 和 `.tgz` 的文件都适应。另外，即可以使用 `gzip -d` 命令也可以使用 `tar` 命令。下面解释 `-z tar` 的用法。

```
tar xvzf python1.x.tar.gz
```

在另一种情况下，在当前目录下解压 Python 文件。接下来是运行自动配置程序，使用如下命令：

```
/configure
```

然后用下面的命令编译可执行文件：

```
make
```

该命令对 Python 文件进行编译和链接。也可以运行自测程序正确执行了全部过程并且全部文件都考虑到了。使用下面的命令进行自测：

```
make test
```

最后，把编译好的可执行文件和所有的支持文件都复制到相应的 Linux 目录下，就完成了安装过程。可以使用如下命令：

```
make install
```

执行了上述的命令后，Python 就可以运行了。当然，如果使用 Python 的预编译 RPF 版，就不必执行以上过程。

注意：Python FAQ 定时发送到 Usenet *comp.lang.python* 新闻组，并且可以从几个 FTP 和 Web 网站得到，包括 <http://www.python.org>。FAQ 包括对语言和版本的修改信息，同时包括在各种平台上构造 Python 可执行文件的建议。

2.1.2 设置 Python 环境变量

为了使 Python 正常工作，很可能需要设置几种 shell 环境变量，这并不需要花很长的时间，也不必对 Linux 十分精通。Python 需要两个环境变量，分别叫做 PATH 和 PYTHONPATH，PATH 变量已经存在，它是登录到 shell 时设置的。如果使用 RPM 包安装 Python，根本就不必修改 PATH 变量，因为 Python 文件将被安放在 /usr/bin 目录下，这正是默认的目录。

一个快速判断 Python 是否被安放在正确的目录下的方法是执行 python 命令，观察输出结果。如果输出错误信息，说明 PATH 变量可能设置得不对。如果 Python 正确运行，可以看到三个直角支架，这表示可执行文件在所查找的路径。就可以按 Ctrl+D 键退出 Python。可以用以下命令确认当前路径：

```
echo $PATH
```

该命令将显示默认路径如下：

```
/bin:/usr/bin:$HOME/bin:.
```

如果 Python 的可执行文件在 /usr/bin 目录下，毫无疑问可以在该路径下找到 Python 的可执行文件。如果 Python 的可执行文件不在当前目录下，就需要修改启动文件（根据 shell 的不同，扩展名可能为 .cshrc, .login, .profile, 或 .kshrc）。只需要把可执行文件的位置加到已经存在的 PATH 变量的设置里，使用冒号分开各项。如果在启动文件中没有 PATH 语句，就添加一个，使用已经存在的默认路径 \$PATH 作为一个目录名。

必须为每个将要使用 Python 的用户设置 PYTHONPATH 环境变量。该路径常用于运行时给文件定位，因为多数文件都不在默认的路径下，所以在大多数情况下需要在启动文件中设置该变量。该变量设置通常包括当前路径，Python 文件的二进制位置（在安装时设置，通常在 /usr/lib 目录下），和 Python 需要的任何其他目录，如 tcl/tk 目录。

最后，为 Python 创建一个初始文件，当运行 Python 时读取该文件（如 shell 的启动文件）。在 PYTHONPATH 环境变量中给出该文件的名称，并且应该设置在启动文件的绝对路径下。启动文件中可以包含任何将运行的有效 Python 命令。

如果使用 tcl/tk 集成 GUI 和 Python，就一定要设置环境变量 TK_LIBRARY 和 TCL_LIBRARY。Python 使用这些变量寻找它所需要的 tcl/tk 文件。

2.2 使用Python的直译器

在 Unix 之类的操作系统上，如果有安装的话，Python 直译器通常安装在 `/usr/local/bin/python` 这个目录中。你可能需要先在 Shell 中设定寻找 `/usr/local/bin` 目录，这样你才可以在 Shell 中打入以下的指令来启动 Python 直译器：

```
python
```

你的 Python 直译器安装的位置是可以在安装时设定的，因此也有可能安装在其他的地方。你也许需要问你周遭的 Python 大师或是系统管理员才能知道正确的安装位置（`/usr/local/python` 是另外一个普遍的可能安装所在，关于这一点可以参考上一节）。

要离开 Python 直译器的话，打入 EOF 的字符在 Unix 上是 `Control-D`，在 DOS 及 Windows 上是 `Control-Z` 就会使得直译器离开（zero exit status）。如果行不通的话，你可以打入以下指令离开直译器：

```
import sys; sys.exit ()
```

Python 直译器使用每行编辑，应该不难使用。在 Unix 上，也许安装 Python 直译器的人有安装使用 GNU `readline` 链接库的功能，这样的话你会有交互式编辑以及编辑过去资料的功能。最简单的检查你有没有这项功能的方法就是在 Python 的提示之下打入 `Control-P`，如果有哔声的话，就表示你有这项功能，你可以翻到附录 A 去看特殊键的用法。如果你没有听到哔声，或是只出现 `P` 的话，就表示你没有这项功能，你得使用退格键（`backspace`）来清除目前所在行的字符了。

Python 直译器的操作方法跟 Unix shell 很像：当被调用时所连接的标准输入是 `tty device`（终端机）的话，直译器会互动的读及执行所输入的指令。当被调用时加入文件名称参数或所连接的标准输入是连到文件的话，直译器就会读入并执行该文件所含有的 `script`。

第三种启动直译器的方法是打入以下的指令“`python -c command [arg] ...`”，这个指令会执行 `command` 所代表的语句（这跟 shell 的 `-c option` 很像），因为 Python 语句（`statement`）常有空白及特殊字符，所以用此法时可以把 `command` 所代表的语句用“”括起来，以免跟 shell 的其他特殊字符或是参数有所混淆。

要注意的是“`python file`”指令跟“`python <file`”指令是有所区分的。对后者而言，不单单只有执行这个 `script`，程序中有关输入的需求（例如调用 `input()`或是 `raw_input()`）也都会由这个 `file` 来满足。由于此 `file` 已经在程序执行之初被从头到尾读过一次，所以一执行这个程

序将会马上碰到了 EOF。相反的对于前一个写法来说，程序的输入需求是由任何连接到 Python 直译器的标准输入（standard input）的装置或文件来满足的，这个也许才是你所想要的结果。

当 script 文件在使用的时候，也许你会想要执行这个 script 然后还可以继续进入互动的模式。这时你可以加入 `-i` 这个选项。但是如同前一段所说的，此 script 是由标准输入来读进去的话就没有办法这样做了。

2.2.1 参数的传递

如果 interpreter 认识 sys 的（话译：可用“import sys”指令），script 的文件名及附加传入的参数都会被记录在 `sys.argv` 这个变量并传给 script 来使用。`sys.argv` 是一列的字符串，长度至少为 1，如果你什么文件或参数都没传的话，`sys.argv[0]` 就是一个空字符串。如果 script 的名字是 `'-'`（就是标准输入的意思）的话，`sys.argv[0]` 就会被设为 `'-'`。如果使用 `-c command` 的话，`sys.argv[0]` 会被设定为 `'-c'`，所有的在 `-c command` 之后的 option（例如 `-i`）都会被当成 `sys.argv` 而被 command 处理，所以就不是当作 option 一样的来看待了。

2.2.2 互动模式

当指令是由 tty 终端机来传入时，我们称之为互动模式（interactive mode）。在此模式之下会出现主要的命令提示符号（primary prompt）来提示输入下一个指令，这个 primary prompt 通常是 `>>>`。如果指令是延续上一行的话就会出现 secondary prompt 符号，这个 secondary prompt 通常是 `...`。一进入 python 的互动模式直译器会出现一个欢迎信息以及版本编号、版权说明，然后是第一个 prompt。如下所示：

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

当你输入需要多行的结构时，直译器会自动出现延续上一行的 prompt 符号，下面的例子是 if 语句的情况：

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



```
...
```

```
Be careful not to fall off!
```

2.2.3 程序错误处理

有错误产生时，直译器就会在屏幕上出错误的信息以及 `stack trace` 的所有资料。在互动模式下，印完资料后会再印出 `prompt` 来。如果输入是来自于文件的话，出现错误时直译器在印出 `stack trace` 后程序会以 `nonzero exit` 的状态结束。（此处讨论不包含已经由 `try` 语句及 `except` 子句处理外的状况（Exceptions））。有些程序错误是没有办法挽救的，且会造成 `nonzero exit` 的结束情况，这常常是由内部不一致或是某种 `running out of memory` 所造成。所有错误信息都会被写入到标准 `error stream` 中，正常的程序执行的输出则会写入到标准（`standard output`）输出之中。

如果在 `primary` 或是 `secondary prompt` 下打入中断字符（通常是 `Control-C` 或是 `DEL`），这会造成输入的中断并且会回到 `prompt` 下。2.1 在指令执行中打入中断字符则会引起 `KeyboardInterrupt` 的 `exception`，而这是可以在 `try` 语句中处理的。

2.2.4 执行 Python 脚本（script）

在 BSD 之类的 Unix 系统上，我们可以在 `script` 的最前面加入以下语句（类似 `shell script`），并改变文件属性为可执行：

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

如此 `script` 就会变成可执行文件，可以直接执行（假设 Python 的直译器是在 `user` 的 `$PATH` 变量中）。“`#!`”这两个字必须在 `script` 文件的最前面。值得一提的是“`#`”在 Python 之中也当作注解（`comment`）部分开始的符号。

2.2.5 交互式启动文件（startup file）

当你使用互动模式的时候，如果可以在每次直译器要启动时先执行一些命令的话将是很有用的。要达成如此功能，你可以设定一个文件名称给环境变量 `$PYTHONSTARTUP`，这个文件可以包含你想要在启动时执行的命令，类似 `.profile` 在 Unix `shell` 中的用法。

这个启动文件（`startup file`）只有在互动模式下有效，你用 Python 读入 `script` 时就没有用，当 `/dev/tty` 是命令的输入来源时也没有用（其他情况与互动模式相类似）。这个 `startup file` 所执行命令的命名空间是与其他互动模式下输入的指令相同的，所以在 `startup file` 内定义或

是 `import` 的对象，在之后的互动模式指令中都是可以直接使用的。你也可以在这个 `startup file` 中改变 `sys.ps1` 及 `sys.ps2`，如此就可以改变你的 `primary prompt` 及 `secondary prompt`。

如果你在你的 `startup file` 中想要使用另外在目前目录的 `startup file`，只需要在主要 `startup file`（`global start-up file`）写入 “`if os.path.isfile('.pythonrc.py'): execfile('.pythonrc.py')`”。如果你想要在你的 `script` 中使用 `startup file` 的话，必须在 `script` 中写入：

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

2.3 要Windows下安装Python

在 Windows 下安装 Python 非常简单，首先下载 Python（目前最新版本 2.0）：

<http://www.python.org/ftp/python/2.0/BeOpen-Python-2.0.exe>

双击 `BeOpen-Python-2.0.exe` 进行安装。

安装完成后生成两个版本的 Python 运行环境。IDLE（Python GUI）和 Python（Command line），运行后如下图：



图 2.1 IDLE（Python GUI）

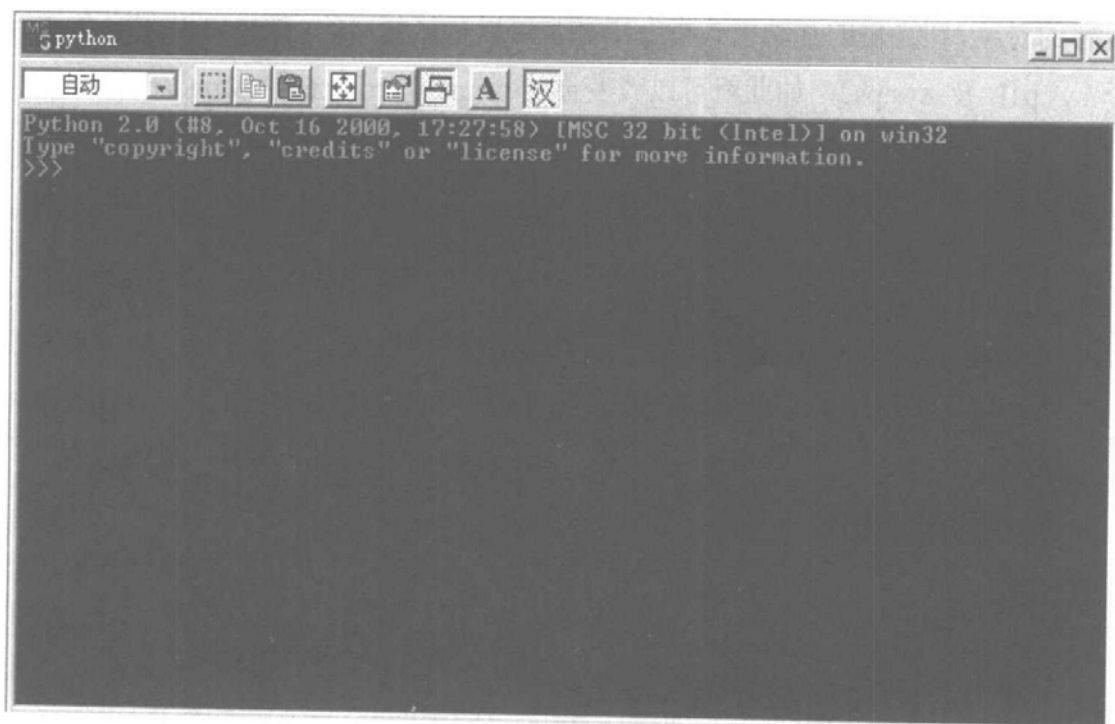


图 2.2 Python (Command line)

2.4 在Apache下设置Python

Python 可以开发 CGI 程序，那么在 Apache 下应如何配置呢？以下讲述了如何以 CGI 方式配置 Apache，使其支持 Python 程序。使用的系统环境为 Windows 98, Apache 1.3.19, Python 2.1 版。

2.4.1 准备

首先检查以下要求是否已经达到：

- 1) Apache 已经安装，并且可以正常使用。
- 2) Python 已经安装，并且可以正常使用（例如，Python 安装目录为 d:\python21）如果一切正常，下面就开始。

2.4.2 配置

1. 修改 DocumentRoot

打开 Apache 安装目录下的 conf 子目录的 httpd.conf 文件。可以修改 DocumentRoot 为“f:/phpsite”。当然你可以按需要改成其他值。

2. 允许任意目录执行 CGI

这个设置是允许被设目录及其子目录下的 CGI 程序可以 CGI 方式运行。在 Apache 中，尽管你可能已经设置了 CGI 文件后缀，但是如果未设置允许 CGI 程序运行选项，则无法运行 CGI 程序。

3. 设置 f:/phpsite 目录属性

```
<Directory "f:/phpsite">
    Options Indexes FollowSymLinks MultiViews ExecCGI
    AllowOverride None
    Order allow, deny
    Allow from all
</Directory>
```

其实并未重新设置新的目录属性，而是将 DocumentRoot 的目录属性（原来指向 Apache 安装目录下的 htdocs 目录，改成为 f:/phpsite 了）增加了 ExecCGI 一项。

4. 只允许特别目录执行 CGI

也可以只允许特别目录下可以执行 CGI 程序。与上一步可以同时执行，也可两种任选其一。只要设置：

```
ScriptAlias /cgi-bin/ "f:/phpsite/cgi-bin/"
```

即可。

5. 增加 CGI 文件名后缀

修改 AddHandler cgi-script 一句为 AddHandler cgi-script .cgi .py。即让 Apache 知道.py 的文件为 CGI 程序。

好了，到此 Apache 就配置好了，先启动 Apache，然后进行测试。

2.4.3 测试

Python 本身不像 PHP，不是一种嵌入式脚本（这种脚本比较适合作 Web 后端程序），所以所有输出要自己做。

```
1      #!d:/python21/python.exe
2      print "Content-type: text/html"
3      print
```

```
4      print "<h1>hello, world!</h1>"
```

第 1 行是让脚本以 CGI 方式运行必需的（这里是我的环境，记得吗？我前面说了 Python 是装在 d:/python21 下的），告诉 Apache 如何找到此文件的解释程序。

第 2, 3 行是告诉浏览器输出内容的 MIME 格式。这里为输出 HTML 文本。

第 4 行，输出"hello, world!"。其实不输出 HTML 的标记头浏览器也可以正常显示，尽管它不是完整的 HTML 格式。

如果测试成功，则一切大功告成。如果不行，听天由命吧（可以给我发信，咱们共同解决）。

2.4.4 后话

在 SourceForge 网站上，有一个名字 mod_snake 的项目。它同 mod_python 一样提供了针对 Apache 的 Python 模块化处理，但是 mod_snake 支持 HTML 文档嵌入 Python 标记，有点像 PHP。不过现在它只支持 Linux，而没有 Windows 下的版本。一定要转到 Linux 下面去！使用 CGI 方式调用 Python 速度不是很快，而如果使用 mod_python 则速度可能要快几十倍，是 mod_python 网站上说的。

2.5 PyGTK在Windows下的安装

GTK+是 Linux 下 Gnome 的底层开发包，原为 Gimp（Linux 下的图像处理工具）的图形库，但是发展已经不只于此。现在 GTK+/Gimp 已经移植到了 Windows 环境下，而且有相应的 Python 包对其进行封装。这样大家就可以在 Windows 下享受用 Python 开放 GTK+程序的乐趣了。这里主要向大家介绍，安装 PyGTK 所需要的东西和安装方法，及一个小的测试程序。

2.5.1 安装准备

在安装前要做好如下准备：

1) GTK+在 windows 上的 DLL 库大家可以去这个地址（<http://user.sgi.com/~tml/gimp/win32/downloads.html>）下载。配套光盘也有。要下载下面的库：

```
glib-dev-20001226.zip  
libiconv-dev-20001007.zip  
gtk+-dev-20001226.zip
```

```
gimp-dev-20001226.zip
extralibs-dev-20001007.zip
```

2) PyGTK 包可以去 Hans.Breuer.Org 下载 pygtk-2000-11-26.zip (<http://hans.breuer.org/ports/pygtk-2000-11-26.zip>), PyGTK 还有一个站点 (<http://www.daa.com.au/~james/pygtk/>), 但是无法直接用在 Windows 下, 需要编译, 因此不推荐此处。

3) 当然还要 Python 2.0 版, 因目前只能用于 Python 2.0。

对于 GTK+在 windows 下的动态链接库, 从上面提供的地址可以下载, 主要有:

```
gtk-1.3.dll
glib-1.3.dll
gmodule-1.3.dll
gnu-intl.dll
gdk-1.3.dll
iconv-1.3.dll
```

为了方便大家使用, 配套光盘中有一个压缩文件 gtkdll.zip, 包含了这些动态链接库。如果大家从上面的主页上下载这些库, 它们是分散在各个目录下的, 同时还有源码。

2.5.2 安装

首先安装 GTK+的动态链接库。将上述 DLL 文件下载后, 拷贝到 windows/system 目录下。

然后安装 PyGTK 包。将文件包 pygtk-2000-11-26.zip 用 winzip 打开, 可以看到有:

- 1) _gtk.pyd 将此文件拷贝到 Python 2.0 的 dll 目录下。
- 2) GDK.py gtk.py GTKconst.py 将这些文件拷贝到 Python 2.0 的 lib 目录下。
- 3) Authors Readme Copying 可以忽略。

到此安装完毕。

2.5.3 测试 “Hello, world!” 程序

下面我们编写一个在标题条上显示 “Hello, world!” 的小程序, 对 PyGTK 进行测试。

```
from gtk import *

window = GtkWindow (WINDOW_TOPLEVEL) # 创建一个顶层窗口
window.set_title ("Hello, world!")
```

```
window.connect ("destroy", mainquit) # 将注销事件与mainquit处理连接

window.show ()                        # 显示主窗口

mainloop ()                          # 进入事件循环
```

在 dos 窗口下（运行 Python 程序最好在命令行下执行），执行 `python helloworld.py`。你会看到一个标题条有“Hello, world!”的空窗口显示出来。

到这里 PyGTK 就安装成功了。

第3章 Python 语法

在下面的例子里，你可以很容易区别凡是需要输入的地方都会出现 prompts (“>>>”或“...”)，凡是输出的结果则没有。如果你想要跟着这个教学文件一起做的话，你就得打入所有在 prompts 之后的指令，凡是没有 prompts 出现的行就是直译器输出的结果。值得注意的是，secondary prompt 之后如果什么东西都没有，表示这是一个空行（直接按 ENTER 的结果），也表示这是一个多行指令的结束。

在本书中的大部分例子，都有加上注释，甚至是那些互动模式下的例子。注释（comment）在 Python 中，“#”之后的东西都是注释（跟 Perl 一样）。注释可以自成一行，也可以跟在空格符或是程序代码的后面。但是，如果“#”是在字符串常数（string literal）之中的话，就不代表注释的意义，而只是一个普通字符罢了。

底下是一些例子：

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

语法特点：

- 1) 变量不用先定义类型，直接由系统识别（大多解释性语言都这样）。
- 2) 用行头空格数来标识块的起始，省去了 Perl/C/C++ 的 {}，没有 {} 匹配问题应该是省了不少头痛的，这一点刚开始会不太习惯，熟了后真是很感谢 G.V. Rossum 的发明。

c) 没有了 Perl 中的 \$%& 等符号。（为方便，下面将变量的结果放在 ==> 的右边）。

2. 1. 字串（单引号等于双引号）：

```
a = 'Well well study' + ', Day day up!'
```

则：

```
a[3] ==> 'l'
```

```
a[-1] ==> 'l'
```

```
a[0:-1] ==> 'Well well study, Day day up!'
```


2. 2. 数列 (List), 相当于数组, 可以含不同类型

```
x = ['abc', 4, 'python']
```

则:

```
x[0] ==> 'abc'
```

```
x[1] ==> 4
```

2. 3. 字典 (Dictionary), 可以用字符串作指标

```
mathScore = { 'xah': 59, 'tinybit': 99 }
```

则:

```
mathScore['tinybit'] ==> 99
```

2. 4. 函数, 可以用缺省参数:

```
def score (user='tinybit', whichExam='yesterday') :  
    if whichExam != 'yesterday': return 0  
    if user=='tinybit': return 99  
    else: return 59
```

于是:

```
score () ==> 99
```

```
score ('xah') ==> 59
```

还可以用关键词来调用:

```
score (whichExam='tomorrow') ==> 0
```

3) 模块。

4) 类 (Class)。

5) 图形界面编程 (GUI), 其缺省为基于 TCL/TK 上的 Tkinter, 不过它的功能不是很多; 基于跨平台 wxWindows 库的 wxPython 功能要强不少 (尚未中文化, 有待各位努力)。还有基于 Qt 和 Gtk 库的, 不过 Gtk 好像目前还不能在 Windows 上很好的运行, Qt 虽能, 也很成熟, 但若在 Windows 上运行它原则上要向 TrollTech 上交\$1, 500 的, 不像 wxWindows 完全免费。

3.1 把Python当作计算器来用

现在我们来试一试一些简单的Python指令吧。请先启动Python的直译器并且等待primary prompt (“>>>”) 的出现 (应该不会很久的)。

3.1.1 数字

直译器就好像一个计算器一样：你可以打入一个表示式（expression），然后直译器会把这个 expression 的执行结果显示出来。Expression 的语法都很简单直接，一般的运算符 +, -, * 以及 / 的用法就跟其他的程序语言（像是 Pascal 或 C）一样。你也可以用括号 “()” 来表示运算执行的先后次序。例子如下：

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

跟 C 语言一样，等于符号（“=”）其实是表示设定某个值给一个变量的意思。虽然设定（“=”）运算本身是有结果值的，但是直译器并不会输出其结果来。

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

一个值是可以同时设给许多变量的：

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

```
0
>>> z
0
```

浮点数的运算在 Python 里面也是支持的，如果整数与浮点数（带小数点或 e 的数）进行运算的话，整数部分会先转换（convert）成浮点数再进行运算。

```
>>> 4 * 2.5 / 3.3
3.0303030303
>>> 7.0 / 2
3.5
```

甚至连复数的运算也支持，只需要把虚数部分加上“j”或是“J”在其后就可以了。如果实部不为零的话，复数的写法就写成“(real+ imagj)”。或者，我们也可以用函数的方式来表示复数为“complex (real, imag)”的形式。

```
>>> 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)
```

复数的虚数部分及实数部分的值都是以浮点数(float point numbers)来表示的，如果 z 代表一个复数的话，你可以很轻易的用 z.real 以及 z.imag 得到一个复数的实数部分及虚数部分。

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

0.5

复数没有办法直接用 (float(), int()或是 long()) 转换成浮点数或是整数。事实上, 复数没有直接对应的实数, 你必须用 abs(z) 来得到 z 的 magnitude (以浮点数表示), 或是如上所述用 z.real 直接得到其实数部分。

```
>>> a=1.5+0.5j
>>> float(a)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use e.g. abs(z)
>>> a.real
1.5
>>> abs(a)
1.58113883008
```

在互动模式之下, 最后一个印出来的 expression 值会储存在一个特殊变量 “_” 之中。这表示, 当你把 Python 的直译器来当作计算器用的时候, 想要连续做运算其实是方便许多的。如下例:

```
>>> tax = 17.5 / 100
>>> price = 3.50
>>> price * tax
0.6124999999999999
>>> price + _
4.1124999999999998
>>> round(_, 2)
4.1100000000000003
```

对于用户来说, “_” 这个变量是一个只读的变量。你没有办法设定一个值给它, 当你这样做的时候, 事实上你是重新创造一个同名的变量, 但是跟之前系统内建的 “_” 这个变量是一点关系也没有的了。

3.1.2 字符串

除了数字之外, Python 也有能力处理字符串 (string)。字符串在 Python 中有很多种表达

方式，它可以放在双括号 “” 之中，也可以放在单括号 ‘’ 里面：

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

字符串常数 (string literals) 是可以跨越多行的，其表示方法有很多。如果要换行的话可以用 “\” 符号来表示之。如下例：

```
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.\n"
print hello
```

这个例子会印出以下的结果：

```
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.
```

你也可以用成对的三个单引号 (""") 或双引号 ("") 来表示字符串。在此情况下你所打入的 ENTER 就会直接被解读为换行符号而不需要再用 \n 了。

```
print """
Usage: thingy [OPTIONS]
    -h                Display this usage message
```

```
-H hostname           Hostname to connect to""
```

这个例子会印出以下的结果：

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

如果你打入的 `expression` 是字符串的运算，运算的结果同样会由直译器显示出来，而且显示的方式跟你直接打入字符串常数（string literals）是一样的。会在引号之中，所有有 escape character “\” 表示的字符都会依样的显示出来。如果字符串本身包含有单引号，整个字符串就会用双引号括起来，要不然就会只用单引号来把整个字符串括起来。（如果你使用 `print` 这个语句（statement）来印出字符串的话，屏幕的输出就不会有引号出现，而且字符串中的 escape character（“\” 表示的特殊字符）都会显示出其所代表的意义来。）

字符串可以用 `+` 这个操作数来相加（连接起来），或是用 `*` 这个操作数来重复之。请看例子：

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

如果你把两个字符串常数放在一起，它们自动就会相加起来。所以，上面的例子的第一行也可以写作“`word = 'Help' 'A'`”。不过这个方法只适用于两个字符串常数的相加，其他情况就不适合了。请看例子：

```
>>> import string
>>> 'str' 'ing'                # <- This is ok
'string'
>>> string.strip ('str') + 'ing'  # <- This is ok
'string'
>>> string.strip ('str') 'ing'    # <- This is invalid
File "<stdin>", line 1
    string.strip ('str') 'ing'
```

```
SyntaxError: invalid syntax
```

如同在 C 语言中一样,字符串是有标记(subscript(index))的,第一个字符的标记(subscript(index))就是 0。在 Python 中没有另外一个字符 character 数据类型,一个字符就是一个长度为 1 的字符串。就像是在 Icon 语言一样,字符串可以用其 subscript(index)来切出(slice notation)中的一部分的,其语法为 ""。

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

与 C 不同的是,Python 的字符串是不可改变的 (immutable),如果你想要改变其中的一个字符或是一个部分 (slice),你会得到一个错误的信息:

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

但是你可以任意使用一个字符串的一个字符或是一个部分 (slice) 来创造出另一个字符串,这是完全可行的:

```
>>> 'x' + word[1:]
'xclpA'
>>> 'Splat' + word[-1:]
'SplatA'
```

当你用字符串切割 (string slice) 的语法时,可以使用其默认 (default) 的 subscript(index) 值,这是很方便的。第一个 subscript(index) 的默认值是 0, 第二个 subscript(index) 的默

取值则是这个字符串的整体长度。

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

所以，`s[i] + s[i]` 会恰好等于 `s`。你可以想一想为什么：

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

如果你用一些奇怪的 `index` 来切割字符串，Python 直译器也都处理得很好。如果第二个 `index` 太大的话就自动替换为字符串的长度，如果第二个 `index` 比第一个 `index` 还要小的话就自动返回一个空字符串。

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

字符串的 `index` 甚至可以是负数，若是负数的话，就必须从字符串的尾巴开始算起。如下例：

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


但是 -0 事实上是等于 0，所以不会从尾巴开始算起。

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

如果负数 index 超过字符串的范围的话，就自动会到最大可能的范围，但是如果不是切割一部分的话就会造成错误的情形：

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

最好避免错误的方法是把 index 看成是指向字符及字符间位置的指针，字符串的最开头是 0，字符串的结尾处就是字符串的长度。如下图所示：

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

上图的数字部分第一行代表的是正数的 index，由 0 到字符串的长度，第二行代表的是负数的 index。字符串的切割 (slice) 很容易就可以看出来，就是两个 index 之间的所有字符组合成的字符串。

对于正数的 index 来说，如果两个 index 都在范围之内，字符串的切割 (slice) 的长度就正好是其两个 index 相减的结果。举例来说 word[1:3] 的长度就正好是 2。

Python 内建的 len() 函数可以帮助我们得到字符串的长度值。

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

3.1.3 Unicode 字符串

从 Python 2.0 开始 Python 支持一种新的储存文字资料的数据类型：Unicode 对象（object）。使用这个对象你可以储存并控制 Unicode 的资料（详见 <http://www.unicode.org>），并且这个对象跟已经存在的字符串（string）对象是完全可以互相整合，并且在需要时可以互相转换的。

使用 Unicode 的好处是可以处理各种不同国家语言的字符。在 Unicode 之前，一个 code page 里只有 256 个字符可以使用在 script 中。这个限制的结果常常造成软件国际化（internationalization，通常写作“i18n”——“i”+ 18 个字符 + “n”）时候的困扰。Unicode 的出现定义一个所有 script 都可以用的 code page，如此就解决了这个问题。

在 Python 中要创建一个 Unicode 字符串就跟创建一个普通字符串一样容易：

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

在引号之前小写的 “u” 代表这个字符串是一个 Unicode 字符串。如果你要用到特殊字符，你可能要使用 Python 的 Unicode 特殊字符编码（Unicode-Escape encoding）。下面的范例示范就告诉你如何使用：

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

上面的 u0020 表示在这个位置要插入一个由十六位 0x0020 所代表的 Unicode 字符（就是空格符）。

其他的字符也会被解读为其对应的 Unicode 字符。由于 Unicode 对应中的前 256 个 Unicode 字符正好就是大部分欧美国家使用的 Latin-1 编码字符，所以其转换更加的容易。

对于专家们来说，有一个字符串的原始模式（raw mode）可以使用。你必须再加上一个小写 ‘r’ 来使 Python 使用这一个原始的 Unicode 特殊字符编码（Raw-Unicode-Escape encoding）。只有当 uXXXX 之中的小写 ‘r’ 有奇数的 ‘\’ 时才会用到这一个编码的。

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

这个原始模式 (raw mode) 通常用在字符串里面有一大堆的反斜线 ‘\’ 时, 例如 regular expressions (正规表示) 时就常用到。

除了这些标准的编码之外, Python 还提供了一整套的方法让你可以从已知的编码中创造出 Unicode 字符串来。

Python 内建的 unicode()、p() 函数可以让你使用所有已注册的 Unicode 译码/编码系统 (codecs (COders and DEcoders))。这个 codes 可以与大部分系统互相转换, 包括 Latin-1, ASCII, UTF-8 以及 UTF-16 等等。上面所提到的最后两种系统是可变长度的编码系统, 可以来储存 8 位及 16 位的 Unicode 字符。Python 预设使用 UTF-8 为预设编码系统。当你印出 Unicode 或是将 Unicode 写入文件时都会使用到。

```
>>> u"äöü"
u'\344\366\374'
>>> str (u"äöü")
'\303\244\303\266\303\274'
```

如果你要使用一个特别的编码系统, 但是要印出对应的 Unicode 码时, 你可以使用 unicode() 函数, 加上这个编码系统的名称当作第二个参数。

```
>>> unicode ('\303\244\303\266\303\274', 'UTF-8')
u'\344\366\374'
```

如果要把 Unicode 字符串转换为一般的字符串编码时, 可以使用 Unicode 对象的 encode() 方法 (method)。

```
>>> u"äöü".encode ('UTF-8')
'\303\244\303\266\303\274'
```

3.1.4 列 (List)

Python 能够了解一些较为复杂的数据类型, 这些数据类型大多是用来处理一群的其他数据值。最方便使用的要算是 list 了, 一个 list 可以写成一串由逗号分开的值 (东西), 然后用角括号括起来便成。放在 list 里的东西不需要是同一个数据类型

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

跟字符串的 index 用法相同，list 的 index 也由 0 开始，同样你可以用 index 来切割 lists、组合两个 list 等等：

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

与字符串不相同的是，字符串的个别字符是不可变动的（immutable），但是 list 的个别成员是可以自由改变的。

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

你也可以设定一个值或是一个 list 给一个 list 的切割部分（slice），但是这样的结果会改变整个 list 的长度：

```
>>> # Replace some items:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Remove some:
... a[0:2] = []
>>> a
[123, 1234]
```

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

内建的 `len()` 函数仍然可用在 `list` 上面:

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

一个 `list` 也可以是另一个 `list` 的成员 (这叫作嵌套 `list`, `nested list`), 参考下例:

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

注意前一个例子, `p[1]` 以及 `q` 事实上指的是同一个对象。我们下面还会再讨论对象的语法 (`object semantics`)。

3.2 迈向程序设计的第一步

当然 Python 能做比 2 加 2 更有用更复杂的事, 例如说, 我们可以写一个程序来印出费氏数列 (`the Fibonacci series`) 来:

```
>>> # 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
```

这个范例告诉了我们很多新的事情：

1. 程序的第一行是一个多重设定（multiple assignment）：两个变量 `a` 以及 `b` 同时都设定了新的值 `0` 与 `1`。程序的最后一行再次使用这个技巧，这次在设定符号（等号）的右边我们使用了 `expression`，所有在右边的 `expression` 会先求得其值（`evaluate`）然后才进行设定（`assign`）的动作。对于在右边的 `expression` 来说，其 `evaluate` 的次序则是由左至右的。
2. 在 `while` 循环中，只要条件符合（在这里是 `b < 10`），这个 `while` 循环就会一直执行。与 `C` 相同的是，对 `Python` 而言只要是非零的整数都表示在决定 `true/false` 的情况下代表 `true`，`0` 则代表 `false`。我们也可以在循环条件的地方放入字符串或是一个 `list`，只要这个字符串或 `list` 的长度不是零就代表 `true`，若是空字符串或空的 `list` 就代表 `false`。在这个例子里，我们比较两个值的大小。比较的操作数与 `C` 是完全相同的：`<`（小于），`>`（大于），`==`（等于），`<=`（小于或等于），`>=`（大于或等于）以及 `!=`（不等于）。
3. 在循环中的执行部分是缩排的：缩排在 `Python` 中是表示一群语句的方法（`way of grouping statements`）。`Python` 没有（还没有）提供够聪明的行排版机制，所以每个要缩排的行你都得打入空格键或是 `tab` 键。在实际的工作环境中，你也许会有自己的文字编辑器，大部分的编辑器会自动帮你做缩排的工作。当在互动模式下输入一个复合的 `statement` 时（一个由许多 `statements` 组合成的 `statement`），最后还需要打入…

个空白行（译：按 ENTER 键）来告诉直译器这个 statement 已经完成了（直译器没办法猜你什么时候完成这个 statement）。值得注意的是，如果你的 statement 是属于同一群（block）的话，你缩排的距离就要是一样的。

4. print 这个语句会印出一个 expression 的结果值，这点与我们之前所做的仅仅打入 expression 是不同的。不同之处在于对字符串及多个 expression 来说，用 print 不会印出字符串的引号，还会在多个 expression 之间印出空白来，这样会让结果好看一点。如下所示：

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

如果不想每次的输出都换行的话，可以在 print 语句之后加上逗号，如下所示：

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

值得注意的是如果最后一行还没有完成的话，直译器会在印出 prompt 之前印出新的一行。

第4章 变量、运算符和表达式

本章将介绍有关数值、变量和操作符的概念。数值是直接放在源码中的数，并且程序不能改变它的值。变量放在保存程序数据的计算机内存中。之所以叫变量，是因为可以按需要赋予不同的值。常量从本质上说是命名的数字（例如 π ），它对程序归档来说有用，并且使代码更容易安排。操作符是告诉计算机要做什么操作。

下面我们来认识 Python 的一些数据类型及其他标识。

4.1 Python语言的基本数据类型

使用计算机工作，简单时就像在学校里写作业，复杂时又像是在解量子理论里的方程式，但不论怎样，计算机做的工作都是处理数据。对于计算机来说数据可以是数、字符或简单的值。Python 使用了几种数据类型，在本章中将介绍其中最重要的部分。

Python 对于类型的定义类似 Java，具体的格式如下：

```
from types import *
def delete (list, item) :
    if type (item) is IntType:
        del list[item]
    else:
        list.remove (item)
```

Python 有很多种基本内置类型，每种类型都有各自的用途和用法，下面介绍几种最常见的内建类型。

1) NoneType

None 类型，这是最简单的一种类型，它的变量赋值就类似于其他语言中的 NULL 值。

2) TypeType

自定义类型。

3) IntType

整型。

4) LongType

长整型。

5) FloatType

实型，使用 IEEE754-1985 标准 32 位单精度浮点书。

6) ComplexType

复合型。

7) StringType

字符型。

8) UnicodeType

Unicode 字符。

9) TupleType

元组类型。

10) ListType

列表类型。

11) DictType

字典类型。

12) FunctionType

函数类型。

4.2 标识符和关键字

标识符是用来标识某种数值的符号，用户可以选择任意一组字符作为标识符来代表任何变量，如常数、类对象等。标识符一经建立，在同一个代码块中它就代表着相同的对象。Python 语言的标识符的组成比较简单，它不限制标识符命名长度，用户可以根据需求定义标识符。但是用户在定义标识符的时候，必须遵行一些规则：

1. 第一个字节必须是字符。而后面跟着的可以是字符也可以是数字。
2. 这些开头字符不能是拉丁数字或普通数字，它们可以是字符标大小写英文字母中任意字符。
3. 下划线（_）可以用在标识符中。
4. 像在 C 和许多其他语言一样，在 Python 中标识符区分字符大小写的。例如：
The_first_Val 和 the_First_Val 是不同的标识符。大写的标识符变量和小写的标识符变量是不

同的。

5. 在描述定义标识符时，最好能根据语义。所以标识符要能够表达完整的含义，不要怕长。短的标识符容易输入、输出得快捷、不易出现错误，而输入长的标识符则更具有描述能力和可读性，用户在命名标识符时应兼顾这两方面的因素。在大的应用程序中选一种命名约定是很有用处的，这样可以减少相像和避免标识符重用（即一个标识符代表两个对象）。下面的做法是不提倡的，创造几个名为 X、X1、X2 和 X4 的标识符，用户很难记住每个变量的用途。另外标识符不能是关键字。

6. 表 4.1 显示了几个合法的标识符和几个非法的标识符。第一个非法的标识符是因为它的数字开头。第二个非法标识符是因为它包含有非法字符。第三个非法标识符也用了非法标识符：空格。第四个非法标识符使用的是数字，数字不能用作标识符。最后一个非法标识符包含有另一个非法字符：连字符，或叫减号。Python 将最后一个非法标识符是当作表达式来处理的，它认为这是一个由两个量执行减法运算的表达式，其中含有两个标识符和一个运算符。

表 4.1 合法标识符和非法标识符的示例

合法标识符	非法标识符
Phonebook	2Phonebook
Yellow	Yellow\$bar
Last	Last bin
Start_on	12345
Been_Ls	Been-Ls

在 Python 语言中有很多标识符是系统自定义的，不能将它们作为变量或函数的名称。表 4.2 列举了 Python 的保留标识符。

表 4.2 Python 的关键字

And	Del	for	is
raise	assert	elif	from
lambda	return	break	else
global	not	try	class
except	if	Or	while
continue	exec	import	pass
def	finally	In	print

4.3 声明变量

在进一步学习之前，用户应该了解 Python 的变量定义机制。Python 一般很少定义变量，其定义方式也和其他语言有所不同，Python 在特殊情况下才会引用内置数据类型转换函数对数值进行转换。当用户在定义转换 Python 的数据类型时，必须知道几件事：

- 1) 用户必须知道要使用的数据类型。在前面我们介绍了 Python 的一些常用内置数据类型，其实 Python 的数据类型非常丰富，如果要深入研究，用户可以在 Python 的官方网站上去查询。
- 2) 用户必须知道什么东西要调用变量。
- 3) 用户也可能要知道变量的初值。

4.4 字符、字符串变量

Python 的字符和字符串类型定义比较自由，没有严格意义上的区分。在定义初值时系统会自动分配相应得存储空间。Python 的字符变量要放在单引号 (') 内，如 'A'、'a'、'!' 是字符变量。反斜杠 (\) 称作 escape 符，表明后面的字符是一个特殊字符。例如，字符串里，\" 表示双引号，\' 表示单引号。\\n 是换行符，使输出设备定位到下一行的行首（与打印机上的回车键类似）。字符 \\ 表示了其本身。表 4.3 归纳了这些字符。

表 4.3 特殊字符

字 符	含 义
\\	\\本身
\'	单引号
\"	双引号
\a	响铃声
\b	将光标左移一个字符
\f	走质换页
\n	转到下一行
\r	回车
\t	水平制表符
\v	垂直制表符
\xhh...	十六进制

例 4.1 首先定义并符值下列字符变量

```
>>>char1 = 'a'
>>>char2 = '\\ '
>>>char3 = '\ '
>>>char4 = '\" '
>>>char5 = '\a'
>>>char6 = '\b'
>>>char7 = '\f'
>>>char8 = '\r'
>>>char9 = '\n'
>>>char10 = '\x23'
```

然后，用户对这些字符变量进行打印并输出结果：

```
>>>print char1
a
>>>print char2
\
>>>print char3
'
>>>print char4
"
>>>print char5
(null)
“嘟！”一次扬声器声
>>>print char6
(null)
看似屏幕无输出内容，但实际在是一个Backspace
>>>print char7
♀
屏幕输出上面符号实际上是走纸换行符
>>>print char8
(null)
屏幕输出一个空行
```

```
>>>print char9
      (null)
      (null)
      屏幕输出空行并到下一行
>>>print char10
      #
      屏幕输出 (#) 号
```

前面我们说过 Python 的字符和字符串定义是没有严格意义上的区分的。实际上 Python 对引号的要求并不严格，用户可以用单引号给变量附字符，也可以用单引号给变量附字符串，同样可以用双引号来实现。

Python 的字符串可以自由地组合而不用加 (+) 来组合，而两个字符或字符串变量组合时必须用加号来连接。

例 4.2

```
>>>str1 = "hello"
>>>str2 = "world"
>>>print str1+str2
helloworld
```

屏幕输出 "helloworld"。用户也可以这样定义：

```
>>>str3 = "hello" "world"
>>>str4 = "hello" "world"
>>>str5 = "hello" 'world'
```

实际上以上三个变量都是一样的，在字符串变量的定义过程中，如果两对引号中间有空隔那么系统会自动地删除。我们可以打印上面三个变量看看结果：

```
>>>print str3
helloworld
>>>print str4
helloworld
>>>print str5
helloworld
```

在实际的变量赋值中，我们必须注意前面所讲的逃逸字符，因此对于字符串的处理必须额外的留心，在遇到字符串中有引号时，我们注意用转意来处理。

例 4.3 如果不进行字符转意处理我们看看结果：

```
>>>print 'It's beautiful!'
File "<stdin>", line 1
print 'It's beautiful'
      ^
SyntaxError: invalid syntax
```

以上输出系统报错，我们把上面语句经过处理后输出：

```
>>>print 'It\'s beautiful!'
It's beautiful
```

这样的输出才会正确，当然上面这个例子我们仅仅是用来举例而设，如果合理处理单双引号同样可以解决上面这个问题，但我们还是提倡用字符转意来处理类似的问题。

4.5 数值类型

Python 有四种数值类型：整型、长整型、浮点型和 Imaginary 型。Python 的数值定义比较自由，用户在定义变量时可以直接赋值。

1. 整型、长整型

Python 和绝大多数编程语言一样，整数是最基本的数据类型，整数表示计算机取值范围内的所有整数。不同的计算机所能表示的整数范围不同。Python 语言中允许使用十进制整数，也可以使用十六进制的整数。

十进制整数的书写与数学上的整数十分相似，要注意的是：一般情况下一个十进制整数必须以非零数字开始（整数零例外）。

以下各数都是十进制整数：

```
0
7
123
```

而以下几个数都不是十进制整数：

```
2.1
4.01
0x22
-88
```

其中 2.1 和 4.0 是浮点型数值，0x22 是十六进制整数。而 -88 在 Python 语言中是一个表达式，其中“-”号是一个运算符。

同样，在 Python 中也定义了长整型，只不过在定义的时候用户必须在数值的后面跟上小写的“l”或是大写的“L”如下：

```
1231
344556L
90000L
```

对于用大小写作为后缀并无区别，但是从字型上我们有时很难区分小写的“l”和阿拉伯数字“1”，所以我们强烈建议用户在定义自己的长整型数值或赋值时使用大写的“L”。

Python 语言允许用户在定义负整数，例如，一个整数类型的变量 i ，其取值范围描述如下：

$$N_{\min-\text{int}} \leq i \leq N_{\max-\text{int}}$$

其中 $N_{\min-\text{int}}$ 和 $N_{\max-\text{int}}$ 的值与机器有关。在 32 位字长的机器上一个整数类型的变量分配一个机器字，而

$$1\text{字} = 4\text{字节} = 32\text{位}$$

所以，一个字可以表示 2^{32} 种不同的状态，其中一半状态用来表示非负整数

$$0, 1, 2, \dots, 2^{31} - 1$$

另一半状态用来表示负整数

$$-1, -2, -3, \dots, -2^{31}$$

因此，在 32 位机器上

$$N_{\min-\text{int}} = -2^{31} = -2147483648$$

$$N_{\max-\text{int}} = 2^{31} - 1 = 2147483647$$

假设我们定义了如下：

```
>>>a = 20000000000
>>>b = 20000000000
>>>c = a + b
Traceback (most recent call last)
  File "<stdin>", line 1, in ?
OverflowError: integer addition
```

虽然在语法上完全正确，但由于 $a+b$ 的结果已经超出 $N_{\max-int}$ ，所以执行的结果是不正确的。这种情况称为溢出，溢出后的 Python 将警告输出出错信息，有时在程序运行过程中并不给出任何错误信息，但计算结果不正确。所以在编制程序时要注意防止溢出。

在 Python 语言中，如果处理的整数变量超过 32 位，这时可以考虑 long 类型。用户可以在表示的整数类型后面加上标志符“L”。

```
>>>a = 123L
>>>b = 200000000000000L
>>>c = a + b
```

以上都是合法的表达式。

2. 浮点型

Python 语言提供了 float 浮点类型。

```
2.003
3.14152
5.00001
```

以上都是浮点类型数值。和 C 语言对浮点型数值定义略有不同的是，Python 语言允许浮点型数值带有负号。

但浮点型的数值表示方式和其他语言类似：

```
1.092332e5
```

代表 $1.092332 \times 10^5 = 109233.2$ 。类似地

```
1.092332e - 3
```


代表 0.001092332。

下面我们就来较为精确地来描述这种指数方式。浮点数

```
111.2222e33
```

为叙述方便，可分解成四个部分：

	整数部分
	小数点
	小数部分
e-33	指数部分

这里每一部分都不能嵌入空白和其他字符。一个浮点数必须包含小数或包含指数部分，并且如果出现小数点的话，那么整数部分和小数部分不能全部没有，如果没有小数点的话，必须有整数和指数部分。也就是说，在 Python 语言中允许出现的浮点数值有以下几种形式：

整数部分.小数部分	指数部分
整数部分.小数部分	
整数部分.	
.小数部分	
.小数部分	指数部分
整数部分指数部分	

其中整数部分和小数部分都是非空数字串，而指数部分则是由 e（或 E）作前导的一个非空数字串，在 e（或 E）与数字串之间允许加入一个正负号时，按正号处理。

以下是一些浮点数值例子。

```
3.14159
3.14.159e-2
314159e-5
0.00315159e+3
0.00314159e3
.314159e003
0e0
.0e0
0.
1.0
```

1e0

以下不是浮点数值:

3.14.1592	(不允许出现逗号)
3.1415 9	(不允许加入空白)
314159	(这是一个整型)
.e0	(整数部分和小数部分不能全部省略)

3. Imaginary 型

Python 语言定义了 Imaginary 数值类型, 它实际上扩充了 float 浮点类型。类似于双精度浮点类型。在这里就不再详细描述。

第5章 Python 数据结构

本章更详细地讨论一些已经讲过的数据类型的使用，并引入一些新的类型。

5.1 列表

列表数据类型还有其他一些方法。下面是列表对象的所有方法：

- 1) `insert(i, x)`—— 在指定位置插入一项。第一个变量是要在哪个元素前面插入，用下标表示。例如，`a.insert(0, x)`在列表前面插入，`a.insert(len(a), x)`等价于 `a.append(x)`。
- 2) `append(x)`—— 等价于 `a.insert(len(a), x)`。
- 3) `index(x)`—— 在列表中查找值 `x` 然后返回第一个值为 `x` 的元素的下标。没有找到时出错。
- 4) `remove(x)`—— 从列表中删去第一个值为 `x` 的元素，找不到时出错。
- 5) `sort()`—— 对列表元素在原位排序。注意这个方法改变列表，而不是返回排序后的列表。
- 6) `reverse()`—— 把列表元素反序。改变列表。
- 7) `count(x)`—— 返回 `x` 在列表中出现的次数。

下例使用了所有的列表方法：

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

```
>>> a
[66.6, -1, 333, 1, 1234.5, 333]
>>> a.reverse ()
>>> a
[333, 1234.5, 1, 333, -1, 66.6]
>>> a.sort ()
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
```

函数程序设计工具

Python 中有一些函数程序设计风格的东西，例如前面我们看到的 `lambda` 形式。关于列表有三个非常有用的内置函数：`filter()`，`map()`和 `reduce()`。

“`filter` (函数, 序列)” 返回一个序列（尽可能与原来同类型），序列元素是原序列中由指定的函数筛选出来的那些，筛选规则是“函数（序列元素）`=true`”。`filter()`可以用来取出满足条件的子集。例如，为了计算一些素数：

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

“`map` (函数, 序列)” 对指定序列的每一项调用指定的函数，结果为返回值组成的列表。`map()`可以对序列进行隐式循环。例如，要计算三次方，可用：

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

可以有多个序列作为自变量，这时指定的函数必须也有相同个数的自变量，函数从每个序列分别取出对应元素作为自变量进行调用（如果某个序列比其他的短则取出的值是 `None`）。如果指定的函数是 `None`，`map()`把它当成一个返回自己的自变量的恒同函数。在函数用 `None` 的情况下指定多个序列可以把多个序列搭配起来，比如 “`map (None, list1, list2)`” 可以把

两个列表组合为一个成对值的列表。见下例：

```
>>> seq = range (8)
>>> def square (x) : return x*x
...
>>> map (None, seq, map (square, seq))
[ (0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49) ]
```

“reduce (函数, 序列)” 用来进行类似累加这样的操作，这里的函数是一个两个子变量的函数，reduce()先对序列的前两项调用函数得到一个结果，然后对结果和序列下一项调用函数得到一个新结果，如此进行到序列尾部。例如，要计算 1 到 10 的和：

```
>>> def add (x, y) : return x+y
...
>>> reduce (add, range (1, 11))
55
```

如果序列中只有一个值则返回此值，序列为空时会产生例外。可以指定第三个自变量作为初始值。有初始值时对空序列函数将返回初始值，否则函数先对初始值和序列第一项作用，然后对结果和序列下一项作用，如此进行到序列尾。例如：

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

5.2 del语句

上面我们看到，列表的 remove()方法可以从列表中删去某个取值的项，我们还可以用 del 语句来删除指定下标的项。也可以用 del 语句从列表中删除一个片断（前面我们是用给片断赋空列表的办法删除片断的）。例如：

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

`del` 也可以用来删除整个变量，例如：

```
>>> del a
```

变量删除以后再引用该变量就会出错（除非又给它赋值了）。后面我们还会看到 `del` 的其他一些应用。

5.3 序表和序列

我们看到列表和字符串有许多共同点，例如，下标和片断运算。它们都属于序列数据类型。因为 Python 是一个正在不断发展的语言，以后还可能会加入其他的序列数据类型。现在还有一种标准的序列数据类型，称为序表（tuple）。

序表由一系列值用逗号分隔而成，例如：

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # 序表允许嵌套：
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

输出的序表总是用括号包围，这样可以保证嵌套序表得以正确解释。输入时可以有括号也可以没有括号，当经常是必须有括号（如果序表是一个大表达式的一部分）。

序表有许多用处，例如，(x, y)坐标对、数据库中的职工记录等等。序表与字符串一样是不可变的，不允许对序表的某一项赋值。

生成序表时对 0 项或 1 项的序表有特殊的规定，空序表用一对空括号表示；只有一项的序表用一个值后面跟一个逗号表示（只把这个值放在括号内是不够的）。这样写不够美观，但很有效。例如：

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

语句 `t = 12345, 54321, 'hello!'` 是序表打包的一个实例：12345, 54321 和 'hello!' 这些值被打包进了一个序表中。相反的操作也是允许的，例如：

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

这叫做序表解包。序表解包要求等号左边的变量个数等于序表的长度。注意多重赋值只是序表打包和序表解包的联合使用。有时也对列表进行类似操作，即列表解包。只要把各变量写成一个列表就可以进行解包：

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> [a1, a2, a3, a4] = a
```

5.4 字典

Python 内置的另一个有用的数据类型是字典。字典在其他语言中有时被称为“关联记忆”或“关联数组”。字典不像序列，它不是用在一个范围之内的数字下标来索引，而是用键值来索引，键值可以是任何不可变类型。字符串和数值总可以作键值。如果序表只包含字符串、数值或序表则序表也可以作键值使用。列表不能用作键值，因为列表可以用其 `append()` 方法就地改变值。

最好把字典看成是一系列未排序的“键值：值”的集合，在同一字典内键值是互不相同的。一对空大括号产生一个空字典：{}。在大括号内加入用逗号分开的“键值：值”对可以在字典内加入初始的键值和值对，字典在输出时也是这样显示的。对字典的主要操作是以某个键值保存一个值，以及给定键值后查找对应的值。也可以用 `del` 删除某个键值：值对。如果用一个已有定义的键值保存某个值，则原来的值被遗忘。用不存在的键值去查找就会出错。

字典对象的 `keys()` 方法返回字典中所有键值组成的列表，次序是随机的。需要排序时只要对返回的键值列表使用 `sort()` 方法。为了检查某个键值是否在字典中，使用字典的 `has_key()` 方法。

下面是字典使用的一个简单例子：

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys ()
['guido', 'irv', 'jack']
>>> tel.has_key ('guido')
1
```

5.5 条件的进一步讨论

在 `while` 语句和 `if` 语句中使用的条件除了可以使用比较之外还可以包含其他的运算符。比较运算符“`in`”和“`not in`”可以检查一个值是否在一个序列中。运算符“`is`”和“`is not`”比较两个对象是否恰好是同一个对象，这只对对象列表这样的可变对象有意义。所有比较运算优先级相同，而比较运算的优先级比所有数值运算优先级低。

比较允许连写，例如，`a < b == c` 检查是否 `a` 小于等于 `b` 而且 `b` 等于 `c`。

比较可以用逻辑运算符 `and` 和 `or` 连接起来，比较的结果（或其他任何逻辑表达式）可以用 `not` 取反。逻辑运算符又比所有比较运算符低，在逻辑运算符中，`not` 优先级最高，`or` 的优先级最低，所以“`A and not B or C`”应解释为“`(A and (not B)) or C`”。当然，可以用括号来表示所需的组合条件。

逻辑运算符 `and` 和 `or` 称为“短路”运算符：运算符两侧的表达式是先计算左边的，如果左边的结果已知则整体结果已知就不再计算右边的表达式。例如，如果 `A` 和 `C` 为真而 `B` 为假，则“`A and B and C`”不会计算表达式 `C`。一般地，当短路运算符的运算结果不是用作逻辑值的时候，返回的，而是最后求值的那个表达式的值。

可以把比较或其他逻辑表达式的结果赋给一个变量。例如：

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

注意 Python 和 C 不同，表达式中不能进行赋值。

5.6 序列与其他类型的比较

序列对象可以和其他同序列类型的对象比较。比较使用字典序：先比较最前面两项，如果这两项不同则结果可以确定；如果这两项相同，就比较下面的两项，如此下去，直到有一个序列到头为止。如果某两项本身也是同类型的序列，则进行递归的字典序比较。如果两个序列的所有各项都相等，则这两个序列相等。如果一个序列是另一个序列的初始子序列，短的一个是较小的一个。字符串的字典序比较按各个字符的 ASCII 次序进行。下面是一些序列比较的实例：

```
(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)
```

注意不同类型的对象比较目前也是合法的。结果是确定的，但却没有什么意义，不同类型是按类型的名字排序的。所以，列表（list）总是小于字符串（string），字符串总是小于序表（tuple），等等。但是程序中不能依赖这样的比较规则，语言实现可能会改变。不同的数值类型可以按数值来比较，所以 0 等于 0.0，等等。

第6章 控制流

控制执行流对任何语言来讲都是很重要的一个方面。程序就好像有感知的生物一样，必须能操作自己的世界，在执行的过程中做出判断和选择。因此，如果编程人员编程时迷失了方向，程序就可能随机地选择方向。否则程序就将定制的方向执行程序。

正如多数编程语言一样，Python 有常用的控制语句集。如果你熟悉 C 或其他编程语言，就会发现 Python 的控制语句和一般语言稍稍有些区别，它更接近 shell 脚本。因此，当你编写 Python 程序的时候更会感觉到 Python 编程的简洁和灵活。

6.1 if语句

在 Python 中，if 句型的 if 行是测试条件，紧接着的是当条件为真时执行的语句。如果条件为假，执行 else 语句。If 语句的语法是：

```
if <condition>:
    statements
else:
    statements
```

当然在实际编程中 elif 和 else 部分的语句是可选的。请注意在 if 和 else 行结尾的冒号，它表示下一行是本部分的继续。但和其他一些语言有区别的地方是 Python 的 if 句型没有中断语句，如 endif 或 fi。

大部分编程语言是用括号或一些特定语句或符号来中断每一个代码块，而 Python 是根据语句的缩进或空行来判断代码块开始执行和结束的地方。在一个代码块中，用来缩进语句的空格，只要它们是连续的，数目多少并不重要。下面的代码片断是一个 if 语句的示例：

```
>>> x = int (raw_input ("Please enter a number: "))
>>> if x < 0:
...     x = 0
...     print 'Negative changed to zero'
```

```
...     elif x == 0:
...         print 'Zero'
...     elif x == 1:
...         print 'Single'
...     else:
...         print 'More'
```

使用条件语句时要注意一些 Python 特有的用法。Python 的条件语句中允许所有常用的数值比较（`==`, `!=`, `>`, `>=`, `<`, `<=`）。但是，在 if 语句的条件判断中，不能用单个等于号。单个等于号是用来赋值，而不是用来比较的。还有，特别要注意的是 Python 条件语句中逻辑与是用 `and` 不是 `&&`，而逻辑或是用 `or` 不是 `||`。

Python 中的 if 语句不测试真实值，它只简单地判断对或错。如果条件为真，就返回一个非零值，若为假，返回零。根据返回值决定执行 if 语句还是 else 语句（如上例）。

Python 允许在 if 语句中使用一个或多个 elif 语句，这点类似 java，但 Python 不提供 switch 和 case 语句，因此在多分支判断时一般都用多个 elif 来完成多条件判断。

虽然在程序代码中有太多 if 语句的循环嵌套时会造成死机，但理论上 Python 允许在 if 语句中有无限的循环嵌套。可是我们不支持实际编程中使用太多 if 语句的循环嵌套，如果必须用到 3 到 4 级嵌套，就应该尽量使用更好的方法编写该部分代码。

当 if 语句的任一部分执行完以后，Python 的解释程序都跳到 if 语句的结束处。这意味着可能从不执行一个嵌套的 if 语句的第二部分。例如下面例子所示，当第一个判断值为真时，执行第一个 print 语句：

```
>>>a = 5
>>>if a > 3:
_     print "a is greater than 3"
_elif a > 1
_     print "a is greater than 1"
_else:
_     print "a is less than 2"
```

即使第二部分测试为真（elif 部分），也不执行，因为 Python 解释程序跳转到 else 部分的结束处。

Python 和其他流行语言一样，if 语句中可以包含布尔（boolean）变量。然而，必须在 Python

中写出布尔变量,而不能像在 C 和 shell 脚本中那样使用符号。Python 常用的布尔语句是 `and`、`or` 和 `not`。下面语句在 Python 中都是正确的:

```
>>>If a < 3 and b > 5:
>>>If a < b or c > d:
>>>If not a < b:
>>>If a < b or (a > 4 and b < 5) :
```

使用 `not` 否定测试,因此第三个例句与 `a>=b` 等价。最后一句使用括号增加一层判断,这是 Python 的优越之处。只要正确地使用条件和布尔变量,就能够在 Python 中创建足够长的复杂语句,而在其他编程语言中几乎不可能只用几行就创建同样复杂的语句。

在编写 Python 程序的时候通常可以把条件语句后面的执行语句压缩到一行中,例如:

```
>>>if a > 3: print "a is greater than 3"
```

当然,下面的书写方法也是正确的,如:

```
>>>if 2 < a < 5: print "a is greater than 2 and less than 5"
```

这和下面写法是等同的:

```
>>>if a > 2 and a < 5: print "a is greater than 2 and less than 5"
```

6.2 while 循环

`while` 语句测试表达式的值,如果为真,则重复执行下一个语句或语句块直到表达式的值为假。当变量或表达式为假时,控制语句就跳过 `while` 语句的下一个语句。下面是 `while` 循环的语法,它与 `if` 语句有些相似:

```
while <condition>:
    statements
```

语句可以根据程序要求有意或无意地变成死循环,只要条件始终为真就会出现这种现象。下面的示例就是一个死循环:

```
>>>while 1:
```

```
...     print "."
```

运行该示例就会不断地打点。在这里当条件为真时执行的语句块用缩进格式表明。这一点我们在变成的时候一定要注意养成良好的编程习惯。只有这样才能提高代码的有效性和可读性。

Python 允许使用一个 `break` 语句随时从 `while` 语句中退出。只要遇到 `break` 语句，解释程序立即认为条件为假，执行 `while` 块最后一行后面的语句。例：

```
>>>total, i = 0
>>>while 1:
-     total += i; i += 1
...     if i>11 or total > 55: break
...
... print "hello!"
```

上例中，`i` 和 `total` 不断地增值，当 `i` 的值大于 11 或 `total` 的值大于 55 时，`if` 条件为真，执行 `break`，由于“`hello!`”语句是 `while` 语句块后面的第一条语句，所以到达 `break` 后立即打印“`hello!`”。通常，遇到某个条件要离开 `while` 循环时就在 `if` 块中使用 `break`。使用 `break` 可以在 `while` 条件中节省大量编码。

6.3 for 循环

`for` 语句相对于 `while` 语句而言要复杂一些。程序员可以用它代替 `while` 语句来循环重复一个列表或一个串。在 Python 里的 `for` 语句的用法与在 C 或是 Pascal 里的用法有所不同。不像是在 Pascal 中一定要执行某个数目的循环，也不像是在 C 中让用户决定执行的进度 (step) 及结束执行的条件，Python 的 `for` 语句会将一个系列 (sequence，像是 `list` 或是 `string`) 里所有的成员走遍一次，执行的顺序是依照成员在 `sequence` 里的顺序，`for` 语句中的任何语句都不应该改变列表或串。Python 中的 `for` 循环格式如下：

```
for var in <list or string>:
    statements
```

正如 `if` 语句一样，Python 通过标志符可以判断 `for` 循环体的位置。解析程序也可以从 `for` 语句后面的冒号得知冒号前后的语句是连续的。语句中的 `var` 假定为列表或串中的单个值。

下面是一个简单的 for 循环示例：

```
for str in "hello world!":
    print str
```

以上代码逐个打印串中的每个字母，每个之间有一个空格，如下所示：

```
h e l l o   w o r l d !
```

我们再来看一个示例：

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

在循环的执行中改变 sequence 的内容是一件危险的事（当然，只有可变的 sequence 像 list 才能作更动），如果你真的需要在循环的执行中改变 list 的成员值，最好先复制一份这个 list 的拷贝，然后针对这个拷贝来做循环。list 的切割（slice）提供了一个简便的制作拷贝的方法：

```
>>> 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']
```

在后面小节中我们有大量的示例来描述 for 循环的用法，事实上在 Python 编程中 for 循环是最常用的一种循环语句。

6.4 try 语句

Python 语言中的 try 语句是非常特别而且有意思的语句，try 语句类似 Java 语言中 try 的

用法。但是其含义稍微有些不一样，而且 Python 的 `try` 语句要更简洁明了一些。下面是 `try` 语句的语法结构：

```
try:
    <statements>
except:
    <statements>
```

当程序执行到 `try` 语句时，程序执行 `try` 语句后面的程序组和前面的控制语句一样，`try` 后面的程序组一般以缩进来定位。所以我们再次强调在实际编写代码中一定要注意程序块的划分。程序如果执行 `try` 后面的语句成功则跳出该控制语句，如果程序组执行失败，那么将执行 `except` 后面的程序组。

下面是一个 `try` 语句的示例：

```
#!c:\python\python.exe
import imptest1
x = imptest1.today ()
print x
print dir ()
print dir (imptest1)
try:
    print _CHANGEOVER
except:
    try:
        print imptest1._CHANGEOVER
    except:
        print "Can't find _CHANGEOVER"
```

该示例可以在光盘中完整程序的一部分，在此先不用考虑改程序片的实际意义，但我们可以看到 `try` 语句一个很优秀的特点是可以嵌套使用。不过在使用嵌套时必须要注意程序块的划分，并做到子 `try` 必须以缩进为程序块开始。

Python 语言中，`try` 语句的功能是非常强大而且易于掌握的，在编写 Python 程序时经常会用到。使用 `try` 语句可以使程序更清晰，更容易判断出错的位置。鉴于本书是比较全面介绍 Python 语言，所以不能一下子更深入地来阐述 `try` 语句更加复杂的功能。如果读者想更深

入地了解 try 语句的其他用法，可以在随书光盘中参考实例。

6.5 range()函数

如果你真的需要一个循环执行一定数目的次数的话，你可以使用内建的 range()函数。这个函数会产生一个含有逐步增加数字的 list。如下：

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

在这个函数中的所传入的参数代表端点，而且这个端点不在产生的 list 之中。range(10)正好产生 10 个数值，正好是这个 list 的 index 是由 0 到 10。我们也可以让这个产生的 list 从某个数值开始，或者规定其每次增加的数值为多少（增加值也可以是负数，这个增加值也叫做 ‘step’）。

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

所以如果我们要循环一次一个 sequence 的 index 的话，我们可以用 range()配合上 len()一起使用：

```
>>> 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
```

从上面代码片段中我们可以看到，range()一般是结合列表一起使用的。用一个参数调用 range()函数时，参数代表在一个基于 0 的列表中元素的个数。用两个参数调用 range()时，第

一个参数是列表中的第一个元素，后一个参数比最后一个元素大 1。一般可以使用列表的地方就可以使用 `range()` 函数。它可以赋给变量或者作为命令结构的一部分使用，它的上限和下限既可以为正数也可以为负数。

6.6 break及continue及循环中的else子句

如同在 C 语言里一样，`break` 语句中断最靠近的一个 `for` 或 `while` 循环。

同样地，从 C 语言借过来的 `continue` 语句会中断目前执行的循环，并且执行下一个循环。

特别的是，Python 的循环有一个 `else` 子句，这个子句之后的程序代码会在整个循环正常结束的时候执行，（对 `for` 循环而言指的是 `list` 已经到底，对 `while` 循环而言指的是条件式变成 `false`）。但是，若是在非正常结束（因为 `break` 语句）的情况下 `else` 子句的程序代码就不会执行。底下的例子是一个循环，用来找出所有的质数：

```
>>> for n in range (2, 10) :
...     for x in range (2, n) :
...         if n % x == 0:
...             print n, 'equals', x, '*', n/x
...             break
...     else:
...         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
```

6.7 pass语句

`pass` 语句其实什么也不做，通常是用在当你的程序的语法上需要有一个语句，但是却不

需要做任何事的时候。例子如下：

```
>>> while 1:
...     pass # Busy-wait for keyboard interrupt
...
```

6.8 定义函数

我们可以定义一个函数，在下面这个函数定义的例子，当我们给定想要印出的范围，这个函数会印出一个费氏数列来：

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

在上例中这个 `def` 关键词代表了一个函数的定义 (*function definition*)，在 `def` 之后必须接着函数的名称以及一个用括号括起来的一连串的参数。接下来一行之后的程序代码就是函数的主体部分，而且必须是缩排的。函数的程序代码部分的第一个 `statement` 可以是一个字符串常数 (*string literal*)，这个字符串常数会被当作是函数的批注部分而叫做批注字符串 (*documentation string* 或是 *docstring*)。

有工具可以使用这个批注字符串来自动的制作出线上的或是印出来的文件，或者是让用户可以交互式的浏览程序代码。写批注是一个好习惯，所以最好养成这个好习惯，把所有的程序代码都写上批注字符串。

执行函数的时候会产生一个目前 (*local*) 的符号表 (*system table*)，这个表是用来记录函数中的所有 *local* 的变量的。更精确的来说，所有在函数中变量的设定值都会记录在这个 *system table* 中，所以当你要使用 (*reference*) 一个变量时，会先检查 *local* 的 *system table*，然后是整个程序 (*global*) 的 *system table*，然后是内建的变量名称。虽然 *global* 变量可以在

函数使用(reference),但是不能在函数之内直接的设定其值(除非是在一个 global 的 statement 中建立的)。

当函数被调用时,实际传入的函数参数是会被记录在被调用函数的 local system table 里的。因此,参数被传入时是以其值传入的(call by value)。此处的值指的是对象的参考(reference),而非对象本身的值。当一个函数调用另一个函数时,就会因此调用而建立一个新的 local system table。

当定义函数的时候,也就在目前正在的 system table 里定义了这个函数的名称。对直译器来说,这个函数名称的数据类型是一个用户自定义的函数。这个函数的值名称可以设定给另一个名称,然后这个新的名称就可以被当作是一个函数名称来使用。这个过程就是一个一般的重新命名的机制。

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

你也许认为 fib 不是一个函数(function)而是一个程序(procedure)。如同在 C 中一样,在 Python 的 procedure 指的是没有返回值的函数(function)。事实上,就技术上而言,procedure 也是有返回值的,只是所返回的是一个 Python 系统内键的值,叫做 None。通常来说,如果只返回 None 的话,直译器不会印出这一个返回值。但是,如果你真想看一看它的话,你可以这样做:

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

如果想让你的函数返回一个包含费氏数列的 list,而不是只印出来的话,其实是很简单的:

```
>>> 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]

```

如同往例，这个例子事实上示范了一些新的 Python 的特点：

1) `return` 语句使得函数返回了一个值。如果单单 `return` 没有其他的 `expression` 来表示返回的值时，就表示这是一个从 `procedure` 返回来的写法（`procedure` 到结束都没有返回值也是表示从 `procedure` 返回来）。这种写法表示返回值是 `None`。

2) `result.append(b)` 这个语句表示调用了 `result` 这个 `list` 对象的一个方法（`method`）。`Method` 是一个特别“属于”某个对象的函数，而且其名称的形式是 `obj.methodname`。在这里 `obj` 指的是某一个对象（我们也可以用 `expression` 来代替），而 `methodname` 指得是由这个对象的数据类型所定义的这个方法的名称。不同的数据类型会定义不同的方法，不同的数据类型也许所定义的方法名称会相同，但是并不会造成冲突（你可以定义你自己的数据类型及其方法，我们称之为类别（`classes`），后面会再谈到的）。在这个例子中的 `append()` 方法是在 `list` 这个数据类型中定义的。这个方法会在 `list` 的最后面加入一个新的成员，在这个例子里也可以写作“`result = result + [b]`”，效果一样，但是用方法来写有效率多了。

在定义函数的时候我们可以加入不定数目的参数，加入参数的写法有三种，是可以混和使用的。

6.8.1 预设内定参数值

最好用的一种写法是对其中的一个或多个参数给它一个特定的默认值。这样的话，当你在调用函数时，就可以不用传入参数，或是传入较少的参数了。请看下例：

```

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

```

```
print complaint
```

当你调用这个函数的时候你可以用 `ask_ok ('Do you really want to quit?')`，或者是 `ask_ok ('OK to overwrite the file?', 2)`。

设定的默认值可以是一个变量，但是这个变量在函数定义的时候就以定义时的情况 (*defining scope*) 决定 (*evaluate*) 了其值，所以以下的例子：

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

印出的结果会是 5。

注意： 这个参数默认值只有被 `evaluate` 一次，这在当默认值是可变的对象像是 `list` 或是 `dictionary` 时会造成重要的差别。举例来说，底下的函数会记录曾经被调用过每次所传入的参数。

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

印出来的结果会是：

```
{1}
[1, 2]
[1, 2, 3]
```

所以如果你的默认值是一个可变的对象，但是你又不想让每次调用都共享的时候，你就必须如此写你的函数：

```
def f (a, l = None) :
    if l is None:
        l = []
```

```
l.append(a)
return l
```

6.8.2 关键词参数

调用函数时也可以使用关键词参数，其形式是“*keyword = value*”，下面的这个函数：

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

用这些方式调用都是正确的：

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

但是用这些方式都是不正确的：

```
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
```

一般来说，一连串的参数次序是先有非关键词参数（也可以没有）然后才是关键词参数，关键词必须是函数定义时所用的参数名称。这个定义时用的参数名称有没有默认值并不重要，但是一个传入的参数只能有一个值（默认值不算），如果你已经先用非关键词参数给了某个参数一个值，接下来你就不能再用关键词参数给它另外的值。下面的例子就违反了规则：

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
```



```
Traceback (innermost last) :  
  File "<stdin>", line 1, in ?  
TypeError: keyword parameter redefined
```

当一个函数定义时参数名称是以 *****name*** 这种形式定义时,表示这个参数要接受的是一个 **dictionary** (译:字典,包含许多关键词以及值的对应),这个 **dictionary** 包含许多的关键词参数,但是这些关键词不能跟其他的参数名称相同。另外参数也可以用 ****name*** 这种形式定义(下一小节会解释),这种方式定义的参数要接受的是一个 **tuple** (译:不可更动的 list),这个 **tuple** 可以接受不限数目的非关键词参数 (****name*** 必须要出现在 *****name*** 之前)。下面的例子就是一个函数定义的范例:

```
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  
    for kw in keywords.keys () : print kw, ': ', keywords[kw]
```

要调用这个函数,你可以这样调用:

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

函数执行的结果如下:

```
-- Do you have any Limburger ?  
-- I'm sorry, we're all out of Limburger  
It's very runny, sir.  
It's really very, VERY runny, sir.  
-----  
client : John Cleese  
shopkeeper : Michael Palin  
sketch : Cheese Shop Sketch
```

6.8.3 随意的参数串

最后，我们要介绍最不常见的形式，也就是定义一个函数可以接受任意数目的参数，这些传入的参数会被放进一个 `tuple` 里面去。在这一个任意数目的参数之前，可以定义没有或是一个或是多个普通的参数：

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

6.8.4 Lambda 形式

由于众多的需求，Python 里面也加入了这一个在其他功能性程序语言及 Lisp 里面常见的特性。你可以使用 `lambda` 这个关键词来定义一些小的没有名字的函数。下面是一个返回两个参数值相加的例子：“`lambda a,b: a+b`”。Lambda 形式可以使用在任何需要函数对象 (function objects) 的地方。语法上限制 lambda 形式只能有一个 `expression`，其功能只是方便的取代一个正常的函数定义。就像是函数里面包含函数定义一样，lambda 形式不能使用 (reference) 外面一层函数的的变量，但是你可以使用传入默认值参数的方式来克服这个问题，像是下面的例子：

```
def make_incrementor (n) :
    return lambda x, incr=n: x+incr
```

6.8.5 批注字符串

批注字符串的内容及形式是有一个新的约定俗成的规范的。

第一行应该是一个有关这个对象的目的的短的、简洁的摘要。因为简洁的缘故，这一行不应该包括对象的名称及类型（除非对象的名称正好是解释对象目的的一个动词），因为对象名称及类型是可以从其他地方得知的。这一行第一个字的第一个字母应该大写，最后应该有一个句点。

如果批注字符串还包含其他行的话，第二行应该是空白的，这样可以使摘要及细部的解释有所区分。底下的各行应该是一个或多个段落，其目的应该是诸如解释对象的调用方法及其副效果 (side effects) 的解释说明。

一般时候，Python 的分析器 (parser) 并不会把多行字符串的缩排拿掉，但是在批注字符串中，批注字符串的处理工具需要特别拿掉那些缩排。底下的一般通用准则可以用来帮助

决定批注字符串如何缩排：在第一行之后所遇到的第一个非空白行决定了整个批注字符串的缩排大小，（我们不能用第一行，因为第一行一定要跟着引号在一起，所以其缩排是不明显的）。在这之后的与这个缩排相等的空白，都会被整个拿掉。如果某行的前面有空白但缩排的空白不足（这是不应该发生的），这些缩排也会被整个拿掉。空白的解释是把 `tab` 展开后（一般为八个空白）的方式来解释的。

这里示范了如何使用多行的批注字符串：

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

第7章 函 数

7.1 定义函数

我们可以定义一个函数，下面这个函数定义的例子中，当我们给定想要印出的范围时，这个函数会印出一个费氏数列来：

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

在上例中这个 `def` 关键词代表了一个函数的定义（function definition），在 `def` 之后必须接着函数的名称以及一个用括号括起来的一连串参数。接下来一行之后的程序代码就是函数的主体部分，而且必须是缩排的。函数的程序代码部分的第一个 `statement` 可以是一个字符串常数（string literal），这个字符串常数会被当作是函数的批注部分而叫做批注字符串（documentation string 或是 docstring）。

这里有工具可以使用这个批注字符串来自动的制作出线上的或是印出来的文件，或者是让用户可以交互式的浏览程序代码。写批注是一个好习惯，所以最好养成这个好习惯，把所有的程序代码都写上批注字符串。

执行函数的时候会产生一个目前（local）的符号表（system table），这个表是用来记录函数中的所有 local 变量的。更准确的来说，所有在函数中变量的设定值都会记录在这个 system table 中，所以当你要使用（reference）一个变量时，会先检查 local 的 system table，

然后是整个程序 (global) 的 system table, 然后是内建的变量名称。虽然 global 变量可以在函数使用 (reference), 但是不能在函数之内直接的设定其值 (除非是在一个 global 的 statement 中建立的)。

当函数被调用时, 实际传入的函数参数是会被记录在被调用函数的 local system table 里的。因此, 参数被传入时是以其值传入的 (call by value)。在此的值指的是对象的参考 (reference), 而非对象本身的值。当一个函数调用另一个函数时, 就会因此调用而建立一个新的 local system table。事实上是, 较恰当的说法是以其对象参考传入的 (call by object reference), 因为如果一个不可改变的对象传入之后, 调用这个函数的地方仍然可以看到这个函数对这个对象的改变 (例如在 list 之中插入一个对象)。

当定义函数的时候, 也就在目前所在的 system table 里定义了这个函数的名称。对直译器来说, 这个函数名称的数据类型是一个用户自定义的函数。这个函数的值名称可以被设定给另一个名称, 然后这个新的名称就可以被当作是一个函数名称来使用。这个过程就是一个一般的重新命名的机制。

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

你也许认为 fib 不是一个函数 (function) 而是一个程序 (procedure)。如同在 C 中一样, 在 Python 的 procedure 指的是没有返回值的函数 (function)。事实上, 就技术上而言, procedure 也是有返回值的, 只是所返回的是一个 Python 系统内键的值, 叫做 None。通常来说, 如果只返回 None 的话, 直译器不会印出这一个返回值。但是, 如果你真想看一看它的话, 你可以这样做:

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

如果想让你的函数返回一个包含费氏数列的 list, 而不是只印出来的话, 其实是很简单的:

```
>>> 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]

```

如同往例，这个例子事实上示范了一些新的 Python 的特点：

1) `return` 语句使得函数返回了一个值。如果单单 `return` 没有其他的 `expression` 来表示返回的值时，就表示这是一个从 `procedure` 返回来的写法（`procedure` 到结束都没有返回值也是表示从 `procedure` 返回来）。这种写法表示返回值是 `None`。

2) `result.append(b)` 这个语句表示调用了 `result` 这个 `list` 对象的一个方法（`method`）。`Method` 是一个特别“属于”某个对象的函数，而且其名称的形式是 `obj.methodname`。在这里 `obj` 指的是某一个对象（我们也可以用 `expression` 来代替），而 `methodname` 指得是由这个对象的数据类型所定义的这个方法的名称。不同的数据类型会定义不同的方法，不同的数据类型也许所定义的方法名称会相同，但是并不会造成冲突（你可以定义你自己的数据类型及其方法，我们称之为类别（`classes`），后面会再谈到的）。在这个例子中的 `append()` 方法是在 `list` 这个数据类型中定义的。这个方法会在 `list` 的最后面加入一个新的成员，在这个例子里也可以写作“`result = result + [b]`”，效果一样，但是用方法来写有效率多了。

7.2 使用参数

在定义函数的时候我们可以加入不定数目的参数，加入参数的写法有三种，是可以混和使用的。

7.2.1 预设内定参数值

最好用的一种写法是，对其中的一个或多个参数给它一个特定的默认值。这样的话，当你在调用函数时，就可以不用传入参数，或是传入较少的参数了。请看下例：

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

```
while 1:
    ok = raw_input (prompt)
    if ok in ('y', 'ye', 'yes') : return 1
    if ok in ('n', 'no', 'nop', 'nope') : return 0
    retries = retries - 1
    if retries < 0: raise IOError, 'refusenik user'
    print complaint
```

当你调用这个函数的时候你可以用 `ask_ok('Do you really want to quit?')`，或者是 `ask_ok('OK to overwrite the file?', 2)`。

设定的默认值可以是一个变量，但是这个变量在函数定义的时候就以定义时的情况（defining scope）决定（evaluate）了其值，所以以下的例子：

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

印出的结果会是 5。

警告： 这个参数默认值只有被 `evaluate` 一次，这在当默认值是可变的对象像是 `list` 或是 `dictionary` 时会造成重要的差别。举例来说，下面的函数会记录曾经被调用过每次所传入的参数。

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

印出来的结果会是：

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

所以如果你的默认值是一个可变的对象，但是你又不想让每次调用都共享的时候，你就必须如此写你的函数：

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

7.2.2 关键词参数

调用函数时也可以使用关键词参数，其形式是 “*keyword* = *value*”，下面的这个函数：

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

用这些方式调用都是正确的：

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

但是用这些方式都是不正确的：

```
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
```

一般来说，一连串的参数次序是先有非关键词参数（也可以没有）然后才是关键词参数，关键词必须是函数定义时所用的参数名称。这个定义时用的参数名称有没有默认值并不重要，但是一个传入的参数只能有一个值（默认值不算），如果你已经先用非关键词参数给了某个参数一个值，接下来你就不能再用关键词参数给它另外的值。下面的例子就违反了这

个规则:

```
>>> def function (a) :  
...     pass  
...  
>>> function (0, a=0)  
Traceback (innermost last) :  
  File "<stdin>", line 1, in ?  
TypeError: keyword parameter redefined
```

当一个函数定义时参数名称是以 ****name** 这种形式定义时,表示这个参数要接受的是一个 **dictionary** (译:字典,包含许多关键词以及值的对应),这个 **dictionary** 包含许多的关键词参数,但是这些关键词不能跟其他的参数名称相同。另外参数也可以用 ***name** 这种形式定义(下一小节会解释),这种方式定义的参数要接受的是一个 **tuple** (译:不可更动的 list),这个 **tuple** 可以接受不限数目的非关键词参数 (***name** 必须要出现在 ****name** 之前)。下面的例子就是一个函数定义的范例:

```
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  
    for kw in keywords.keys () : print kw, ': ', keywords[kw]
```

要调用这个函数,你可以这样调用:

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

函数执行的结果如下:

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

```

It's very runny, sir.
It's really very, VERY runny, sir.
-----
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch

```

7.2.3 随意的参数串

最后，我们要介绍最不常见的形式，也就是定义一个函数可以接受任意数目的参数，这些传入的参数会被放进一个 `tuple` 里面去。在这一个任意数目的参数之前，可以定义没有或是一个或是多个普通的参数：

```

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

```

7.2.4 Lambda 形式

由于众多的需求，Python 里面也加入了这一个在其他功能性程序语言及 Lisp 里面常见的特性。你可以使用 `lambda` 这个关键词来定义一些小的没有名字的函数。下面是一个返回两个参数值相加结果的例子：“`lambda a, b: a+b`”。Lambda 形式可以使用在任何需要函数对象（function objects）的地方。语法上限制 lambda 形式只能有一个表达式，其功能只是方便的取代一个正常的函数定义。就像是函数里面包含函数定义一样，lambda 形式不能使用（reference）外面一层函数的的变量，但是你可以使用传入默认值参数的方式来克服这个问题，像是下面的例子：

```

def make_incrementor (n) :
    return lambda x, incr=n: x+incr

```

7.2.5 批注字符串

批注字符串的内容及形式是有一个新的约定俗成的规范的。

第一行应该是一个有关这个对象的目的的短的、简洁的摘要。因为简洁的缘故，这一行不应该包括对象的名称及类型（除非对象的名称正好是解释对象目的的一个动词），因为对象名称及类型是可以从其他地方得知的。这一行第一个字的第一个字母应该大写，最后应

该有一个句点。

如果批注字符串还包含其他行的话，第二行应该是空白的，这样可以让摘要及细部的解释有所区分。下面的各行应该是一个或多个段落，其目的应该是诸如解释对象的调用方法及其副效果（side effects）的解释说明。

一般时候，Python 的分析器（parser）并不会把多行字符串的缩排拿掉，但是在批注字符串中，批注字符串的处理工具需要特别拿掉那些缩排。下面的一般通用准则可以用来帮助决定批注字符串如何缩排：在第一行之后所遇到的第一个非空白行决定了整个批注字符串的缩排大小，（我们不能用第一行，因为第一行一定要跟着引号在一起，所以其缩排是不明显的）。在这之后的与这个缩排相等的空白，都会被整个拿掉。如果某行的前面有空白但缩排的空白不足（这是不应该发生的），这些缩排也会被整个拿掉。空白的解释是把 tab 展开后（一般为八个空白）的方式来解释的。

这里示范了如何使用多行的批注字符串：

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

第 8 章 类与对象

8.1 Class (类)

Python 的类别机制在引入最少新的语法及语意的情况下加入了类别的支持。Python 的类别机制是 C++ 以及 Modula-3 的综合体。正如同在 modules 里面的情况一样，Python 的 class 也没有在其定义及用户之间加入绝对的障碍，而是仰赖用户有礼貌的不要去闯入其定义之中（not to “break into the definition”）。对于 class 来说最重要的一些特性在 Python 里面都完全保留：类别的继承可以继承自各基础类别（base classes），一个子类别（derived class）可以不理睬其所有基础类别（base class）的任何方法（method），一个 method 也可以调用一个基础类别的同名方法，对象可以自由决定是否要让某些资料是私有的。

以 C++ 的术语来说，Python 所有的类别成员（包含其资料成员）都是 public 的，而且所有的成员函数（member functions）都是 virtual 的。也并没有所谓的构造器（constructors）或是解构器（destructors）的存在。如同在 Modula-3 里面一样，从对象的方法（method）里面要使用对象的成员并没有快捷方式可以使用：成员函数的声明必须在第一个参数中明白的表示存在其中的对象，而此参数在调用时是不用传的。如同在 Smalltalk 里面一样，类别本身也是一个对象，事实上在 Python 里面，所有的数据类型（data type）都是对象。这提供了在 import 以及重新命名时候的语意（semantics）。但是如同在 C++ 或是 Modula-3 里面，内建的基本类型是不能被用户拿来当作基础类别使用的。与 C++ 类似但不同于 Modula-3 的是，大部分有特别语法的内建操作数（operators），例如数值运算及 subscripting，都可以被拿来在类别中重新定义的。

8.2 术语的使用说明

由于缺乏普遍性的术语可以讨论类别，我只好偶而从 Smalltalk 或是 C++ 的术语中借来用。（我其实更想用 Modula-3 的术语，因为它的术语在语意上比 C++ 还要接近 Python，但是我想大部分的读者都没有听过它）。

我也要警告你的是，对象这个字在 Python 里面不必然指的是类别的一个特例（instance），

这是一个在面向对象读者中常见的陷阱。与 C++ 及 Modula-3 相同但与 Smalltalk 不同的是，并非所有在 Python 里面的数据类型都是类别，像是整数及 list 这类的基本内建类型就不是类别，甚至一些特别的数据类型像是 file 都不是类别。无论如何，所有的 Python 的数据类型都或多或少有一些基本相同的语意特性，我们可以把这个相同点叫做对象。

对象有其个体性 (individuality, 独特性)，而且你可以用不同的名字连接到同一个对象去，这在其他的程序语言中也叫做别名 (aliasing)。通常你第一次看到 Python 不会觉得这有什么特别，而且你在处理不可变动的 (immutable) 基本类型 (例如数目字，字符串及 tuple) 时，根本可以不去管它。但是对于一些都可变动的 (mutable) 对象，像是 list, dictionary 以及其他用来表现在程序之外的实体 (像是文件及窗口) 的数据类型，对它们来说 aliasing 就与它们有关的 Python 程序代码语意的解释，有 (故意的) 一些影响。这样的影响通常对程序有正面的效益，因为别名 (alias) 运作的方式就像是一个有礼貌的指针 (pointer)。举例来说，当你传一个对象当参数时，因为所传的其实只是一个指针，所以所费的资源就不多。而且，当在函数之内对这个传入的对象进行修改时，在外面调用这个函数的人 (caller) 会看得见函数所做的修改，这大大地简化了在 Pascal 里面需要两种不同参数传递机制才能作到的事。

8.3 Python 的可用范围 (Scopes) 及命名空间 (Naming Spaces)

在介绍类别 (class) 之前，我首先必须介绍 Python 有关可用范围 (scope) 的一些准则。类别的定义也对命名空间 (namespace) 做了一些小技巧，所以你需要对 scope 及 namespace 的运作有一些了解才能够完全掌握到底发生了什么事。对于高级的 Python 程序设计师来说，有关这个主题的了解是很有帮助的。

现在让我们先来定义一些东西：

一个 namespace 指的是名称与对象的对应关系的组合。目前来说，namespace 都是用 Python 的 dictionary 实作出来的，但是这应该没有多大意义 (除非对程序的效率)，而且在未来可能也有所改变。Namespace 的例子有：一组的内建名称 (像是 abs() 的函数，还有内建的 exception 名称)，在 module 里的全局变量 (global variables)，以及在函数里的 local 变量。从某种意义上来说，一个对象里的特性 (attributes, 译为成员) 也组成一个 namespace。在这里要知道的重点是，不同的 namespace 里面所定义的名称是彼此没有任何关系的。举例来说，两个不同的 module 都可以定义一个叫做 “maximize” 的函数。这并不冲突，因为用户必须要在该函数的名称前加上 module 的名称。

我在这里所用的 attributes 一字指的是所有在点号后面的东西，举例来说在 z.real 这个

expression 里面 `real` 就是一个属于 `z` 对象的 `attributes`。严格说来,使用 `module` 里面的名称也是一个 `attributes` 的指称 (references),在 `modname.funcname` 这个 expression 里面,`modname` 就是一个 `module` 对象,而 `funcname` 就是其 `attributes`。此例子里,刚好 `module` 的 `attributes` 对应了在 `module` 里面定义的全局变量,所以我们说它们就是在一个 `namespace` 里面。

`Attributes` 可以是只读的或是可写的。对可写的 `attributes`,你可以给它设定值。`Module` 的 `attributes` 是可写的:所以你可以写 “`modname.the_answer = 42`” 来改变它的值。可改写的 `attributes` 也可以被删除掉,你可以用 `del` 语句像是 “`del modname.the_answer`” 来做。

命名空间 (Name spaces) 是在不同的时候被创造出来的,而且其存在的时间也都不一定。内建名称的 `namespace` 是在当 Python 直译器启动时就被创造出来,而且不会被删除掉。`Module` 里全局 (global) 的 `namespace` 是在 `module` 的定义被读入的时候创造出来,通常在直译器离开之前也不会被删除。那些在 top-level 启动直译器里面被执行的指令,不管是从互动模式或是 script 里来的,都隶属于一个叫做 `_main_` 的 `module`,所以它们也算有自己的一个 `global namespace`。(事实上,内建的名称也都在一个 `module` 里面,这个 `module` 叫做 `_builtin_`)。

函数所有的 `namespace` 叫做 `local namespace`,是在函数被调用时才创造的,而且当函数返回一个值或是引发一个本身无法处理的 `exception` 时,这个 `namespace` 就被删除 (事实上,也许说遗忘是比较贴切的形容词)。当然,递归的函数调用会使每个调用都有自己的 `local namespace`。

可用范围 (scope) 是一个在 Python 程序里面文字上的范围,在这个范围里你可以直接使用某个 `namespace`。直接使用 (“Directly accessible”) 的意思是指对一个名称而言不合格的参考 (unqualified reference) 试图想要在 `namespace` 里面找某一个名称。

虽然 `scope` 是静态 (statically) 被决定的,但是我们使用 `namespace` 的时候是动态 (dynamically) 的。在任何一个程序执行的地方,都有三层 `scope` 正在被使用 (也就是有三个可以直接使用的 `namespace`): 首先寻找的是最内圈的 `scope`,包含有 `local` 的名称; 其次搜寻的是中间一层,包含目前所在的 `module` 的全局名称 (global names); 最后搜寻的是最外面的一层,也就是包含有内建名称的 `namespace`。

通常, `local scope` 指的是在文字上面目前函数所拥有的 `local` 名称。如果在函数之外, `local scope` 就指的是 `global scope` 所指的 `namespace`。类别的定义在 `local scope` 里面又放入了另外的一个 `namespace`。

要注意的是 `scope` 的决定是依文字的安排来决定的。一个定义在 `module` 里面的函数,其 `global scope` 就是 `module` 的 `namespace`,不管这个函数是从哪里或是用哪一个别名被调用的。

在另一方面来说，真正的名称搜寻路线是动态决定的（在程序执行的时候）。但是 Python 语言本身的定义好像慢慢的转向静态决定变化（也就是在编译的时候），所以，不要过分依赖动态的名称解释。（事实上，local 的变量都是静态就已经决定了的）。

Python 有一个很特别的变化就是当设定（assignment）的时候一定是进入到了最内层的 scope。设定并不是复制资料，相反的，它只是把对象及名称连接起来而已。对于删除也是一样的，“del x”事实上只是把 x 的连接从 local scope 所代表的 namespace 中除去。事实上，所有会引进新名称的动作都是使用 local scope，特别是 import 语句以及函数的定义就是把 module 以及函数的名称都连接到 local scope 里面来了。（global 这个语句可以用来特别指定某个特殊的变量要放在 global scope 里）

8.4 Class（类别）初探

类别（Classes）的观念引进了许多新的语法，以及三种新的对象以及一些新语言上的意义。

8.4.1 定义 Class（类别）的语法

最简单的类别定义的形式看起来是这样的：

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

类别的定义与函数的定义（都是用 def 语句）相同，都必须要在它们有任何作用之前定义好。（你也可以在 if 语句或是一个函数里面放入类别的定义）。

在实务上，存在于类别定义内的语句通常都是函数的定义，但是我们也可以放入其他的语句。这种做法有时也很好用，我们以及会再看这个用法。类别定义内的函数定义通常有一个特别的参数形式，这是为了 method 的特别调用习惯。我们还是留到后面再来讨论它。

当一个类别的定义进来时，会创造出一个新的 namespace，而且会当作一个 local scope 来用。所以所有对 local 变量的设定都会进入到这个新的 namespace 里。具体来说，函数的定义也会把新的函数的名称连接到这里来。

当一个类别的定义正常离开时（通过定义的尾端），一个类别对象（class object）就被创造出来。这个类别对象基本上来说只是一个包装起来的東西，其内容是由这个类别定义所创造出来的 namespace 里面的内容。我们在下一节就会有更多有关类别对象（class objects）的讨论。另外在类别的定义离开时，原来的 local scope（在进入类别定义之前的那一个 local space）就会被重新使用，并且这个创造出来的类别对象会被放在这个 local scope 里，并且被连接到你所定义的类别名称（上面的例子里是 ClassName）上。

8.4.2 类别对象（Class Objects）

类别对象可以做两件事情，一是 attribute 的指涉（references），另一个是创造出一个特例来（instantiation）。

Attribute references 使用的是在 Python 里面标准的 attribute reference 语法：obj.name。有效的 attributes 的名称指的是当类别对象被创造时，所有在类别的 namespace 里的名称。所以，如果你的类别定义如下面例子的话：

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

你就可以使用 MyClass.i 以及 MyClass.f 这两个有效的 attribute references 语法，它们分别会返回一个整数以及一个 method 对象来。你也可以设定值给这些类别的 attributes，如此就可以改变 MyClass.i 的值了。_doc_也是类别对象的一个有效的 attributes，其返回值是这个类别的注释字符串（docstring），也就是：“A simple example class”。

类别的特例化（Class instantiation）是使用函数的表示方法。看起来好像这个类别对象是一个没有参数的函数，然后返回来的就是这个类别的一个特例（instance）。我们前面的类别为例子：

```
x = MyClass()
```

这样创造出一个新的类别的 instance，然后我们再把这个对象设定给 x 这个 local 的变量。

类别的特例化（Class instantiation）动作（也就是“调用”一个类别对象）所创造出来的

是一个空对象。有许多类别希望创造出来的对象有一个特定的初始状态，所以你可以在类别里面定义一个特别的 method 叫做 `_init_()`，如同下例：

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

当你的类别定义一个 `_init_()` method 时，那么在特例化 (instantiation) 你的类别时，就会自动的引发 `_init_()` 执行，并且创建一个类别的特例 (instance)。所以，一个新的对象就可以通过下面的调用来创造出来：

```
x = MyClass ()
```

当然，`_init_()` 这个 method 可以有参数传入，这样能增加使用时的弹性。这样做，使用特例化 (instantiate) 类别的语法时，所传入的参数就会被传到 `_init_()` 里去。如范例：

```
>>> 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)
```

8.4.3 特例对象 (instance objects)

现在对于这个被创造出来的特例对象 (instance objects)，我们又该怎么用呢？对于这样的特例对象，它们唯一懂得的就是 attribute references。有两种的 attribute names 可以使用：

第一种我叫他是数据特性 (data attributes)，这类似于 Smalltalk 中所说的特例变量 (“instance variables”) 以及在 C++ 中的数据成员 (“data members”)。如同 local 变量一样，Data attributes 不需要再声明，你第一次设定给它们值的时候就自动存在了。举例来说，如果 `x` 是 `MyClass` 这个对象的一个 instance，下面这个程序代码就会印出 16 这个结果来：

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

```
print x.counter  
del x.counter
```

第二种 instance object 可以使用的 attribute references 叫做方法 (methods)。一个 method 就是一个隶属于某个对象的函数。(在 Python 中, method 一词并不只特定用于类别的 instances, 其他的对象数据类型也可以有自己的 method, 例如 list 对象就有很多 methods 像是 append, insert, remove, sort 等等。但是我们下面用到 method 这个词的时候, 除非特别说明, 要不然我们倒是单独指着 instance objects 的 method 说的。)

一个 instance objects 可以用的有效 method 名称是由其类别决定的。从定义上来说, 所有类别里面 (用户定义) 为函数对象的 attributes, 都会成为其 instance 的相对应 method。所以在我们的例子里, x.f 就是一个有效的 method 的 reference, 其原因是 MyClass.f 为一个函数; 但是 x.i 就不是一个 method 的 reference, 因为 MyClass.i 不是一个函数。可是, 要注意的是, x.f 和 MyClass.f 是两回事, 它是一个 method 对象 (method object), 而非一个函数对象。

8.4.4 Method Objects (方法对象)

通常, 一个 method 可以马上被调用, 例如:

```
x.f ()
```

在上面的例子里, 这个调用会返回来“hello world”字符串。但是, 因为 x.f 是一个 method 对象, 所以我们没有必要马上就调用它, 我们可以把它储存起来, 稍后再调用它。举例如下:

```
xf = x.f  
while 1:  
    print xf ()
```

这个例子同样会不断的印出“hello world”来。

你的 method 被调用时, 到底什么事情发生了呢? 你也许注意到了当我们调用 x.f() 时并没有传入任何参数, 但是我们在类别定义的时候确实定义了 f 所传入的参数。这是怎么回事呢? 当然, 依照 Python 的定义, 当一个函数需要参数而你调用时没有传入参数的话, 是会引发一个例外状况 (exception) 的, 甚至这个传入的参数没有被用到也是一样的。

事实上, 你也许已经猜到答案了。Method 的特殊点是, method 所处的对象会被当作函数传入的第一个参数。所以在我们的例子里, 当调用 x.f() 的时候, 我们事实上是调用

`MyClass.f(x)`。一般来说，如果你调用 `method` 时传了 `n` 个参数，其实你是调用背后所代表的类别的函数，而且该 `method` 所在的对象会插入在传入的参数中作为第一个参数。

如果你还不了解到底 `method` 如何运作，你也可以看看它的实作。当一个 `instance` 的 `attributes` 被 `reference`，而这个 `attributes` 又不是一个 `data attributes` 的时候，该 `instance` 的类别会被寻找。如果这个 `class attributes` 的名字在类别里面代表的是一个函数对象的话，就会有一个 `method` 对象被创造出来。这个 `method` 对象是一个由这个 `instance` 对象（指针），以及刚刚找到的这个函数对象所包装起来的一个抽象对象。当这个 `method` 对象被带着一串参数调用的时候，这个 `method` 对象会先打开原来的包装，然后用 `instance` 对象（指针）以及那一串传进来的参数组成新的参数串，然后我们再用这个新的参数串来调用在 `method` 对象里的函数对象。

8.5 一些随意的想法

这些东西其实应该多花点心思加以处理的。

如果 `data attributes` 和 `method attributes` 有相同名称的话，`data attributes` 会覆盖 `method attributes`。要避免这个命名的冲突（这常常是许多 `bug` 的由来），你可能需要一些命名的规则。比如说，让 `method` 的名称都是大写的，在 `data attributes` 的前面加上一些小字符串（或下划线），或者对于 `method` 都用动词，对 `data attributes` 都用名词。

除了一般 `object` 的用户（`client`）之外，`Data attributes` 也可以在 `method` 里面被使用到。也就是说，类别（`class`）是不能用来实作出纯粹的抽象数据类型（`abstract data types`）的。事实上，在 `Python` 里没有东西可以保证数据的隐藏（`data hiding`），我们只能仰赖彼此的约定及尊重了。（另一方面来说，用 `C` 写成的 `Python` 是可能完全隐藏其实作的细节，并且在需要的时候控制对对象的存取权限的：这是用来给 `C` 所写成的 `Python` 延伸机制（`extension to Python`）使用的。）

使用 `data attributes` 的人要特别小心，你有可能把由 `method` 管理的 `data attributes` 弄得一塌糊涂。值得注意的是，类别的用户可以自行在 `instance` 对象里加入 `data attributes`，只要小心处理命名的问题，这不会对 `method` 的正确性有所影响。再次提醒，你可以用命名的规则来避免此事发生。

从 `method` 里使用 `data attributes`（或者是其他的 `methods`）并没有快捷方式。我发现这样的好处是程序的可读性增加很多，因为当你在读 `method` 程序代码的时候，`local` 变量和 `instance` 变量混淆的机会就会少许多。

习惯上，我们把一个 **method** 的第一个参数叫做 **self**。这只是一个习惯而已，**self** 这个名字对 Python 来说没有什么特殊的意义。（但是你要注意，如果你不用这个习惯的话，对于某些读你程序的 Python 程序设计师来说，也许可读性就低了一点。而且可能有一些类似 **class browser** 之类的程序是靠这个约定来分辨 **class** 的特性，所以你不遵守的话，你的类别可能就读不懂）所有在类别里的函数对象，在定义上都是该类别之 **instance** 的一个 **method**。在类别里的意思不限定于一定要在文字上是在类别的定义里，你也可以把一个函数对象设定给一个在类别里的 **local** 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** 以及 **h** 都是类别 **C** **attributes**，而且都指涉（reference）到某个函数对象去。同时，当 **C** 有 **instance** 的时候，**f**, **g** 以及 **h** 都会变成 **instance** 的 **method**（事实上 **h** 所指的函数跟 **g** 是同一个的）。值得注意的是，如果你真这样做的话，只是令读你程序的人头昏眼花罢了。

你也可以在 **method** 里调用其他的 **method**，只是用 **self** 这个参数的 **method attribute** 就可以了。例如：

```
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)
```

method 跟一般的函数对象一样可以使用全局名称（global name）。Method 的 **global scope**

指的是类别定义所存在的 module, (注意: 类别本身绝不会是一个 global scope)。你大概很少有机会在 method 里需要用到 global scope, 但是还是可以使用 global scope 的, method 可以使用在 global scope 中所 import 进来的函数以及 module, 也可以使用在 global scope 里面定义的函数及类别。通常, 包含 method 的这个类别本身就定义在 global space 里, 而且下一段我们就要讲到为什么你会需要在 method 里用到本身的类别。

8.6 继承 (Inheritance)

当然啦, 一个程序语言如果没有继承的话就不需要担心类别 (class) 这个字了。一个子类别 (derived class) 的定义看起来是这样的:

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

其中, 基础类别的名字 BaseClassName 这个字必须是在子类别所处的 scope 里有定义的。除了直接使用基础类别的名字之外, 你也可以使用一个 expression。这在基础类别是定义在别的 module 里的时候特别有用:

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

子类别定义的执行过程与基础类别定义的执行过程是一样的。当一个类别对象被创造出来时, 基础类别也同样会存在内存中。这是为了确保能够找到正确的 attributes 的所在, 如果你的子类别没有定义某个 attributes 的话, 就会自动去找基础类别的定义。如果这个基础类别也是某个类别的子类别的话, 这个法则是一直延伸上去的。

子类别的特例化 (instantiation) 也没有什么特别之处, 使用 DerivedClassName () 就会创造出子类别的一个新 instance。子类别的 method 则是由以下的过程来寻找: 先找该类别的 attributes, 然后如果需要的话会沿着继承的路线去找基础类别, 如果找到任何函数对象的话, 这个 method 的参考 (reference) 就是有效的。

子类别可以 override 基础类别里的 method。因为 method 在调用自己对象的其他 method 的时候没有特别的权限, 当一个基础类别的 method 调用原属于该基础类别的 method 的时候,

有可能真正调用到的是一个在子类别里定义的 override 的 method。(给 C++ 的程序设计师们: 所有在 Python 里面的 method 都是 virtual 的。)

一个在子类别里 override 的 method 也许会需要延伸而非取代基础类别里同名的 method, 这时候你就需要调用基础类别里的 method: 只要调用 “BaseClassName.methodname (self, arguments)” 就可以了。这对于类别的用户来说, 有时也是有用的。(需要注意的是, 如果这样做, 你要将基础类别定义在 global scope 或是 import 到 global scope 里。)

多重继承

Python 也支持部分的多重继承形式。一个类别如果要继承多个基础类别的话, 其形式如下:

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

唯一需要解释的规则是, 当你寻找一个 attributes 的定义时要如何寻找。其规则是先深, 而后由左至右 (depth-first, left-to-right)。所以当你要找一个在子类别 DerivedClassName 里的 attributes 却找不到时, 会先找 Base1, 然后沿着 Base1 的所有基础类别寻找, 如果还没有找到的话再找 Base2 及其基础类别, 依此类推。

也许有些人认为由左至右然后在深才对, 应该是先找 Base2 及 Base3, 然后才找 Base1 的基础类别。如果这样的话, 你可以再想一想, 当找 Base1 的时候, 你需要先知道这个 attributes 到底是定义在 Base1 本身还是其基础类别里, 如此才不会与 Base2 里的 attributes 有同名的困扰。如果你使用先深, 而后由左至右的规则的话, 就不会有这个困扰。

大家都知道如果不小心使用的话, 多重继承可能变成在维护程序时的一个恶梦。Python 仰赖程序设计师们约定俗成的习惯来避免可能的名称冲突。例如一个众所周知多重继承的问题, 如果一个类别继承了两个基础类别, 这两个基础类别又分别继承了同一个基础类别。也许你会很容易了解在这样的情况下到底会是什么状况, (这个 instance 将会只有一个单一共享基础类别的 instance variables 或是 data attributes), 但是很难了解这到底有什么用处。

8.7 Private变量

在 Python 里面只有有限度的支持类别中的 private 指称 (class-private identifiers, 译: 指变量及函数)。任何的 identifier, 在之前是以 `_spam` 形式存在的 (前面至少要有两个下划线, 最后面最多只能有一个下划线) 现在都要以 `_classname_spam` 这个形式来取代之。这里 `classname` 指的是所在的类别名称, 拿掉所有前面的下划线。这个名称的变化不受限于这个 identifier 语法上所在的位置, 所以可以套用在定义类别的 private instance、类别变量、method, global 名称, 甚至用来储存其他的类别 instance 里, 对目前这个类别来说是 private 的 instance 变量。当这个变化过的名称超过 255 个字符时, 有可能超过的部分是会被截掉的。在类别之外, 或者是当类别的名称只包含下划线的时候, 就没有任何名称的变化产生了。

这个名称的变化主要是用来给类别一个简单的方法来定义 private 的 instance 变量及 methods, 而不需要担心其他子类别里定义的 instance 变量, 或者与其他的在类别之外的程序代码里的 instance 变量相混淆。这个变化名称的规则主要是用来避免意外的, 如果你存心要使用或修改一个 private 变量的话, 还是可行的。从某方面来说这也是有用的, 比如用在除错器 (debugger) 上, 这也是为什么此漏洞没有被补起来的一个原因。(如何制造 bug, 如果一个类别继承自某个基础类别时用了相同的名字, 这会使能从子类别里使用基础类别里的 private 变量。)

值得注意的是, 被传到 `exec`, `eval()` 或 `evalfile()` 的程序代码不用考虑引发这个动作的类别是目前的类别, 这类似于 global 语句的效果, 但是这个效果只限于此程序代码是一起被编译器编译成 bytecode 的时候。同样的限制也存在于 `getattr()`, `setattr()` 以及 `delattr()`, 或是直接使用 `_dict_` 的时候。

下面这个例子是一个类别里面定义自己的 `_getattr_()` 以及 `_setattr_()` 的两个方法, 并且把所有的 attributes 都储存在 private 的变量里面。这个例子适用于所有的 Python 版本, 甚至包括在这个特性加入之前的版本都可以:

```
class VirtualAttributes:
    _vdict = None
    _vdict_name = locals().keys()[0]

    def _init_(self):
        self._dict_[self._vdict_name] = {}
```

```
def _getattr_ (self, name) :  
    return self._vdict[name]  
  
def _setattr_ (self, name, value) :  
    self._vdict[name] = value
```

8.8 其他

有时候如果有一个像是 Pascal 的 `record`，或者是 C 的 `struct` 这类的数据类型是很方便的，这类的数据类型可以把一些的数据成员都放在一起。这种数据类型可以用空白的类别来实作出来，例如：

```
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
```

如果一段的 Python 程序代码需要一个特别的抽象数据类型的时候，通常你可以传给这段程序代码一个类似功能的类别来代替。例如，你有一个函数是用来格式化一些来自于 `file` 对象的数据，就可以定义一个类别，类别里面有类似 `read()` 以及 `readline()` 之类 `method` 可以从一个字符串缓冲区（string buffer）读出数据，然后再把这个类别传入函数当作参数。

Instance 的 `method` 对象也可以有 `attributes`，`m.im_self` 就是其 `method` 为 `instance` 的一个对象，`m.im_func` 是这个 `method` 相对应的函数对象。

例外（Exceptions）也可以是类别

用户自定义的 `exception` 不用被限定于只是字符串对象而已，它们也可以用类别来定义了。使用这个机制，就可以创造出一个可延伸的 `exception` 的阶层了。

有两个新的有效的（语意上的）形式现在可以用来当作引发 `exception` 的语句：


```
raise Class, instance
raise instance
```

在第一个形式里面，`instance` 必须是 `Class` 这个类别或其子类别的一个 `instance`。第二种形式其实是下面这种形式的一个简化：

```
raise instance._class_, instance
```

所以现在在 `except` 的语句里使用字符串对象或是类别都可以了。`exception` 子句里的类别可以接受一个该类别的 `exception`，或者是该类别之子类别的 `exception`。（相反就不可以了，一个 `except` 子句里如果用的是子类别，就不能接受一个基础类别的 `exception`。）例如，下面的程序代码就会依序的印出 `B`，`C`，`D` 来：

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

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

值得注意的是，如果上面例子里的 `except` 子句次序掉转的话（“`except B`”是第一个），这样印出来的就是 `B`，`B`，`B`，可见只有第一个可以接受的 `except` 子句被执行了。

当一个没有被处理到的 `exception` 是一个类别时，印出来的错误信息会包含其类别的名称，然后是 `(:)`，再是这个 `instance` 用内建的 `str()` 函数转换成的字符串。

第9章 Python 语言调试

通常，程序员爱上 Python 是因为它能提高生产力。由于没有编译过程，编辑—测试—调试周期相当快。调试 Python 程序很简单：一个错误永远不会导致一个段错误。当解释器发现错误时，它就引发一个异常。当程序没有捕捉到异常时，解释器就打印一个堆栈跟踪。一个源码级调试器允许我们检查局部和全局变量、计算表达式、设置断点、单步跟踪等等。调试器是用 Python 写的，这证明了 Python 的能力。另外，最快的调试程序的方法是增加几条打印语句，快捷的编辑—测试—调试周期使得这个简单的办法十分有效。

Python 还提供错误信息，至少有两种不同错误：句法错和例外错（exceptions）。

9.1 句法错

句法错也称为语法分析错，是你在学习 Python 的时候最可能犯的错误。

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

语法分析器重复出错行，并用一个小“箭头”指向行内最早发现错误的位置。错误是由箭头前面的记号引起的（至少是在这里检测到的）。在本例中，错误在关键字 `print` 处检测到，因为它前面应该有一个冒号（:）。错误信息中显示了文件名和行号。这样的话当错误发生在一个脚本文件中时你就知道到哪里去找。

9.2 例外

即使语句或表达式句法没有问题，在试图运行的时候也可能发生错误。运行时检测到的错误叫做例外，这种错误不一定是致命的。你很快就会学到如何在 Python 程序中处理例外。然而，多数例外不能被程序处理，而只是会产生错误信息，如：

```
>>> 10 * (1/0)
Traceback (innermost last) :
  File "<stdin>", line 1
ZeroDivisionError: integer division or modulo
>>> 4 + spam*3
Traceback (innermost last) :
  File "<stdin>", line 1
NameError: spam
>>> '2' + 2
Traceback (innermost last) :
  File "<stdin>", line 1
TypeError: illegal argument type for built-in operation
```

错误信息的最后一行显示发生的情况。例外有不同的类型，类型作为错误信息的一部分显示：上例中错误的类型有 `ZeroDivisionError`、`NameError` 和 `TypeError`。作为例外类型显示的字符串是发生的例外的内置名。这对于所有内置例外成立，但对用户自定义例外不一定成立（用户最好能遵守这样的约定）。标准例外名是内置的标识符（不是保留关键字）。

此行的其余部分是错误的细节，其解释依赖于例外类型。错误信息前面的部分以堆栈反跟踪的形式显示了发生错误的上下文环境。一般这包含了列出源代码行的一个列出源程序行的堆栈反跟踪；然而，它不会显示从标准输入读进的行。

9.3 例外处理

可以编程序来处理选定的例外。请看下面的例子，显示一些浮点数的倒数：

```
>>> numbers = [0.3333, 2.5, 0, 10]
>>> for x in numbers:
...     print x,
...     try:
...         print 1.0 / x
...     except ZeroDivisionError:
...         print '*** has no inverse ***'
... 
```

```

0.3333 3.00030003
2.5 0.4
0 *** has no inverse ***
10 0.1

```

try 语句是这样工作的：

- 1) 首先，运行 try 子句（在 try 和 except 之间的语句）。
- 2) 如果没有发生例外，跳过 except 子句，try 语句运行完毕。
- 3) 如果在 try 子句中发生了例外错误而且例外错误匹配 except 后指定的例外名，则跳过 try 子句剩下的部分，执行 except 子句，然后继续执行 try 语句后面的程序。
- 4) 如果在 try 子句中发生了例外错误但是例外错误不匹配 except 后指定的例外名，则此例外被传给外层的 try 语句。如果没有找到匹配的处理程序则此例外称作是未处理例外，程序停止运行，显示错误信息。

try 语句可以有多个 except 子句，为不同的例外指定不同处理。至多只执行一个错误处理程序。错误处理程序只处理相应的 try 子句中发生的例外，如果同 try 语句中其他的错误处理程序中发生例外，错误处理程序不会反应。一个 except 子句可以列出多个例外，写在括号里用逗号分开，例如：

```

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

```

最后一个 except 子句可以省略例外名，作为一个通配项。这种方法要谨慎使用，因为这可能会导致程序实际已出错却发现不了。

try ... except 语句有一个可选的 else 子句，如有的话要放在所有 except 子句之后。else 的意思是没有发生例外，我们可以把 try 子句中没有发生例外时要做的事情放在这个子句里。例如：

```

for arg in sys.argv[1:]:
    try:
        f = open (arg, 'r')
    except IOError:
        print '不能打开', arg
    else:

```

```
print arg, '有', len (f.readlines ()), '行'
f.close ()
```

例外发生时可能伴有一个值，叫做例外的参数。参数是否存在及其类型依赖于例外的类型。对于有参数的例外，**except** 在自居可以在例外名（或表）后指定一个变量用来接受例外的参数值，如：

```
>>> try:
...     spam {}
... except NameError, x:
...     print 'name', x, 'undefined'
...
name spam undefined
```

有参数的例外未处理时会在错误信息的最后细节部分列出其参数值。

例外处理程序不仅处理直接产生于 **try** 子句中的例外，也可以处理 **try** 子句中调用的函数（甚至是间接调用的函数）中的例外。如：

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

9.4 产生例外

raise 语句允许程序员强行产生指定的例外。例如：

```
>>> raise NameError, 'HiThere'
Traceback (innermost last) :
  File "<stdin>", line 1
```

```
NameError: HiThere
```

`raise` 语句的第一个参数指定要产生的例外的名字。可选的第二参数指定例外的参数。

9.5 用户自定义例外

程序中可以定义自己的例外，只要把一个字符串赋给一个变量即可。例如：

```
>>> my_exc = 'my_exc'
>>> try:
...     raise my_exc, 2*2
... except my_exc, val:
...     print 'My exception occurred, value:', val
...
My exception occurred value: 4
>>> raise my_exc, 1
Traceback (innermost last):
  File "<stdin>", line 1
my_exc: 1
```

许多标准模块用这种方法报告自己定义的函数中发生的错误。

9.6 定义清理动作

`try` 语句还有另一个 `finally` 可选子句，可以用来规定不论出错与否都要执行的动作。例如：

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

`finally` 子句不论 `try` 子句中是否发生例外都会执行。例外发生时，先执行 `finally` 子句然后重新提出该例外。当 `try` 语句用 `break` 或 `return` 语句退出时也将执行 `finally` 子句。

要注意的是，`try` 语句有了 `except` 子句就不能有 `finally` 子句，有了 `finally` 子句就不能有 `except` 子句，不能同时使用 `except` 子句和 `finally` 子句。需要的话可以嵌套。

第 10 章 Python 的杀手程序 Zope

10.1 Zope 简介

Zope 是一个完整的网站管理系统，它包括了所有有关创造及管理一个网站所需的功能。Zope 所有的功能都可以通过 web 接口来使用，你可通过 web 接口加入一个新的网页，设定它的权限，改变网页的内容。所有的网页都可以由 Zope 提供的叫做 DTML 的语言动态产生。同时 Zope 还是一个很完整的网站软件，它还提供了以下内容：

1. 网页服务器。

Zope 的网页服务器是一个高效能的 multi-threads 服务器，虽然这个服务器本身是使用 Python 而不是如 C 等编译式语言所写成。但由于其是专为 Zope 所设计的，又使用 DTML 相同的语言所完成，其效能很强大。

2. 面向对象数据库。

Zope 提供了一套 ZSQL 方法，这样 Zope 就也能够访问关系数据库或是外部数据库了。之所以有外部数据库一说，是因为 Zope 本身带有一个数据库，这个数据库是一个面向对象的数据库，也就是说所有 Zope 有关的数据内容都是以对象的形式存放起来的。那么 Zope 如何实现与外部的关系数据库的连接呢？

Zope 的 ZSQL 方法支持关系数据库和 Zope 对象的结合，在 Zope 中关系数据库查询的结果不是单纯的数据，他们是以对象序列的形式存在的，并且以 Zope 中特有的 Python 对象返回，所以它们具有 Zope 环境中数据对象所具有的方法和行为。并且每一条记录就是一条 Python 对象，所以说由 Zope 查询返回的结果记录集实际上是 Python 对象集。可以看出在 Zope 中，单纯的提到关系型数据或是对象型数据库已不太合适了，Zope 的数据库实际上是一种关系型数据库和对象数据库的合成（Object-Relational data integration）。

3. 内容管理器（content manager）

Zope 的内容管理器提供了非常可视化的界面及时地对我们的内容进行修改与更新。尤其是在用 DTML 语言进行网页的制作与修改上，在对网页的修改完成之后，你可以直接利用 view tab 显示修改后的网页，如果有什么语法错误，会给予及时的提示。Zope 还允许你对

standard_html_footer 和 standard_html_header 这一 DTML 方法进行编辑, 就像 Microsoft Word 的页眉页脚一样, 在网页的页眉和页脚处也插入相应的超级链接。内容管理器还提供了其他一些非常简单易用的 tab, property 可以对 DTML 方法中的属性做出调整, Undo 可以撤消你刚才所做的对对象的删除等操作, import/export 可以用来导入导出对象, Security 进行管理权限的设置。

Zope 的 folder 是一个对象也是一个容器, 它包含了许多对象, 是一个对象的集合, Zope 在 folders 之间也成功的实现了对对象的剪切, 粘帖功能。方便了对象地移动。Zope 的对象可以全部都放在 folders 中, 也可以就以对象的形式出现在 content manager 中。

4. Zope 的安全管理

Zope 的安全管理分为 4 个层次:

1) Users: 以 “Users” 或 “User objects” 的身份与 Zope 进行交互, 负责认证信息的管理和对使用人员所拥有的访问权限进行控制。

2) Roles: 表征访问权限和责任的种类, 即你所在的位置到底是 “Manager” 还是 “Author”, 角色 (role) 在身份认证和权限认证之间建立了一种关联, 相应的角色就与一定的权限和责任相对应, 在功能上与群组 (groups) 很有些类似。有三种角色上的区分, Anonymous, 代表所有的用户, 匿名的权限实际上是一个公共的权限; Manager, 管理人员角色, 管理网站或是网站的一部分, 缺省时, 所有与修改 Zope 对象有关的操作 Permission 都是 Manager 所拥有的; Owner, 任何创建对象的用户都会分到一个 Owner 角色, 缺省时, Owner 和 Manager 具有相同的权限。

3) Permissions: 具体到了对对象的具体的操作, 提供了对象访问的控制机制。就像是文件系统中的 “read, write, excute” 等功能一样, 只不过这里针对的是对象。

4) Acquisition: 提供了在 Zope 的 folders 和 subfolders 中的对象之间共享信息的机制, 对 Acquisition 和 Permissions 进行设置, 就可以实现 folders 对它其中的对象的访问控制。

Zope 提供了方便, 丰富, 完善的安全管理机制, 大大改善了我们网络管理的效率。Zope 还有一个最大的特点就是它提供了从 URL 到对象的一种简单直接的影射方式, 在 Zope 中所有的对象都是通过它们的 id 标识来引用的。

您可以去 <http://www.zope.org/Products> 下载最新版本的 Zope。在本书的配套光盘中也有 Zope 的安装程序和源码。

10.2 Zope 动态网页发展及管理系统简介

10.2.1 Zope 的内容管理器 (content manager)

Zope 的内容管理器是 Zope 之所以强大的原因之一，传统上的网站管理需要系统管理员对整个系统有很清楚的认识且有很好的记忆力。在找一个对系统很熟的人来管理系统或者还不算太难。但要在二个系统管理员中间交接可能就是大问题了。

UNIX 提供了我们非常有弹性的方法来设定整个系统的权限，所以系管理员可以根据需求的不同组织系统的权限。但也正因为弹性大，不同的人可能用不同的方法来做同一件事，所以后任的系统管理员可能对前任管理员的设定做了不正确的假设。一个好的系统管理会对所有的假设加以验证，那可能问题不大。但是不小心的管理员可能会忽略了这个步骤，那系统的安全可能就处于危险的状态之下了。

Zope 的内容管理器配合 Zope 对象导向化的设计，将内容的管理和权限管理系统合成一体。我们可以用同一个非常容易使用的界面来加、删除对象及修改对象的权限。权限的部分后面会加以说明，这里我将先说明 Zope 管理内容的方法。

10.2.2 新增一个对象

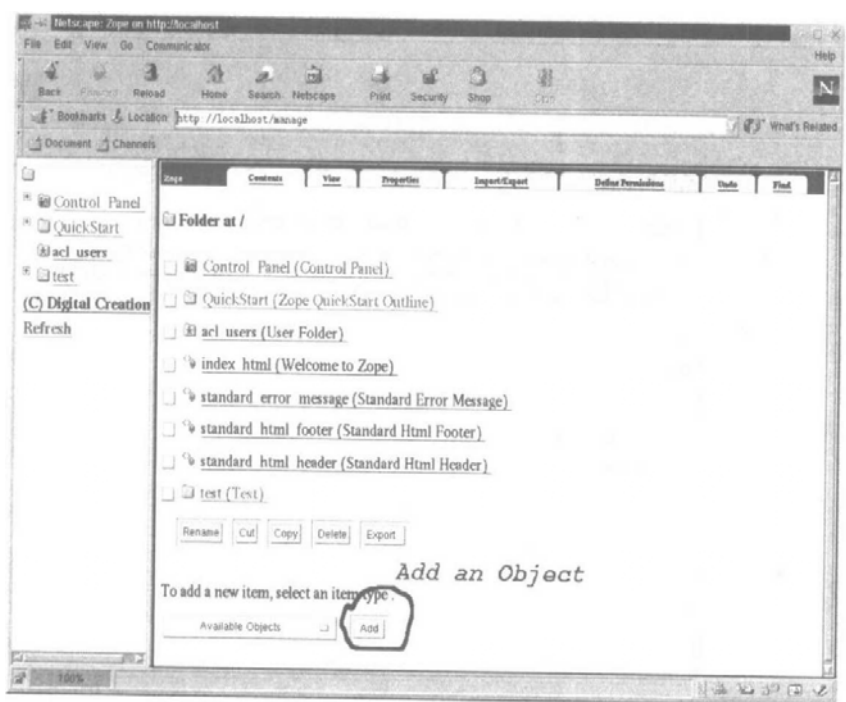


图 10.1

Zope 可以使用 PCGI 使用 Apache 作为网站服务器或是使用其内建的网页服务器。如果你使用 Apache+PCGI, Zope 的内容管理器就位于

`http://www.yoursite/manage`

如果你使用内建的服务器, 通常它会安装在 8080 这个 port 上以避免和原先的服务器相冲。此时内容管理器的 URL 变成

`http://www.yoursite:8080/manage`

使用这个 URL 便可以看见 Zope 内容管理器的主画面, 左方是目录, 右方则是目录的内容。要增加一对象, 只要先使用左下方的选单先选则一个类别, 然后按旁边的 Add 键就可以了。依据模块不同, 可以选择的对象也不尽相同。在此我只说明最常用的 DTML method、DTML document 及 Folder。另外有一些和数据库有关的在后面会提到。

基本上这 3 种对象并没有什么特别的地方, DTML method 和 DTML document 其实大同小异, 在使用上没有什么差别。它们都是用来产生 HTML 文件的样版。Folder 则在一般的目录是相同的东西, 事实上在 Zope 中 Folder 便实际被对映到真正的目录上去。使用这叁种对象就足以构一个很复杂的网站了。

现在我们首先新增一个 Folder, 先选 Folder 然后按 Add。你会看到下图。

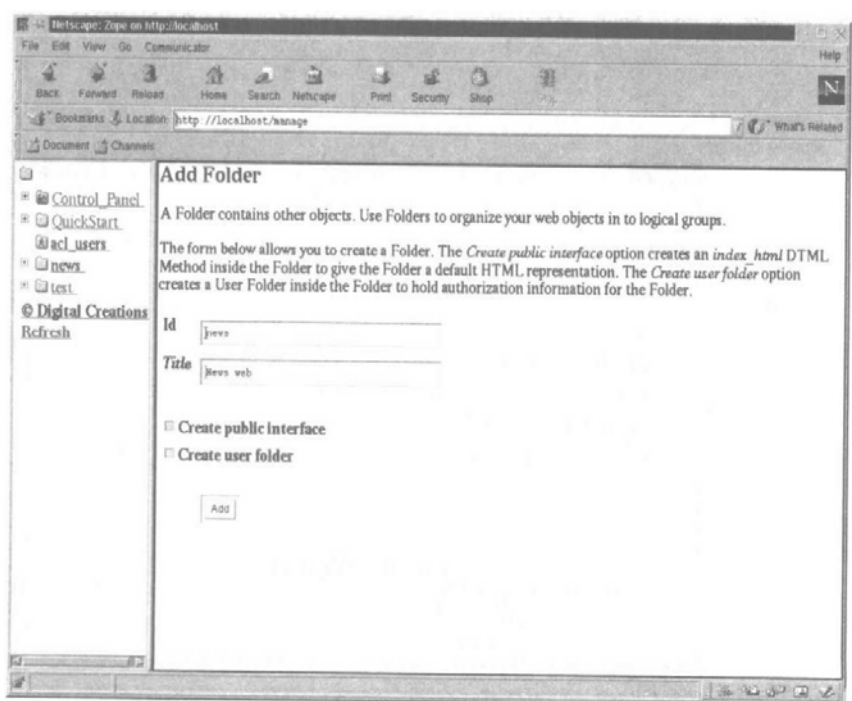


图 10.2

输入 id 和 title 就可以新增一个 folder 了。所有的 Zope 对象都有一个 id 和 title。id 是用来在 python module 或是 DTML 命令中参考一个对象用, 而 title 则是用来当作浏览器的 title 用。

10.2.3 编辑一个 DTML 文件对象

接下来点选这个 folder 就可以进入目录之中, 你也可以用屏幕左方的目录树来选择目前显示的目录。在目录中你会看到 Zope 会自动帮你产生两个文件, 一个是 acl_user, 它是 Zope 的安全管理系统的一部分, 用来设定这个 folder 中每一个用户的权限。另一个是 index_html, 当你输入 `http://www.yoursite/folder` 时, folder 中的 index_html 文件便会被用来产生目录的 view。这个动作和 Apache 一样, 只是 default 的首页由一个叫 index.html 的 HTML 文件变成一个叫 index_html 的 DTML 文件而已。

如果你要改变对象的内容, 只要点选 folder 的 icon 即可。当你选择 index_html 后, 你会看到下面的画面。

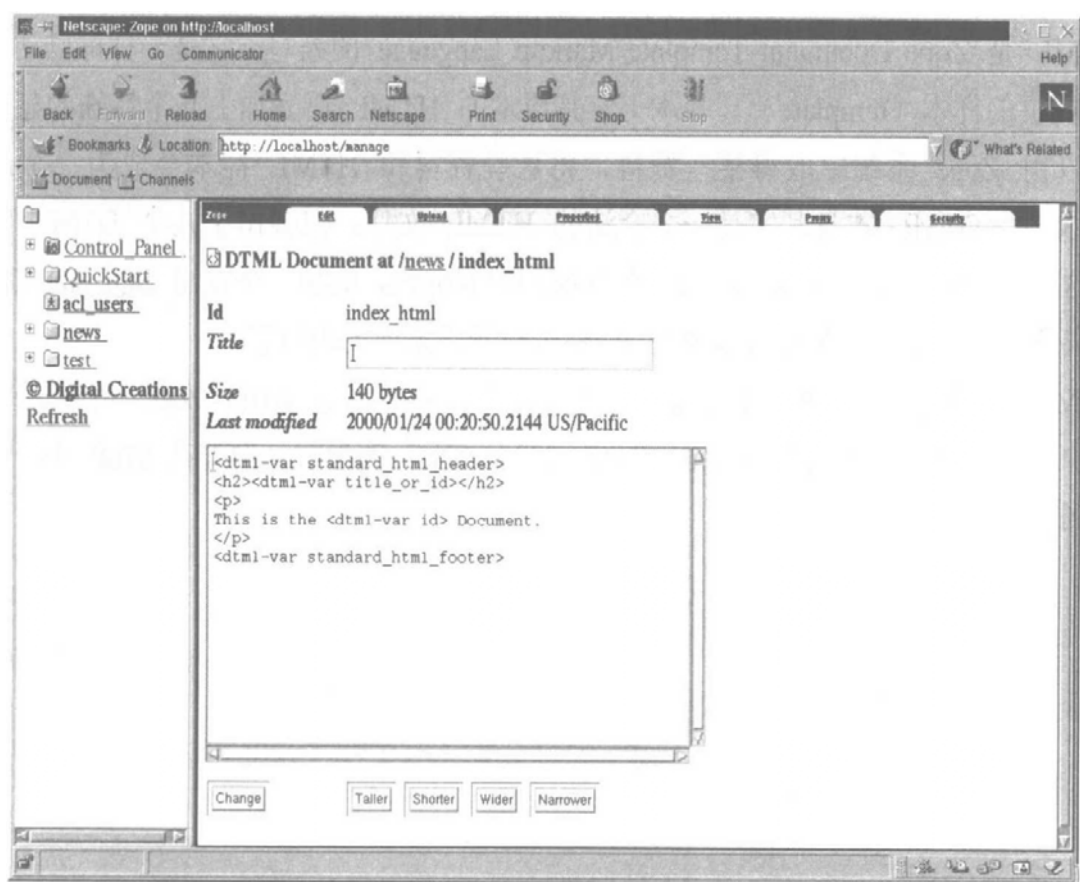


图 10.3

你可以看到 id 就是 index_html。记得，Zope 的 id 约略等于文件名，是用来参考 Zope 对象用的。下面文本框的内容就是一个典型的 DTML 文件，它使用标准的文件头和文件尾，这使得网站的外观更有一致性，也使得改变网站外观更为容易。

当你改变了文件的内容后，可以用上方的 view tab 来预视其效果。当然，记得先按 change 储存你的改变。你也可以选择使用你习惯的编辑器改变文件的内容，然后使用 Upload 这个 tab 来上传更新的文件。

10.2.4 文件的属性

每一个 Zope 对象都可以定义一些属性，这些属性可以在其他的对象中以 DTML tag 引入。例如你可以定义一个叫 company_address 的属性，然后在所有需要公司电话的地方都参考这个属性，而不直接输入电话号码。如此一来，万一公司的电话号码改变，只是改变这个值就可以了。而不必一一的到每一个网页中去更改公司的电话号码。

10.2.5 Zope Document Template Markup Language

DTML 是 Zope Document Template Markup Language 的缩写。它是一种可以动态产生 HTML 网页的样本 (template)，基本上它是原本的 HTML 语言加上几个特殊的延伸。使得它可以由 Zope 系统中取得某些资料，将这些资料和 HTML 样本合成为一个最终的 HTML 文件。DTML 本身其实是一个合法的 HTML 文件，但多出了几个 DTML 特有的 TAG。这些 TAG 通常会被 Zope 用一些其他的内容取代，例如一个发出 SQL 的 TAG 会被其结果所取代，如此我们就可以使用 web 来查看数据库的内容了。

DTML 和 ASP 及 PHP 最大的不同之处在于它并不是将 script 放在一个特殊的文字区块之中。而是将每一个语言的关键字写成 HTML TAG 的形式。例如在 PHP 中，我们可能会看到

```
<?
if ($a == 0) {
    echo "hello"
}
?>
```

但在 DTML 中，相同的程序则会被写成

```
<dtml-if a==0>
```

```
hello  
</dtml>
```

你可以发现，和 PHP 及 ASP 比起来，它更接近一个样本语言（template language），而 PHP 和 ASP 则比较接近传统的语言，只是它们是被放在一个 HTML 文件中就是了。

接下来，我们逐一简单的介绍 DTML 的用法。当然，DTML 的参考手册永远是最详细的参考资料，不过本文的内容可能已经足够大多数人的需要了。

10.2.6 特殊 TAG 的格式

你可以用下面三种格式之一将 DTML TAG 放入一个 HTML 文件之中。

```
<dtml-name>  
% (name) x  
<!--#name-->
```

第一种是使用特殊 TAG 的方式将 DTML 命令放入 HTML 文件中，其后可以跟着一个名字及多个属性串行。例如，

```
<dtml-var date fmt=Date len=long>
```

这个 TAG 在将来会被变量 `date` 的值取代，而其格式则由后面的二个属性来决定。有时我们会用较短的第二种格式来取代，它的格式有些类似 `printf` 的格式参数。例如

```
% (date fmt=Date) s
```

这种格式这将 HTML 原有的逸出序列（escape sequence）加以延伸通常用于比较简单的文字插入，我并不鼓励大家把它当成一般的应用，因为以我的观点而言，用它写出来的程序可性并不太好。第3种格式则是由 Server Side Include 而来。当然，在 DTML 中，`name` 可以是任何合法的 DTML 命令。它通常被用来调用一个已经被定义的 DTML 变量（variable）或是方法（method）。

10.2.7 变量与运算式

变量的引用是 DTML 最重要的一个部分，我们通常比较少用 DTML 写一个很复杂的程序。超过一定复杂度的程序最好还是使用外部方法（external method）的方法来完成，

因为 DTML 是一个纯直译式的语言，它的效能远不及能产生 byte code 的 python 及 perl，当然更不及 C 及其他编译式的语言。

一个 DTML 变量可能被在许多个不同的地方定义，Zope 本身是由 Python 写成的，所以一个 DTML 变量可能被对映成一个 Python 变量。此外它也可能指向一个 Zope 对象，或

1) 由 Python 提供的环境，Python 程序在执行一个 DTML 程序时可以同时将环境传入 engine 之中。DTML 解译器会首先检查这个环境的内容。

2) 如果变量名称是 document_id 或是 document_title，则 DTML 方法 (method) 或文件 (document) 的 id 及 title 会被使用。

3) 对象的 properties。请注意，Zope 的 properties 是会遗传的，所以包括了对象本身及其上层 Folder 的 properties 在内。

4) Zope 定义的 REQUEST 环境。

5) HTML Form 所定义的变量

6) HTML Cookie 所定义的变量

7) 如果变量名称是 URL_n，如 URL₀, URL₁,。它们会被如下转译，假设文件的 URL 是 http://localhost/A/B/C/D,

URL₀ : http://localhost/A/B/C/D

URL₁ : http://localhost/A/B/C

URL₂ : http://localhost/A/B

URL₃ : http://localhost/A

URL₄ : http://localhost

URL₅: undefined

8) CGI 变量，如 REMOTE_HOST 等。

9) 如果变量的名称以 HTTP_开头，则试图由 HTTP 的 header 内找到变量的值。如 HTTP_REFERER 会去找 'REFEREE:' 这个 header 的值。

10) 如果变量的名称为 BASE_n，则使用和 URL_n 一样的规则取值。但此时有值的变量可能不像 URL 那样多。例如对象向 ZPublish 注册的 URL 是 http://localhost/A/B 时，只有 URL₀, URL₁, URL₂ 有值。其他的都是 undefined。

DTML 的运算式 (expression) 和一般的程序语言并没有什么不同。除了上述的变量外，DTML 还提供了一大群的函数供我们使用。

要设定一个变量的值，你必需使用 `<dtml-let>` 这个 tag。这个 tag 可以一次设定多个变量的值。

```
<dtml-let a=1 b=2 c=a*b>
<dtml-var a>*<dtml-var b>=<dtml-var c>
```

10.2.8 条件式

DTML 提供了 `<dtml-if>` 和 `<dtml-unless>` 两个不同的条件 tag。它们的使用非常的直觉，

```
<dtml-if "a>0">
a is zero
<dtml-else>
a is not zero
</dtml-if>
```

10.2.9 循环

`<dtml-in>` 可能是 DTML 最强大的 tag。它使得 DTML 可以执行循环的功能。例如下面的程序可以用来将一个串行变量的值用 Table 列出，

```
<table>
  <dtml-in employee sort=name>
  <tr>
    <td> <dtml-var name> </td>
    <td> <dtml-var age> </td>
  </tr>
  <dtml-else>
    You have no employee data.
  </dtml-in>
</table>
```

请注意程序中的 `<dtml-var name>` 事实上是 `<dtml-var sequence_item.name>`。sequence_item 是 employee 串行中的一项，它可能是 SQL 返回值的一个记录，或是一个串行变量的一项。

DTML-IN 这个 tag 会定义一大堆的变量用来做 iteration 用。sequence_item 只是其中

的一个。在这里我不打算详细说明，它实在太复杂了，将来我也许会用一篇文章来说明它。下面我只用一个简单的例子说明它强大的功能。

```
<dtml-in employee name=sort>
<dtml-if sequence-start>
  <table>
</dtml-if>

<dhtml-in>
<td> <tml-var name> </td>
<td>
  <dtml-if "age > 30">
    Old
  <dtml-else>
    Young
  </dtml-if>
</td>

<dtml-if sequence-end>
  </table>
  <p>The average age is <!--#mean-age-->
</dtml-if>
<dhtml-in>
```

10.2.10 Zope 的安全机制

设定 web 的安全机制一直都不是一个很容易的事，一不小心网站管理员很可能会在网站中开了个天窗。虽然被发现的机会不大，但一旦被发现那小则被开个玩笑，严重时可能整个网站都不知道到那去了。

Zope 将整个网站组织成一个对象树，你可以在树的每一个阶层设定不同的安全等级。这个等级是会遗传的，也就是说上层节点所设的等级会自动遗传到下一层去。当然你也可以为任何一个节点设定一个新的安全等级以取代遗传下来的安全等级。所有的安全设定都可以用一个很容易使用的 web 界面来改变。

在 Zope 中，安全机制包括了三个部分。

1) 用户，指的是每一个 HTTP 调用都会有一个用户。这通常是通过 web 的认证机制来决定的。

2) 权限 (permission)，每一个对象都有一组方法 (method)，不同的对象包括的方法亦不相同。可能的方法包括了，

- 显示文件
- 删除文件
- 修改文件
- 增加属性
- 修改属性
- 上传资料
- 其他....

3) 角色 (role)，不同的角色可以使用的权限亦不相同。例如系统管理者 (admin) 可以执行所有的命令，但文件拥有者 (owner) 通常只能显示、修改文件而已。

使用这三者就可以组合出任何我们想要的权限组合，例如。一个用户可以扮演角色 A，而对象 1 让角色 A 可以有全部的权限，而对象 B 只给角色 A 观看的权限。如此这个用户就只有修改对象 1 的能力，而没有修改对象 B 的能力了。

如果把 UNIX 的权限对照到这个模型之中，UNIX 用户还是 Zope 的用户，而 UNIX 只定义了一组无法改变的角色。每一个用户都可以扮演 (自己、同群组或是其他) 三个角色之一，而权限则只能是读取、写入和执行三者之一。

10.2.11 Zope 如何决定用户

Zope 使用 HTTP 的机制来决定用户的身份，在一般基本的设定下，网页服务器会要求用户输入 id 及密码。在 Zope 的内容管理器中，每一个目录中都有一个叫做 `acl_user` 的项目，你可以使用它来管理用户的数据。

Zope 在每一个子目录中都可以定义一个 `acl_user`，这意谓着每一个子目录内都可以有不同的用户。这对于网站的管理是很方便的，例如你可以为每一个不同的功能定义一个子目录。例如在 LinuxFab 中我们有新闻、软体和专栏三个网站，我们可以在最顶层的子目录中创造 `news`、`software` 及 `column` 三个目录。然后在三个目录内定义不同的用户，如此一来三个网络的用户是完全独立的。在 `news` 中的系统管理员对 `software` 而言完全没有意义，如此可以让三个系统管理员完全独立的管理一个网站的不同部分，而不会有权限的问题。

1. 如何新增一个用户

前面说过，在 Zope 中新增一个用户只要选择执行每一个子目录中的 `acl_user` 方法即可。

2. 如何新增一个角色

角色的定义在 `acl_user` 界面的最下方，你只要输入一个名称然后按 `Add` 按钮就可以了。

3. 如何设定 Zope 的权限

每一个 Zope 中的对象，包括方法、文件、目录、数据库等都可以分别设定其权限。每一个对象的权限可以自行定义，或者是由继承而来。通常对象的属性都会由继承而来，我们不会一个一个的去定义每一个对象的权限。而是把相同权限的对象放在同一个目录或其子目录之下，然后在最上层的目录定义其权限就可以了。

10.2.12 结语

Zope 的功能当然不是如此，其他如论坛、数据库存取、企业级分布式管理及 python 扩语言等限于篇幅就不在本文多加描述，在将来有机会再专门解释。但 Zope 强大之处已不言而喻，它可能是目前在 Linux 上可以和 Code Fusion 之类系统抗衡的 Web 应用程序发展及管理系统。它最大的一个问题是采用 Python 这套语言，虽然 Python 是一个很好用且强大的 script 语言，但由于目前熟习它的人还是不如 C/C++，perl 或 java 来得多。使得它的发展还有待观察，也许它会成为 Python 的 killer 应用程序也说不定。

在这一点上，另一个相似的系统 BLADE 可能就占了优势。BLADE 使用 CORBA 来架构它的系统，因为 CORBA 本身与语言无关的能力，使得 BLADE 的应用程序可以轻易的用任何可以支持 CORBA 的语言来写作。不过 Blade 现在所提供的功能还很有限，离一个真正实用的系统还有些距离。

虽然 Zope 本身不支持，但它支持 XML-RPC，所以理论上我们还是可以用各种不同的语言来写 Zope 的程序，只是我没试过，不敢说它 XML-RPC 的功能到底有多完整。也许有经验的人可以给我们一些意见。

10.3 Zope 与 Python 的关系

ZOPE 构架于 PYTHON 之上，ZOPE 中的每一个部件，如 FOLDER，ACL_USER 等都与 PYTHON 中的对象相对应，因此我们在写 DTML 文件时经常会与 PYTHON 中的对

象和变量打交道，所以了解一下 ZOPE 和 PYTHON 之间的调用关系对开发是大有好处的。

通常 PYTHON 中需要被 ZOPE 直接调用的 CLASS，如 Folder，UserFolder 等一般都具有以下结构（以 UserFolder 为例）：

```
class UserFolder (BasicUserFolder) :
    """Standard UserFolder object
    A UserFolder holds User objects which contain information
    about users including name, password domain, and roles.
    UserFolders function chiefly to control access by authenticating
    users and binding them to a collection of roles."""
    # 该类的类型代码，在 DTML 中通过类型代码获取该类的实例
    # 如 objectValues (['User Folder']) 就是获取当前目录下所有的 User Folder
    # 对象（尽管对象的 id 为 acl_user）。
    meta_type='User Folder'
    # 该类的 id，作为一种特殊对象，User Folder 的 id 始终为 acl_user 因为每个
    # 目录下最多只存在一个 User Folder 对象，不存在重复问题。
    # 而对于其他对象，如 Folder，则同一目录下的对象 id 就必须不同。
    id = 'acl_users'
    title = 'User Folder'
    # 该类显示时的图标，我们看到 UserFolder 的图标和 Folder 的图标不同，区别就
    # 在这里。
    icon = 'p_/UserFolder'
    # 这里是该类的构造函数，self 表示该类本身，相当于 C++ 中的 this
    def __init__ (self) :
        self.data=PersistentMapping ()
    # 以下是该类的成员函数，其中以 _开头的函数在 DTML 中无法直接调用，类似于 C++
    # 中的保护函数。反之则为公用函数。变量亦然。
    def getUserNames (self) :
        """Return a list of usernames"""
        names=self.data.keys ()
        names.sort ()
        return names
```

```
def getUsers (self) :
    """Return a list of user objects"""
    data=self.data
    names=data.keys ()
    names.sort ()
    users=[]
    f=users.append
    for n in names:
        f (data[n])
    return users

def getUser (self, name) :
    """Return the named user object or None"""
    return self.data.get (name, None)

def _doAddUser (self, name, password, roles, domains) :
    """Create a new user"""
    self.data[name]=User (name, password, roles, domains)

def _doChangeUser (self, name, password, roles, domains) :
    user=self.data[name]
    user.__=password
    user.roles=roles
    user.domains=domains

def _doDelUsers (self, names) :
    for name in names:
        del self.data[name]
```

通常在这些直接被用户调用的类中还要有以下定义，下列内容摘自 `BasicUserFolder` 即 `UserFolder` 的基类。

```
class BasicUserFolder (Implicit, Persistent, Navigation, Tabs,
    RoleManager,
    Item, App.Undo.UndoSupport) :
    """Base class for UserFolder-like objects"""
    meta_type='User Folder'
    id ='acl_users'
```

```

title='User Folder
isPrincipiaFolderish=1
isAUserFolder=1
# 这里定义了管理页面的结构，即页面中 TAB 的内容。label 是 TAB 名，action
# 是链接。
manage_options= (
    {'label': 'Contents', 'action': 'manage_main'},
    {'label': 'Security', 'action': 'manage_access'},
    {'label': 'Undo', 'action': 'manage_UndoForm'},
)

# 这里定义了该类中固有的权限。
_ac_permissions_= (
    ( 'Manage users', ( 'manage_users', 'getUserNames', 'getUser', 'getUsers',
    )),
)

# 以下是把 DTML 中的 action 重定向为 DTML 页面，
# 如上文中的 action = manage_access 就重定向为 mainUser.dtml
mainUser=HTMLFile ('mainUser', globals ())
_add_User=HTMLFile ('addUser', globals ())
remote_user_mode=_remote_user_mode}
_editUser=HTMLFile ('editUser', globals ())
remote_user_mode=_remote_user_mode}
manage=manage_main=_mainUser

```

当用户进入某一页面时，系统将该页面与 PYTHON 中相应的类的实例相关联，因此对该页面来说，该实例中的一切公共函数和变量均为可见，页面可以自由地调用这些函数和变量。

例如我在根目录下的 index_html.dtml 文件中加入以下内容：

```

<dtml-in "objectValues (['User Folder'])">
<dtml-in "getUsers ()">
<p><dtml-var name></p>
</dtml-in>

```

```
</dtml-in>
```

假设我在根目录下的 `acl_users` 目录下有以下用户: `frank`, `xyz` 则当我调用该页面时会出现以下内容:

```
...  
frank  
xyz  
...
```

在这里, `getUsers()` 是 `UserFolder` 类的公共成员函数, 可供 DTML 直接调用。

`UserFolder` 类中还有很多保护函数, 在 DTML 中无法直接调用。不过在必要时我们也可以修改它, 将其变为公共函数。如我们将函数 `_doAddUser` 改名为 `doAddUser`, 并在 `index_html.dtml` 中加入下列内容:

```
<dtml-in "objectValues (['User Folder']) ">  
<dtml-call "doAddUser ('crystal', 'crystal', ['manager'], []) ">  
</dtml-in>
```

则当我调用该页面时会向 `acl_users` 目录下添加一个名为 `crystal` 的用户。

第 11 章 Python 实例

本章节分类举了一些 Python 语言的实例，内容由浅入深，基本覆盖了 Python 语言编程的各个方面。文中给出了程序代码以及运行后的截图。程序代码及所用到的相关资源（文件、图片等）均可在配套光盘中找到。

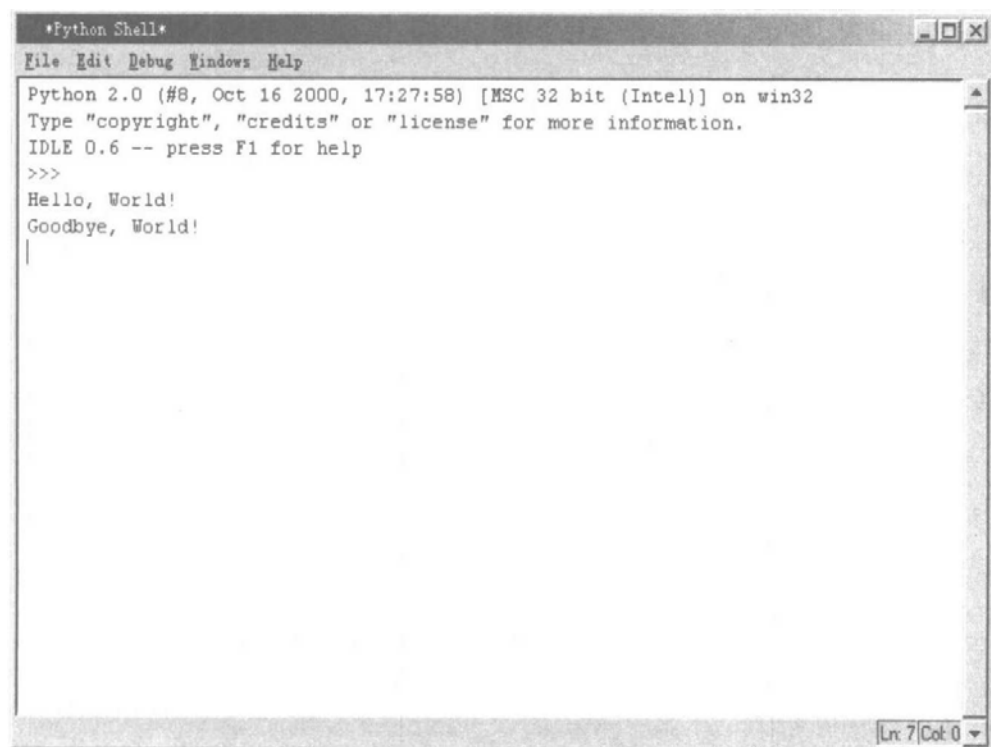
11.1 Hello World 程序

这是经典的 Hello World 程序。

例 1: helloworld.py

```
#!c:\python\python.exe  
print "Hello, World!"  
print "Goodbye, World!"
```

运行结果：



11.2 变量和控制流

这是一个判断是否为闰年的程序。

例 1: leap.py

```
#!/c:\python\python.exe
import sys
import string

if len ( sys.argv ) < 2 :
    print "Usage: leap.py year, year, year..."
    sys.exit ( 0 )

for i in sys.argv[ 1 : ] :
    try :
        y = string.atoi ( i )
    except :
        print i, "is not a year."
        continue

    leap = "no"

    if y % 400 == 0 :
        leap = "yes"
    elif y % 100 == 0 :
        leap = "no"
    elif y % 4 == 0 :
        leap = "yes"
    else :
        leap = "no"

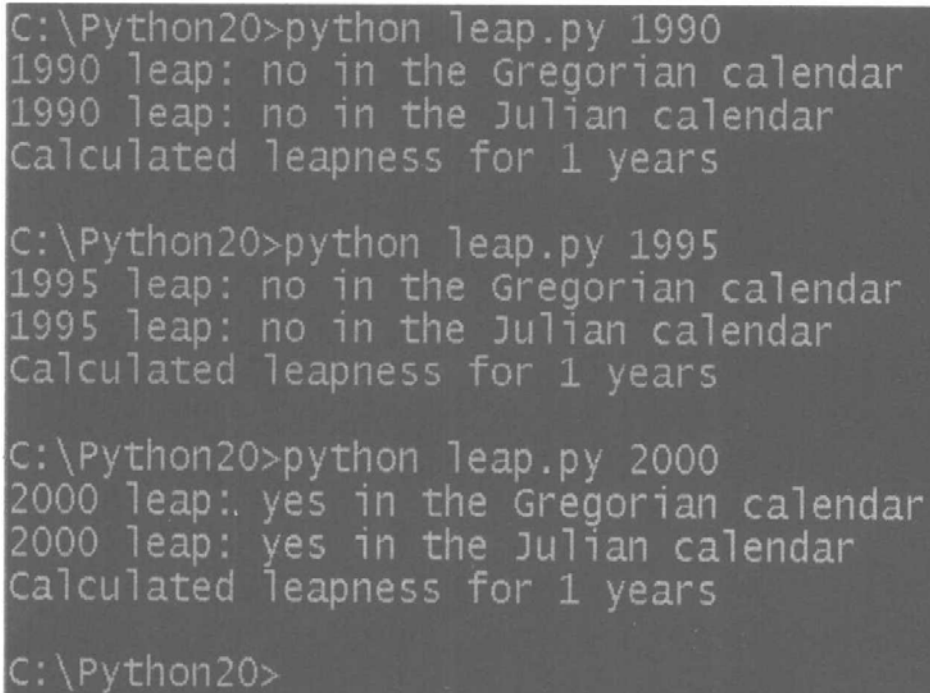
    print y, "leap:", leap, "in the Gregorian calendar"

    if y % 4 == 0 :
        leap = "yes"
    else :
```

```
        leap = "no"
    print y, "leap:", leap, "in the Julian calendar"

    print "Calculated leapness for", len ( sys.argv ) - 1, "years"
```

运行结果:



```
C:\Python20>python leap.py 1990
1990 leap: no in the Gregorian calendar
1990 leap: no in the Julian calendar
Calculated leapness for 1 years

C:\Python20>python leap.py 1995
1995 leap: no in the Gregorian calendar
1995 leap: no in the Julian calendar
Calculated leapness for 1 years

C:\Python20>python leap.py 2000
2000 leap: yes in the Gregorian calendar
2000 leap: yes in the Julian calendar
Calculated leapness for 1 years

C:\Python20>
```

11.3 基本数据类型

这是一个测试整型数据类型的程序。

例 1: inttest.py

```
#!c:\python\python.exe
from sys import *

z = long ( maxint )
z1 = -z

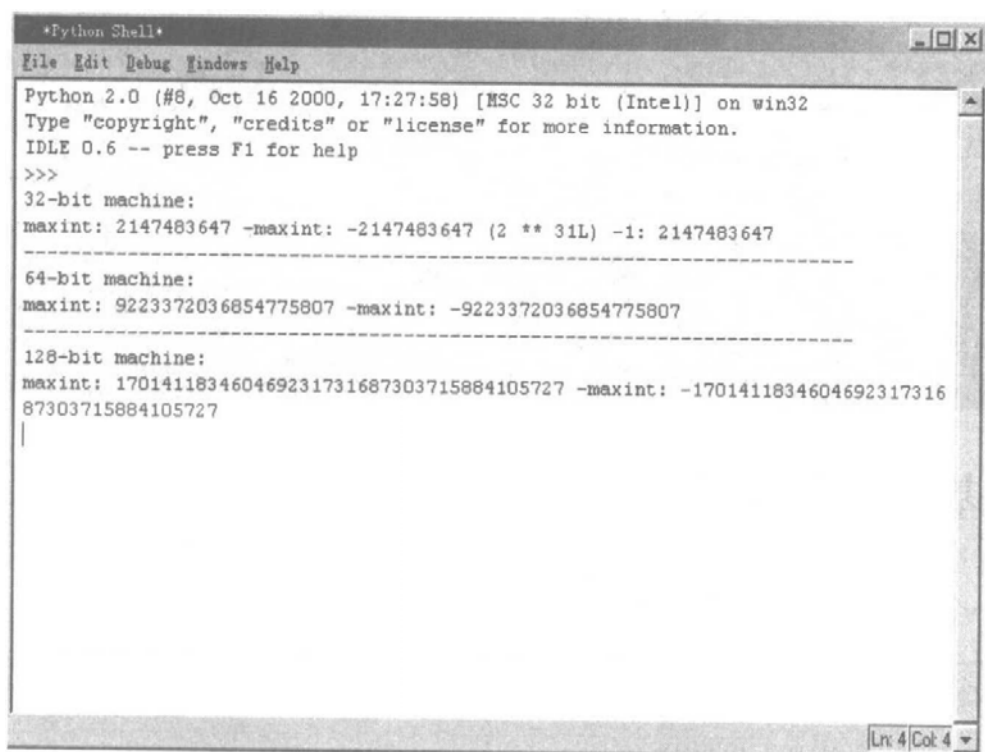
print "32-bit machine:"
print "maxint:", z, "-maxint:", -z, " (2 ** 31L) -1:", (2 ** 31L) -1
```

```
print
"-----"

y = (2 ** 63L) - 1
print "64-bit machine:"
print "maxint:", y, "-maxint:", -y
print
"-----"

z = (2 ** 127L) - 1
print "128-bit machine:"
print "maxint:", z, "-maxint:", -z
```

运行结果:



```
*Python Shell*
File Edit Debug Windows Help
Python 2.0 (#8, Oct 16 2000, 17:27:58) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.6 -- press F1 for help
>>>
32-bit machine:
maxint: 2147483647 -maxint: -2147483647 (2 ** 31L) -1: 2147483647
-----
64-bit machine:
maxint: 9223372036854775807 -maxint: -9223372036854775807
-----
128-bit machine:
maxint: 170141183460469231731687303715884105727 -maxint: -1701411834604692317316
87303715884105727
|
```

11.4 基本数据类型II: 次序和字典

例 1: `ascii.py`

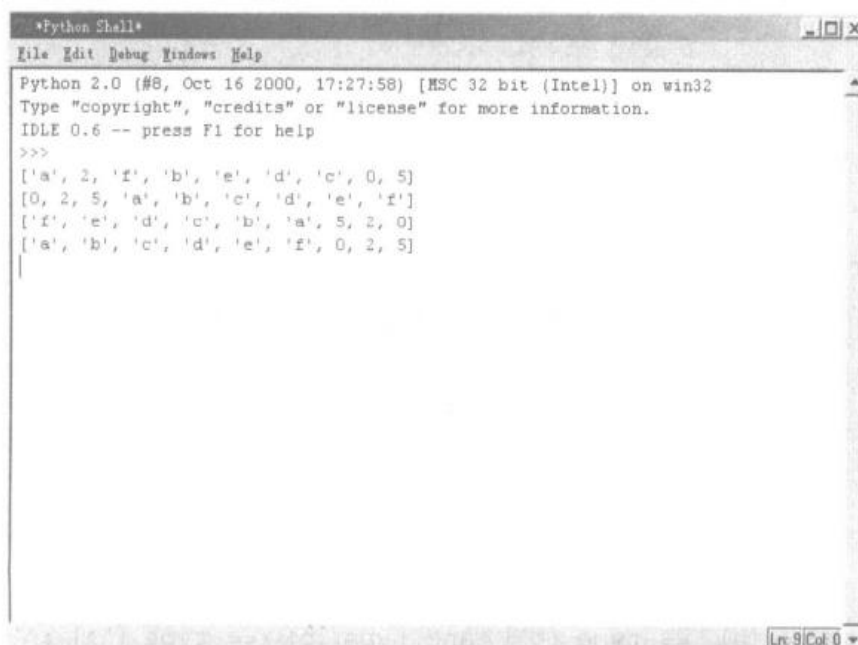
```
#!c:\python\python.exe
i = 0
while i < 256 :
```



```
        if a < b:
            return -1
        elif a > b:
            return 1
        return 0
    if type(a) == type(0) and type(b) == type(""):
        return 1
    if type(a) == type("") and type(b) == type(0):
        return -1
    return 0

print l
l.sort ()
print l
l.reverse ()
print l
l.sort (srt)
print l
```

执行结果:



```
Python 2.0 (#6, Oct 16 2000, 17:27:58) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.6 -- press F1 for help
>>>
['a', 2, 'f', 'b', 'e', 'd', 'c', 0, 5]
[0, 2, 5, 'a', 'b', 'c', 'd', 'e', 'f']
['f', 'e', 'd', 'c', 'b', 'a', 5, 2, 0]
['a', 'b', 'c', 'd', 'e', 'f', 0, 2, 5]
```

例 3: xtst.py

```
#!c:\python\python.exe

s = "c:\\Beginner"
s1 = "e:" "\\ " "Beginner"
s2 = s1 + \
    "\\tst.py"

print "This is a DOS path:", s
print "This is a DOS path:". s1
print "This is a DOS path:". s2

s3 = "I contain 'single' quotes"
s4 = 'I contain "double" quotes'
s5 = """I am a triple-quoted string that contains \"\"\" quotes"""

print s3
print s4
print s5

s6 = "I contain\t\t\tthree\t\t\ttabs"
s7 = "I contain a\t\v\tvertical tab"
s8 = "I contain a\t\a\tBELL. which you can hear"

print s6
print s7
print s8

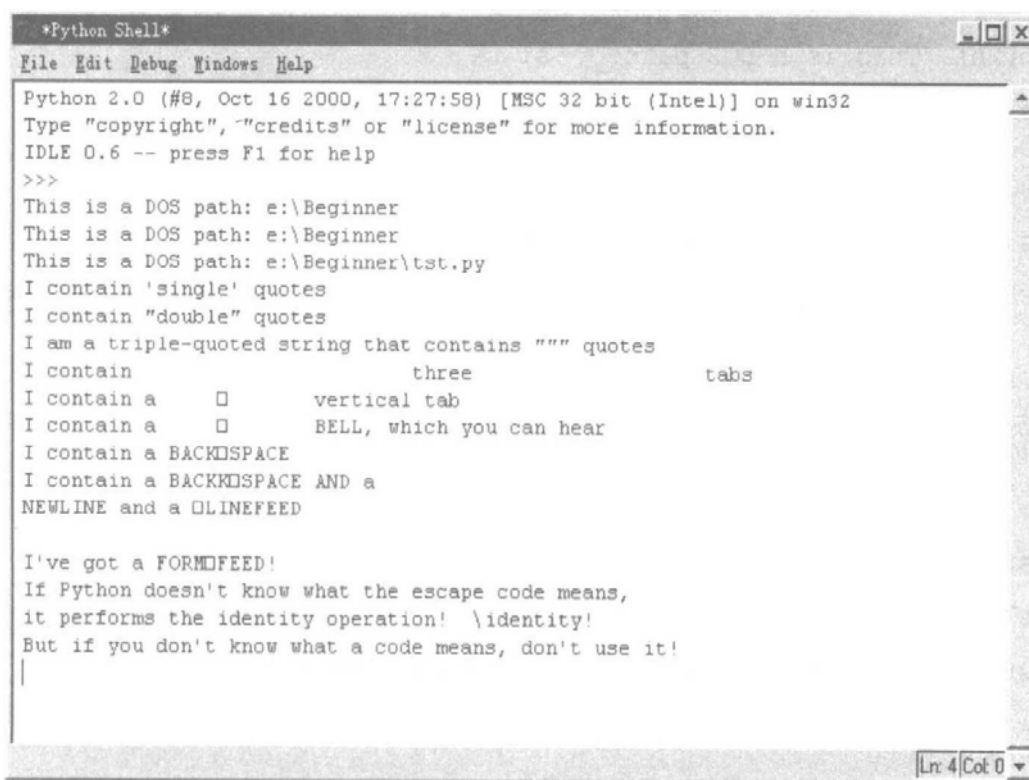
s9 = "I contain a BACK\bSPACE"
s10 = "I contain a BACKK\bSPACE AND a \nNEWLINE and a \rLINEFEED"
s11 = "I've got a FORM\I FEED!"

print s9
print s10
print
print s11
```

```
s12 = "If Python doesn't know what the escape code means, \n" \
      "it performs the identity operation! \identity!"
s13 = "But if you don't know what a code means, don't use it!"

print s12
print s13
```

执行结果:



```
*Python Shell*
File Edit Debug Windows Help
Python 2.0 (#8, Oct 16 2000, 17:27:58) [MSC 32 bit (Intel)] on win32
Type "copyright", "-credits" or "license" for more information.
IDLE 0.6 -- press F1 for help
>>>
This is a DOS path: e:\Beginner
This is a DOS path: e:\Beginner
This is a DOS path: e:\Beginner\tst.py
I contain 'single' quotes
I contain "double" quotes
I am a triple-quoted string that contains """ quotes
I contain          three          tabs
I contain a      vertical tab
I contain a      BELL, which you can hear
I contain a BACKSPACE
I contain a BACKSPACE AND a
NEWLINE and a OLDFEED

I've got a FORMFEED!
If Python doesn't know what the escape code means,
it performs the identity operation! \identity!
But if you don't know what a code means, don't use it!
|
Ln: 4|Col: 0
```

11.5 函数和模块

例 1 : leap2.py

```
#!c:\python\python.exe
import sys
import string
def julian_leap (y=2000) :
    if (y%4) == 0:
        return 1
```

```
    return 0
def gregorian_leap (y=2000) :
    if (y%400) == 0:
        return 1
    elif (y%100) == 0:
        return 0
    elif (y%4) == 0:
        return 1
    return 0

if __name__ == "__main__":
    years = [1999, 2000, 2001, 1900]

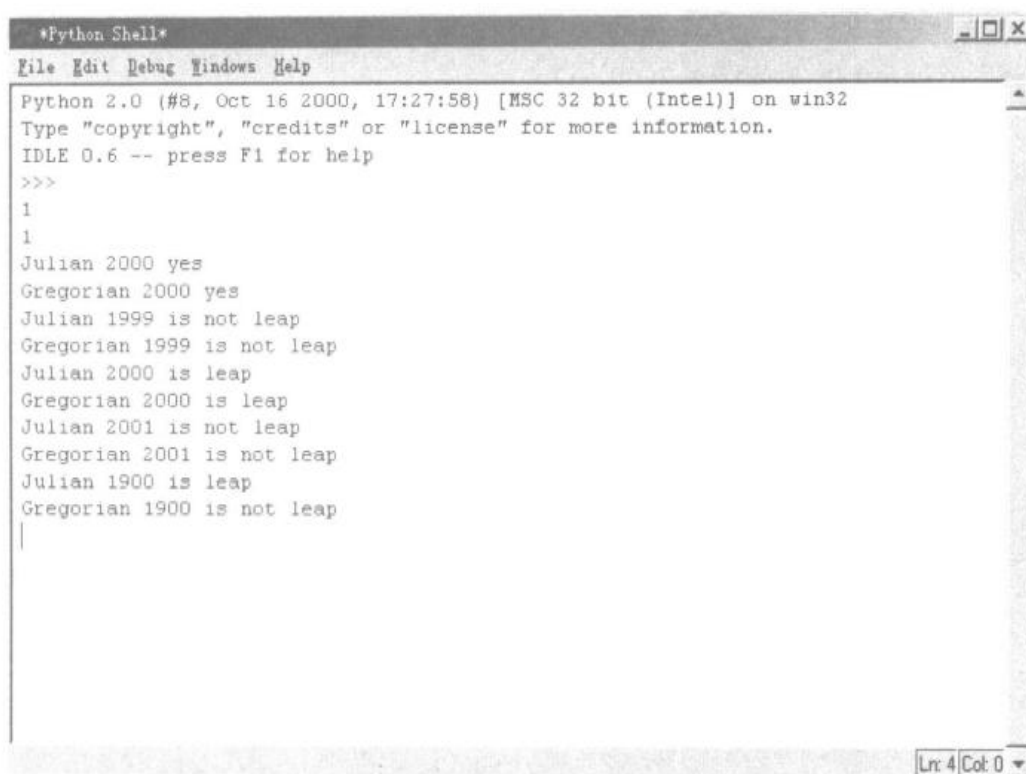
    print julian_leap ()
    print gregorian_leap ()

    if julian_leap () :
        print "Julian 2000 yes"
    if gregorian_leap () :
        print "Gregorian 2000 yes"

    for x in years:
        if julian_leap (x) :
            print "Julian". x, "is leap"
        else:
            print "Julian". x, "is not leap"

        if gregorian_leap (x) :
            print "Gregorian". x, "is leap"
        else:
            print "Gregorian". x, "is not leap"
    #else:
    #    print __name__
```

执行结果:



```
Python 2.0 (#8, Oct 16 2000, 17:27:58) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.6 -- press F1 for help
>>>
1
1
Julian 2000 yes
Gregorian 2000 yes
Julian 1999 is not leap
Gregorian 1999 is not leap
Julian 2000 is leap
Gregorian 2000 is leap
Julian 2001 is not leap
Gregorian 2001 is not leap
Julian 1900 is leap
Gregorian 1900 is not leap
|
```

例 2: ly.py

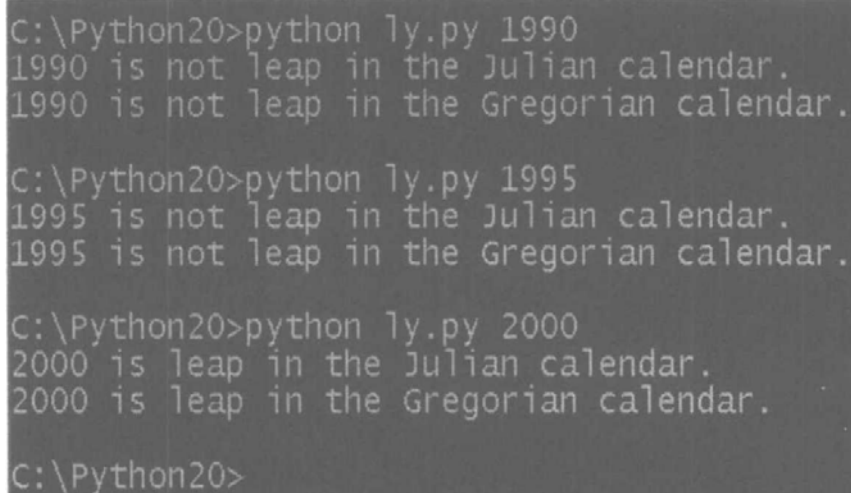
```
#!/c:\python\python.exe

import sys
import string
import leap2

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print "Usage:", sys.argv[0], "year year year..."
        sys.exit(1)
    else:
        for i in sys.argv[1:]:
            y = string.atoi(i)
            j = leap2.julian_leap(y)
            g = leap2.gregorian_leap(y)
            if j != 0:
                print i, "is leap in the Julian calendar."
            else:
```

```
        print i, "is not leap in the Julian calendar."  
    if g != 0:  
        print i, "is leap in the Gregorian calendar."  
    else:  
        print i, "is not leap in the Gregorian calendar."
```

执行结果:



```
C:\Python20>python ly.py 1990  
1990 is not leap in the Julian calendar.  
1990 is not leap in the Gregorian calendar.  
  
C:\Python20>python ly.py 1995  
1995 is not leap in the Julian calendar.  
1995 is not leap in the Gregorian calendar.  
  
C:\Python20>python ly.py 2000  
2000 is leap in the Julian calendar.  
2000 is leap in the Gregorian calendar.  
  
C:\Python20>
```

11.6 有用的混合运算

这是一个对文件操作（读文件）的程序。

例 1: readit.py

```
#!c:\python\python.exe  
  
import sys  
import os  
import errno  
  
if __name__ == "__main__" :  
    if len (sys.argv) >1:  
        try:  
            f=open (sys.argv[1], "rb")  
        except:
```

```
x=sys.exc_info ()
print type (x)
print type (x[0])
print dir (x[0])
print type (x[1])
print dir (x[1])
print x[1].errno
print type (x[2])
print "No file named %s!" % (sys.argv[1],)
sys.exit (1)
while 1:
    t=f.readline ()
    if t=='':
        break
    if '\n' in t:
        t=t[:-1]
    if '\r' in t:
        t=t[:-1]
    sys.stdout.write (t + '\n')
f.close ()
```

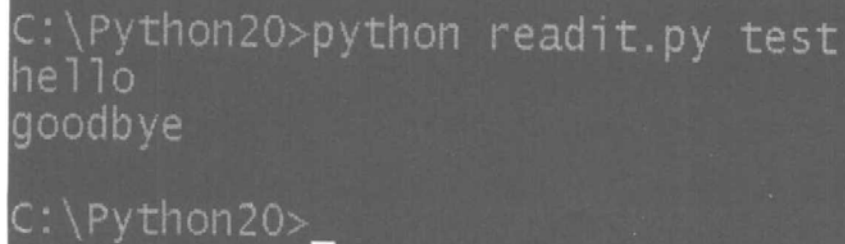
文件“test”的内容:

hello

goodbye

执行python readit.py test

结果:



```
C:\Python20>python readit.py test
hello
goodbye
C:\Python20>_
```

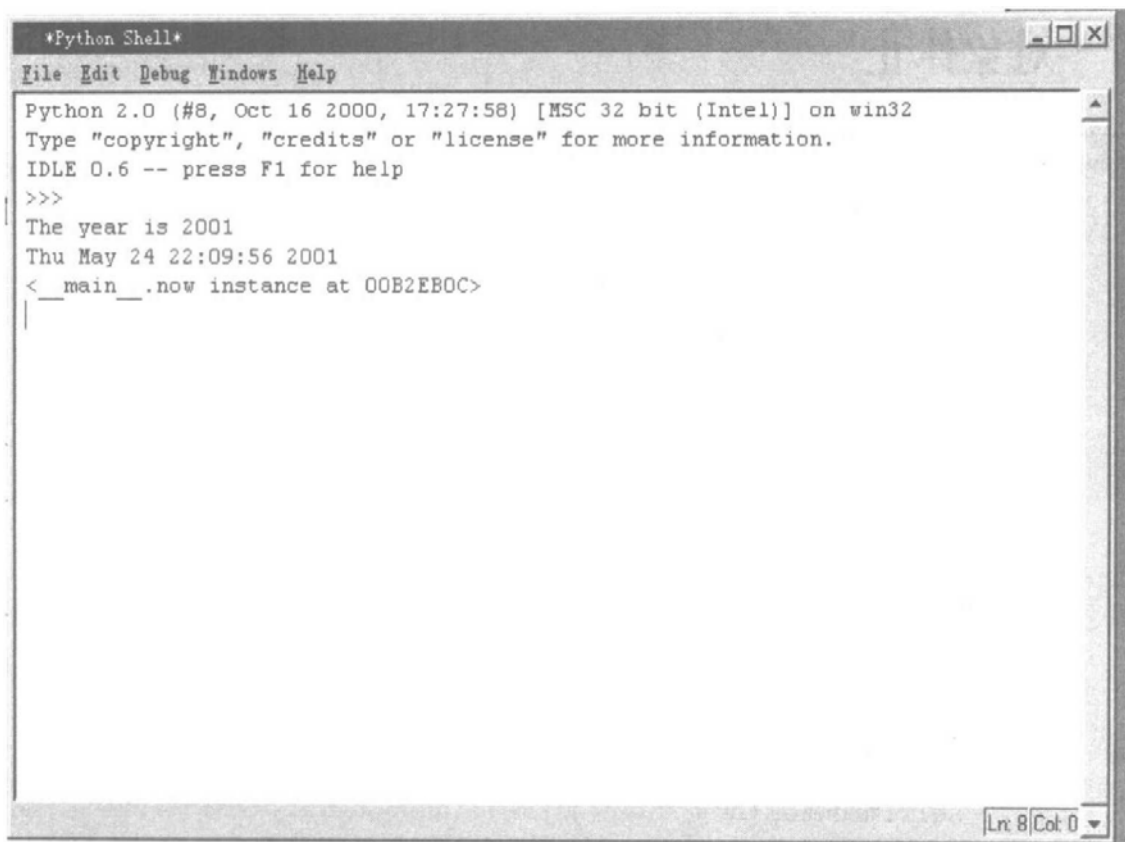
11.7 对象休止

例 1: now.py

```
#!/usr/local/bin/python
import time
class now:
    def _init_ (self) :
        self.t = time.time ()
        self.storetime ()
    def storetime (self) :
        self.year, \
        self.month, \
        self.day, \
        self.hour, \
        self.minute, \
        self.second, \
        self.dow, \
        self.doy, \
        self.dst = time.localtime (self.t)
    def _str_ (self) :
        return time.ctime (self.t)

n = now ()
print "The year is", n.year
print n
s='n'
print s
```

执行结果:

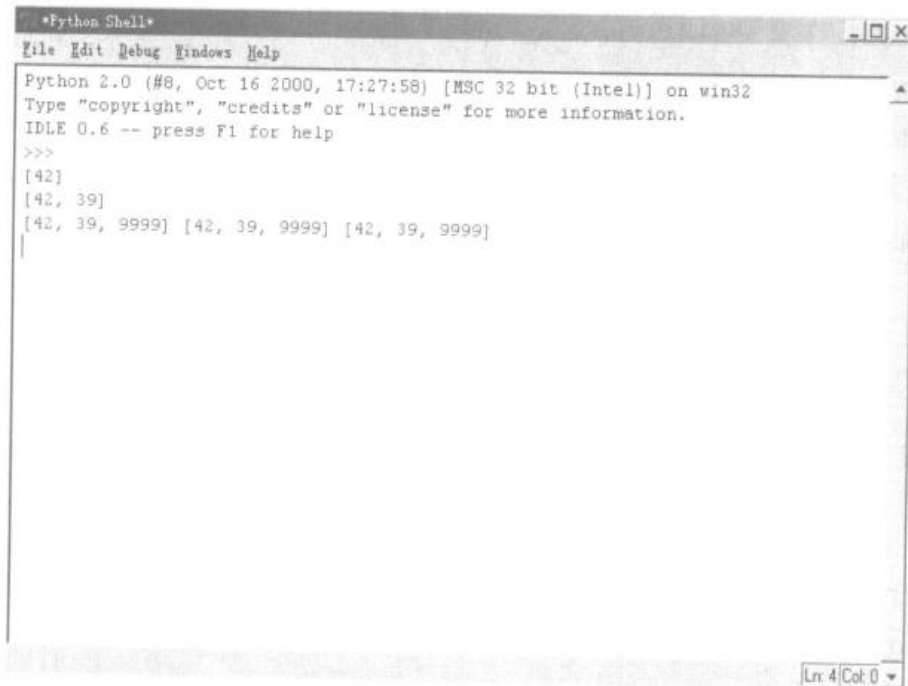
**例 2: spam1.py**

```
#!/usr/local/bin/python

def spam (n, l=[]):
    l.append (n)
    return l

x = spam (42)
print x
y = spam (39)
print y
z = spam (9999, y)
print x, y, z
```

执行结果:



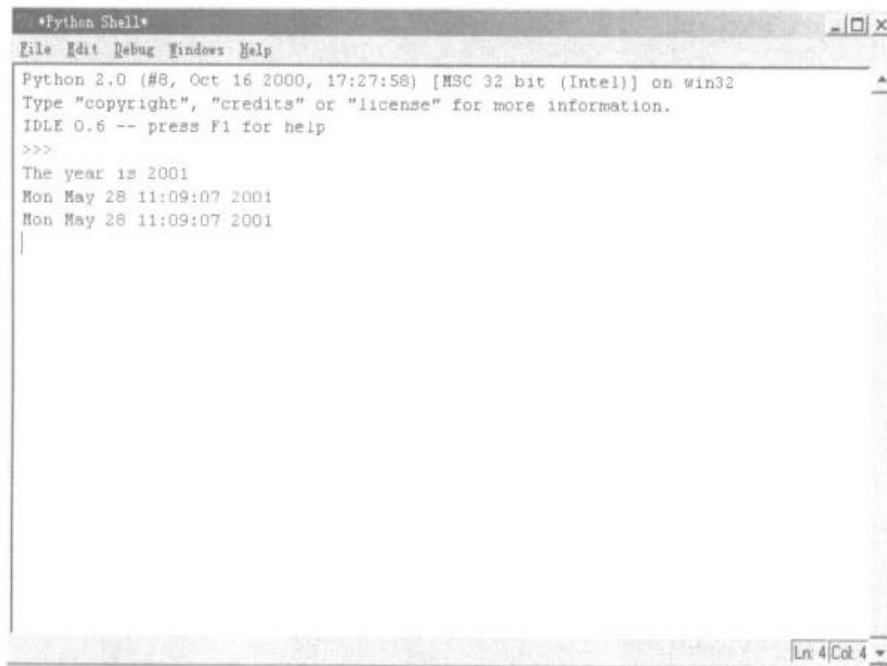
11.8 定义对象

例1: now.py

```
#!/usr/local/bin/python
import time
class now:
    def _init_(self) :
        self.t = time.time ()
        self.storetime ()
    def storetime (self) :
        self.year, \
        self.month, \
        self.day, \
        self.hour, \
        self.minute, \
        self.second, \
        self.dow, \
        self.doy, \
        self.dst = time.localtime (self.t)
```

```
def _str_ (self) :  
    return time.ctime (self.t)  
def _repr_ (self) :  
    return time.ctime (self.t)  
def _call_ (self, t=-1.0) :  
    if t < 0.0:  
        self.t = time.time ()  
    else:  
        self.t = t  
    self.storetime ()  
  
n = now ()  
print "The year is", n.year  
print n  
s=`n`  
print s
```

执行结果:



```
Python Shell  
File Edit Debug Windows Help  
Python 2.0 (#8, Oct 16 2000, 17:27:58) [MSC 32 bit (Intel)] on win32  
Type "copyright", "credits" or "license" for more information.  
IDLE 0.6 -- press F1 for help  
>>>  
The year is 2001  
Mon May 28 11:09:07 2001  
Mon May 28 11:09:07 2001  
Ln: 4/Cot 4
```

例 2: today.py

```
#!c:\python\python.exe
```

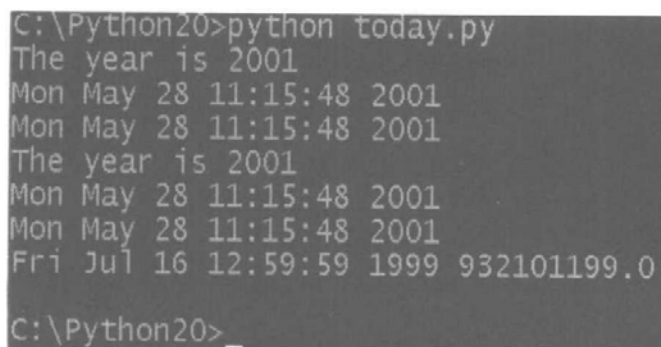
```
import time
import now

class today (now.now) :
    def _init_ (self, y = 1970) :
        now.now._init_ (self)
    def update (self, tt) :
        if len (tt) < 9 :
            raise TypeError
        if tt[0] < 1970 or tt[0] > 2038:
            raise OverflowError
        self.t = time.mktime (tt)
        self (self.t)

if _name_ == "_main_":
    n = today ()
    print "The year is", n.year
    print n
    x = today ()
    s = 'x'
    print s

    tt = (1999, 7, 16, 12, 59, 59, 0, 0, -1)
    x.update (tt)
    print x, x.t
```

执行结果:



```
C:\Python20>python today.py
The year is 2001
Mon May 28 11:15:48 2001
Mon May 28 11:15:48 2001
The year is 2001
Mon May 28 11:15:48 2001
Mon May 28 11:15:48 2001
Fri Jul 16 12:59:59 1999 932101199.0

C:\Python20>_
```


11.9 面向对象的概念

例 1: bunch.py

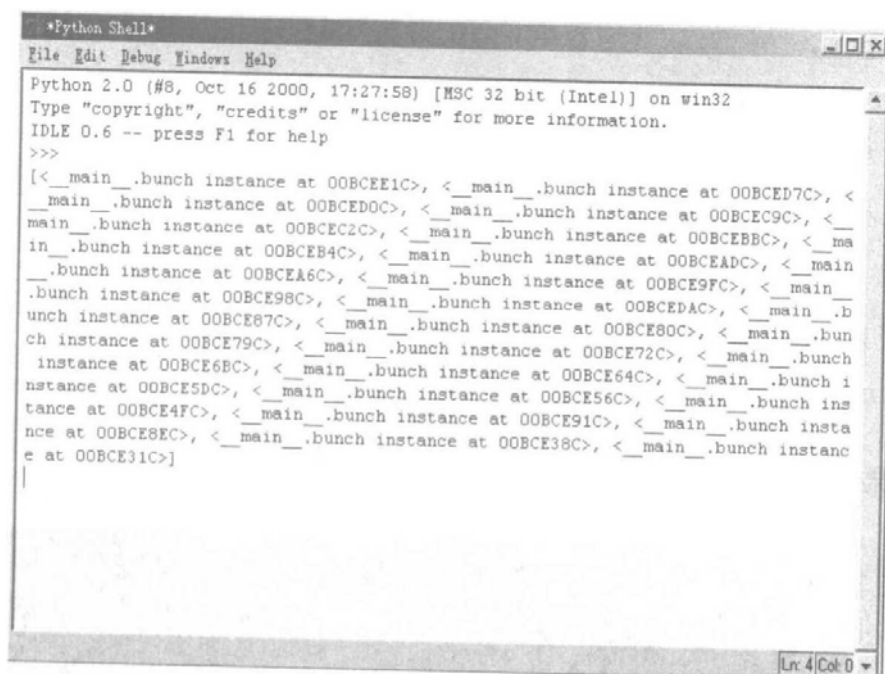
```
#!c:\python\python.exe

class bunch:
    def __init__(self):
        pass

if __name__ == "__main__":
    b = []
    for i in range (0, 25):
        b.append (bunch ())

print b
```

执行结果:



例 2: nesting.py

```
#!c:\python\python.exe

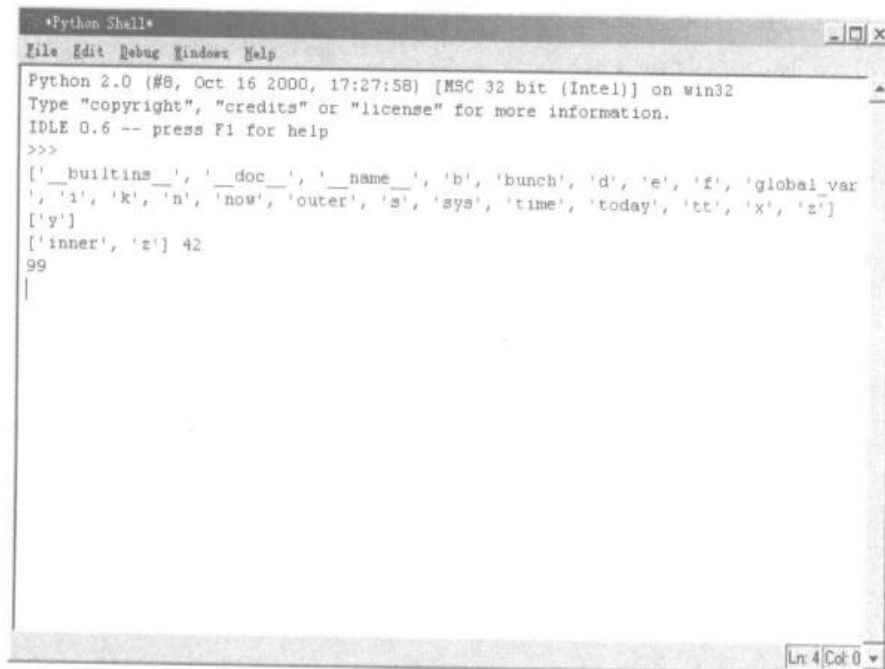
import sys
```

```
global_var = 9999

def outer () :
    z = 42
    def inner () :
        global z
        y = 666
        z = 99
        print dir ()
    inner ()
    print dir (), z

print dir ()
outer ()
print z
```

执行结果:



例3:

now.py:

```
#!/usr/local/bin/python
```

```
import time
class now:
    def __init__ (self) :
        self.t = time.time ()
        self.storetime ()
    def storetime (self) :
        self.year, \
        self.month, \
        self.day, \
        self.hour, \
        self.minute, \
        self.second, \
        self.dow, \
        self.doy, \
        self.dst = time.localtime (self.t)
    def __str__ (self) :
        return time.ctime (self.t)
    def __repr__ (self) :
        return time.ctime (self.t)
    def __call__ (self, t=-1.0) :
        if t < 0.0:
            self.t = time.time ()
        else:
            self.t = t
        self.storetime ()

n = now ()
print "The year is", n.year
print n
s=`n`
print s

namespace.py:
#!c:\python\python.exe
```

```
import sys
import now

z = 666

def f () :
    "doc string"
    z = 42
    print dir ()

k = sys.modules.keys ()
print "Keys:", k
print "-----"
for i in k:
    if i == "_main_":
        print ">>>", i, "_dict_", sys.modules[i]._dict_

print dir ()
print "-----"
print dir (f)
print "---", "f", dir (sys.modules["_main_"]._dict_["f"])

f ()

def d () :
    "another doc string"
    z = 44
    x = 9.999
    print dir ()

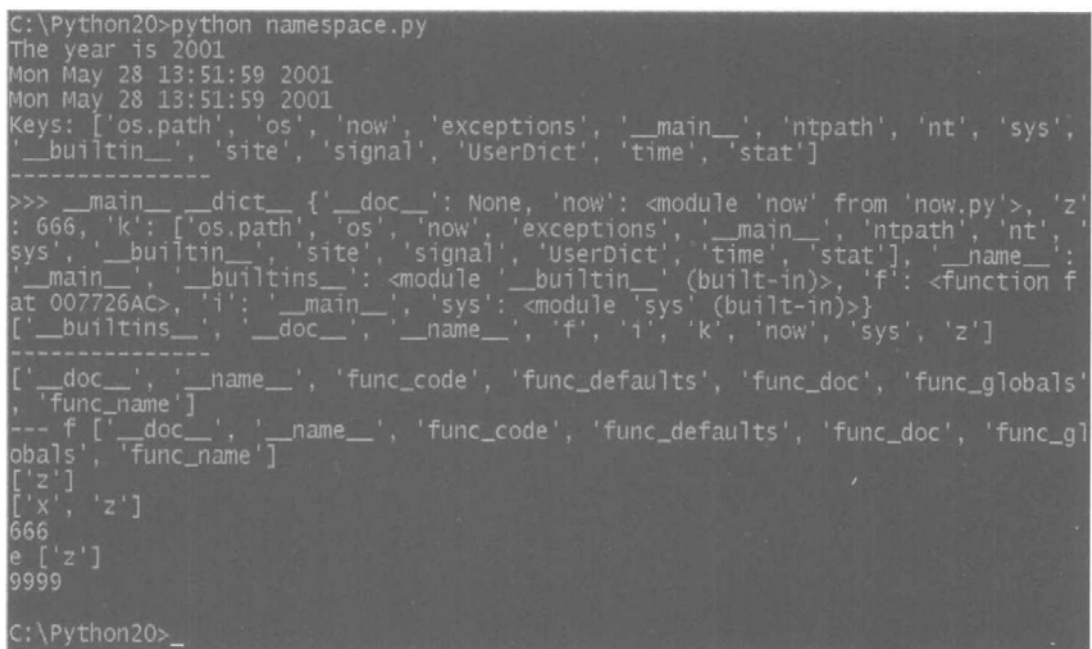
d ()

def e () :
    "yet another doc string"
    z = 22
    global z
    z = 9999
```

```
print "e". dir ()

print z
e ()
print z
```

执行结果:



```
C:\Python20>python namespace.py
The year is 2001
Mon May 28 13:51:59 2001
Mon May 28 13:51:59 2001
Keys: ['os.path', 'os', 'now', 'exceptions', '__main__', 'ntpath', 'nt', 'sys',
'__builtin__', 'site', 'signal', 'UserDict', 'time', 'stat']
-----
>>> __main__ __dict__ {'__doc__': None, 'now': <module 'now' from 'now.py'>, 'z'
: 666, 'k': ['os.path', 'os', 'now', 'exceptions', '__main__', 'ntpath', 'nt',
'sys', '__builtin__', 'site', 'signal', 'UserDict', 'time', 'stat'], '__name__':
'__main__', '__builtins__': <module '__builtin__' (built-in)>, 'f': <function f
at 007726AC>, 'i': '__main__', 'sys': <module 'sys' (built-in)>}]
['__builtins__', '__doc__', '__name__', 'f', 'i', 'k', 'now', 'sys', 'z']
-----
['__doc__', '__name__', 'func_code', 'func_defaults', 'func_doc', 'func_globals'
, 'func_name']
--- f ['__doc__', '__name__', 'func_code', 'func_defaults', 'func_doc', 'func_g
obals', 'func_name']
['z']
['x', 'z']
666
e ['z']
9999
C:\Python20>_
```

11.10 更多的面向对象的概念

例 1:

```
today.py:
#!c:\python\python.exe

import time
import now

class today (now.now) :
    def __init_ (self, y = 1970) :
        now.now.__init_ (self)
    def update (self, tt) :
        if len (tt) < 9 :
```

```
        raise TypeError
    if tt[0] < 1970 or tt[0] > 2038:
        raise OverflowError
    self.t = time.mktime (tt)
    self (self.t)

if __name__ == "__main__":
    n = today ()
    print "The year is", n.year
    print n
    x = today ()
    s = 'x'
    print s

    tt = (1999, 7, 16, 12, 59, 59, 0, 0, -1)
    x.update (tt)
print x, x.t

imptest1.py:
#!c:\python\python.exe

_CHANGEOVER="Sep 3 1752"

from today import *

imptest2.py:
#!c:\python\python.exe

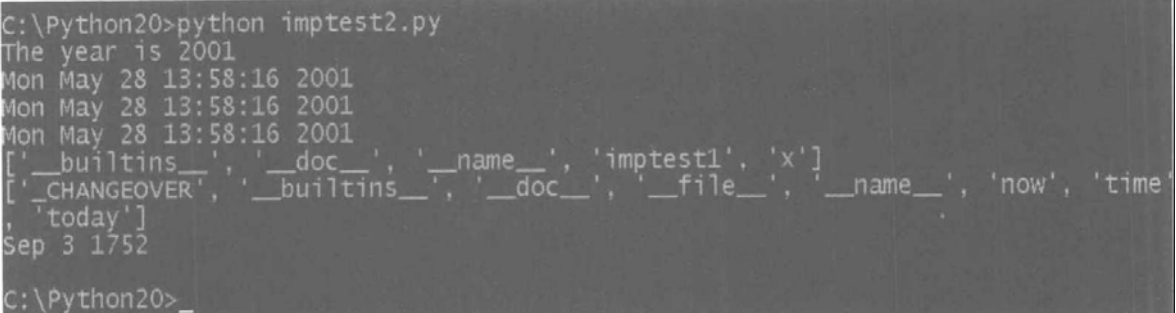
import imptest1

x = imptest1.today ()
print x
print dir ()
print dir (imptest1)

try:
    print _CHANGEOVER
except:
```

```
try:
    print imptest1._CHANGEOVER
except:
    print "Can't find _CHANGEOVER"
```

执行结果:



```
C:\Python20>python imptest2.py
The year is 2001
Mon May 28 13:58:16 2001
Mon May 28 13:58:16 2001
Mon May 28 13:58:16 2001
['_builtins_', '_doc_', '_name_', 'imptest1', 'x']
['_CHANGEOVER', '_builtins_', '_doc_', '_file_', '_name_', 'now', 'time',
'today']
Sep 3 1752
C:\Python20>_
```

例 2:

now.py:

```
#!/usr/local/bin/python
import time
class now:
    def __init__(self):
        self.t = time.time()
        self.storetime()
    def storetime(self):
        self.year, \
        self.month, \
        self.day, \
        self.hour, \
        self.minute, \
        self.second, \
        self.dow, \
        self.doy, \
        self.dst = time.localtime(self.t)
    def __str__(self):
```

```
        return time.ctime (self.t)
    def _repr_ (self) :
        return time.ctime (self.t)
    def _call_ (self t=-1.0) :
        if t < 0.0:
            self.t = time.time ()
        else:
            self.t = t
        self.storeTime ()

n = now ()
print "The year is"  n.year
print n
s=`n`
print s

roman.py:
#!c:\python\python.exe

import string
import sys

class roman:
    def _init_ (self, y) :
        if y < 1:
            raise ValueError
        self.rlist = []
        ms = y / 1000
        tmp = y % 1000
        if ms > 0:
            self.rlist.append ("M" * ms)

        ds = tmp / 500
        tmp = tmp % 500
        if ds > 0:
            self.rlist.append ("D" * ds)
```



```
        cs = tmp / 100
        tmp = tmp % 100
        if cs > 0:
            self.rlist.append("C" * cs)

        ls = tmp / 50
        tmp = tmp % 50
        if ls > 0:
            self.rlist.append("L" * ls)

        xs = tmp / 10
        tmp = tmp % 10
        if xs > 0:
            self.rlist.append("X" * xs)

        vs = tmp / 5
        tmp = tmp % 5
        if vs > 0:
            self.rlist.append("V" * vs)

        js = tmp
        if js > 0:
            self.rlist.append("I" * js)

    def ryear (self) :
        s = ""
        for i in self.rlist:
            s = s + i
        return s

    def _repr_ (self) :
        return (self.ryear ())

if __name__ == "__main__":
    if len (sys.argv) > 1:
        yr = string.atoi (sys.argv[1])
```

```
        else:
            yr = 1999
        x = roman (yr)
    print x.ryear ()

today-roman.py:
#!c:\python\python.exe

#!/usr/local/bin/python

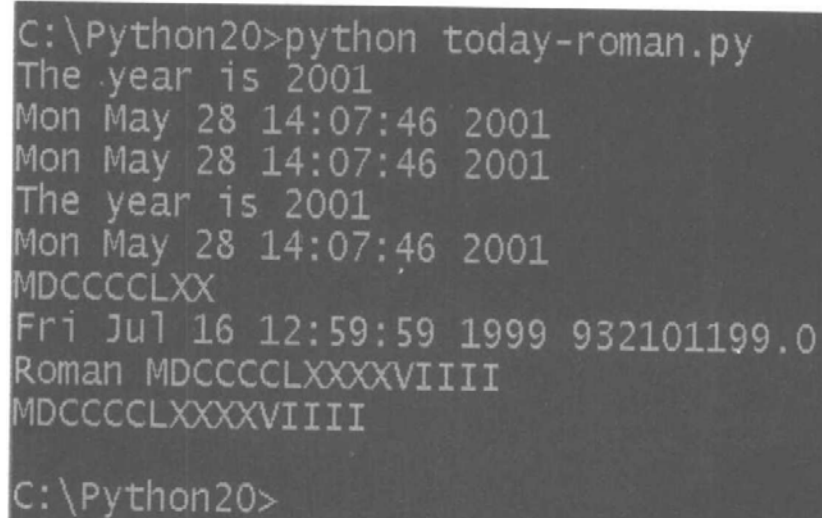
import time
import now
import roman

#class today (now.now, roman.roman) :
class today (roman.roman, now.now) :
    def __init__ (self, y = 1970) :
        now.now.__init__ (self)
        roman.roman.__init__ (self, y)
    def update (self, tt) :
        if len (tt) < 9 :
            raise TypeError
        if tt[0] < 1970 or tt[0] > 2038:
            raise OverflowError
        self.t = time.mktime (tt)
        self (self.t)
        roman.roman.__init__ (self, self.year)

if __name__ == "__main__":
    n = today ()
    print "The year is", n.year
    print n
    x = today ()
    s = `x`
    print s
```

```
tt = (1999, 7, 16, 12, 59, 59, 0, 0, -1)
x.update (tt)
print x, x.t
print "Roman", x.ryear ()
st = `x`
print st
```

执行结果:



```
C:\Python20>python today-roman.py
The year is 2001
Mon May 28 14:07:46 2001
Mon May 28 14:07:46 2001
The year is 2001
Mon May 28 14:07:46 2001
MDCCCCLXX
Fri Jul 16 12:59:59 1999 932101199.0
Roman MDCCCCLXXXVIII
MDCCCCLXXXVIII
C:\Python20>
```

11.11 特殊类程序

例 1: del.py

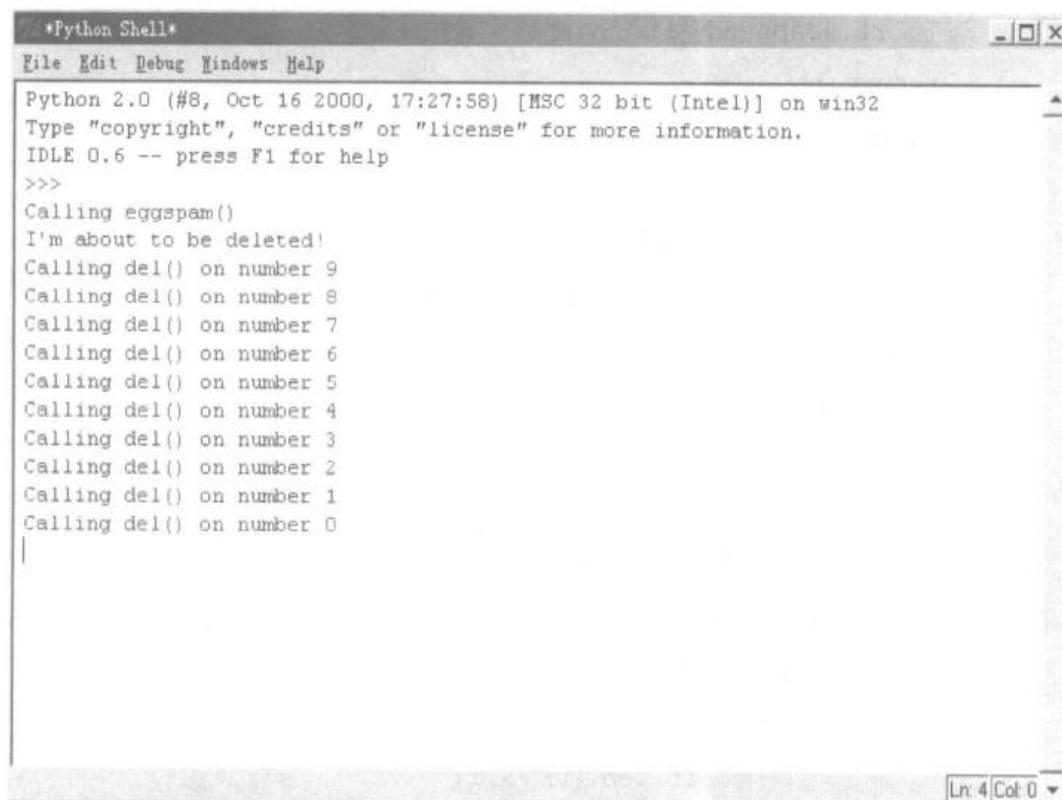
```
#!c:\python\python.exe
class spam:
    def _init_ (self) :
        pass
    def _del_ (self) :
        print "I'm about to be deleted!"

a = spam ()

def eggspam () :
    z = spam ()
```

```
if __name__ == "__main__":  
    print "Calling eggspam () "  
    b = eggspam ()  
    t = []  
    for i in range (10) :  
        t.append (a)  
  
    for i in range (9, -1, -1) :  
        print "Calling del () on number", i  
        del (t[i])  
  
x = spam ()
```

执行结果:



```
*Python Shell*  
File Edit Debug Windows Help  
Python 2.0 (#8, Oct 16 2000, 17:27:58) [MSC 32 bit (Intel)] on win32  
Type "copyright", "credits" or "license" for more information.  
IDLE 0.6 -- press F1 for help  
>>>  
Calling eggspam()  
I'm about to be deleted!  
Calling del() on number 9  
Calling del() on number 8  
Calling del() on number 7  
Calling del() on number 6  
Calling del() on number 5  
Calling del() on number 4  
Calling del() on number 3  
Calling del() on number 2  
Calling del() on number 1  
Calling del() on number 0  
|  
Ln: 4 Col: 0
```

例 2: parrots.py

```
#!c:\python\python.exe
```

```
import string
```

```
import sys
```

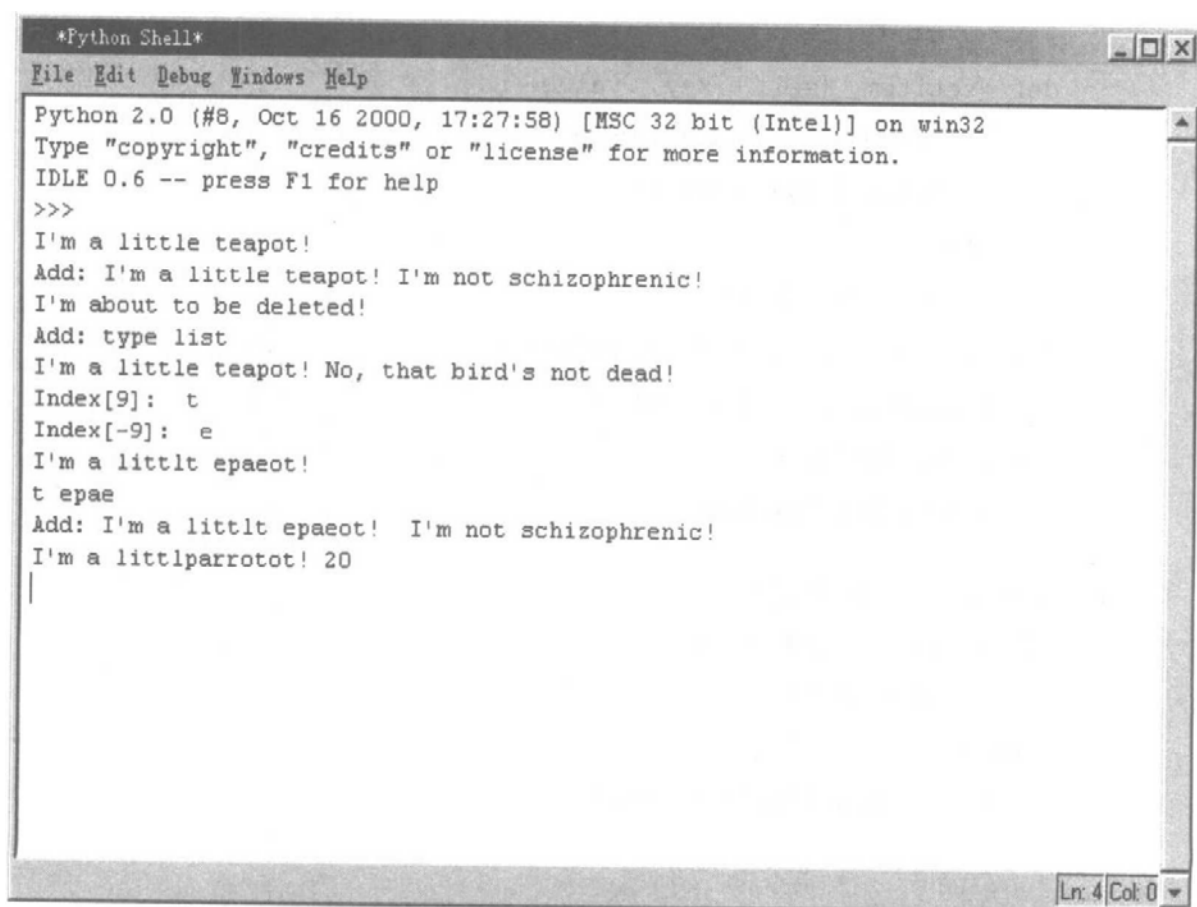
```
class parrot:
    def _init_ (self, s="") :
        self.s = list (s)
    def _repr_ (self) :
        t = ""
        for i in self.s:
            t = t + i
        return t
    def _str_ (self) :
        return parrot._repr_ (self)
    def _add_ (self, other) :
        rt = parrot ('self')
        for i in other.s:
            rt.s.append (i)
        return rt
    def _radd_ (self, other) :
        return self + other
    def _coerce_ (self, other) :
        if type (other) == type (self) :
            return (self, other)
        elif type (other) == type ("") :
            rt = parrot (other)
            return (self, rt)
        elif type (other) == type ([]) :
            print "type list"
            rt = parrot ()
            for i in other:
                if len (i) > 1:
                    for j in i:
                        rt.s.append (j)
                else:
                    rt.s.append (i)
            return (self, rt)
```

```
        elif type (other) == type ({}):
            return None
        else:
            rt = parrot (str (other))
            return (self, rt)
def _getitem_ (self, key):
    if type (key) == type (0):
        return self.s[key]
    else:
        raise TypeError
def _getslice_ (self, i, j):
    s = self.s[i:j]
    t = ""
    for k in s:
        t = t + k
    return t
def _setitem_ (self, key, value):
    if type (key) == type (0):
        self.s[key] = value
    else:
        raise TypeError
def _setslice_ (self, i, j, value):
    self.s[i:j] = list (value)
def _len_ (self):
    return len (self.s)

if __name__ == "__main__":
    if len (sys.argv) > 1:
        s = sys.argv[1]
    else:
        s = "I'm a little teapot!"
    p = parrot (s)
    print p
    q = parrot (" I'm not schizophrenic!")
```

```
print "Add:", p + q
x = ['N', 'o', ' ', 'that bird's not dead', '!']
print "Add:", p + ' ' + x
print "Index[9]: ", p[9]
print "Index[-9]: ", p[-9]
p[11], p[13] = p[13], p[11]
p[14], p[16] = p[16], p[14]
print p
pa = p[11:17]
print pa
print "Add:", p + ' ' + q
p[11:17] = "parrot"
print p, len (p)
```

执行结果:



```
*Python Shell*
File Edit Debug Windows Help
Python 2.0 (#8, Oct 16 2000, 17:27:58) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.6 -- press F1 for help
>>>
I'm a little teapot!
Add: I'm a little teapot! I'm not schizophrenic!
I'm about to be deleted!
Add: type list
I'm a little teapot! No, that bird's not dead!
Index[9]:  t
Index[-9]:  e
I'm a littlt epaeot!
t epae
Add: I'm a littlt epaeot!  I'm not schizophrenic!
I'm a littlparrotot! 20
|
```

11.12 Python GUI 编程简介

例 1: ave.py

```
#!/usr/local/bin/python

import sys
from Tkinter import *

def die (event) :
    sys.exit (0)

root = Tk ()
button = Button (root)
button["text"] = "测试"
button.bind("<Button-1>", die)
button.pack ()
root.mainloop ()
```

执行结果:



11.13 TK小部件

例 1: tkoptionmenu.py

```
#!/usr/local/bin/python

from Tkinter import *
import tkMessageBox
import sys

def die () :
    global xx
```



```
    print xx.get ()
    sys.exit (0)

def callee () :
    print "I was called, few are chosen"

def about () :
    tkMessageBox.showinfo ("tkmenu", "This is tkmenu.py Version 0")

root = Tk ()
bar = Menu (root)

filem = Menu (bar)
filem.add_command (label="Open...", command=callee)
filem.add_command (label="New...", command=callee)
filem.add_command (label="Save", command=callee)
filem.add_command (label="Save as...", command=callee)
filem.add_separator ()
filem.add_command (label="Exit", command=die)

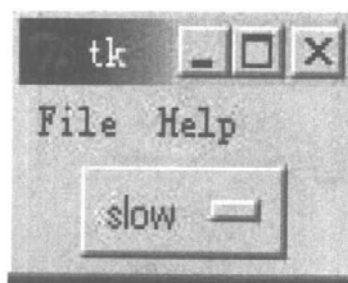
helpm = Menu (bar)
helpm.add_command (label="Index...", command=callee)
helpm.add_separator ()
helpm.add_command (label="About", command=about)

bar.add_cascade (label="File", menu=filem)
bar.add_cascade (label="Help", menu=helpm)

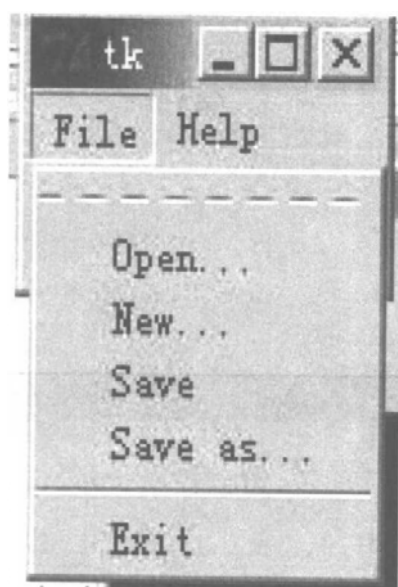
root.config (menu=bar)
frame = Frame (root)
frame.pack ()
xx = StringVar (frame)
xx.set ("slow")

fm = OptionMenu (frame, xx, "slow", "slower", "slowest", "even slower")
fm.pack ()
root.mainloop ()
```

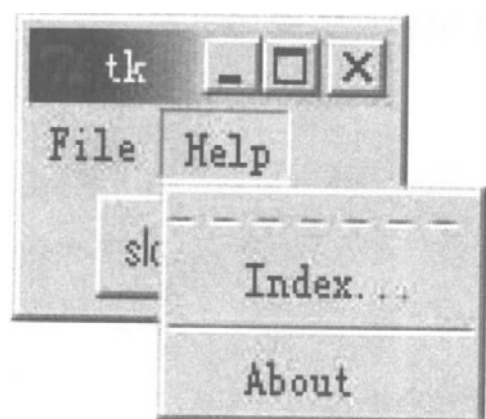
执行结果截图:



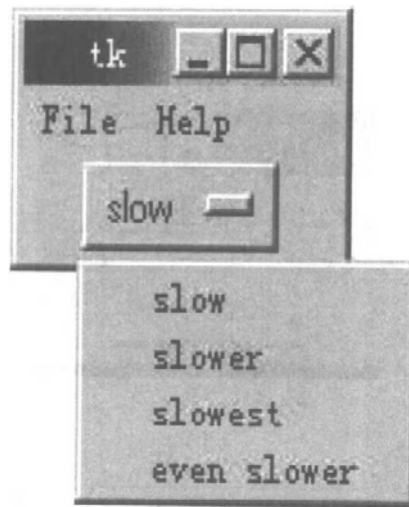
(1)



(2)



(3)



(4)

11.14 TK 部件2

这是一个小小的编辑器。

例 1: multi-editor.py

```
#!/usr/local/bin/python

from Tkinter import *
from ScrolledText import *
import tkMessageBox
from tkFileDialog import *
import fileinput

t1 = []
root = None

def die () :
    sys.exit (0)

def about () :
    tkMessageBox.showinfo ("Tkeditor", "简单编辑器 版本1.0\n"
        "写于 2001年\n"
        "作者: XXXX")
```

```
class editor:
    def _init_ (self, rt) :
        if rt == None:
            self.t = Tk ()
        else:
            self.t = Toplevel (rt)
        self.t.title ("Tkeditor %d" % len (t1))
        self.bar = Menu (rt)

        self.filem = Menu (self.bar)
        self.filem.add_command (label="Open...", command=self.openfile)
        self.filem.add_command (label="New...", command=neweditor)
        self.filem.add_command (label="Save as...", command=self.savefile)
        self.filem.add_command (label="Close", command=self.close)
        self.filem.add_separator ()
        self.filem.add_command (label="Exit", command=die)

        self.helpm = Menu (self.bar)
        self.helpm.add_command (label="About", command=about)

        self.bar.add_cascade (label="File", menu=self.filem)
        self.bar.add_cascade (label="Help", menu=self.helpm)
        self.t.config (menu=self.bar)

        self.f = Frame (self.t, width=512)
        self.f.pack (expand=1, fill=BOTH)

        self.st = ScrolledText (self.f, background="white")
        self.st.pack (side=LEFT, fill=BOTH, expand=1)

    def close (self) :
        self.t.destroy ()

    def openfile (self) :
        pl = END
        oname = askopenfilename (filetypes= ( ("Python files", "*.py") ))
        if oname:
```

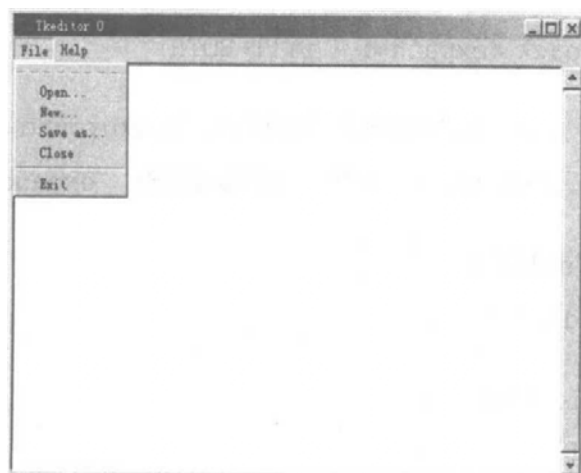
```
        for line in fileinput.input (oname) :
            self.st.insert (pl, line)
        self.t.title (oname)

    def savefile (self) :
        sname = asksaveasfilename ()
        if sname:
            ofp = open (sname, "w")
            ofp.write (self.st.get (1.0, END))
            ofp.flush ()
            ofp.close ()
            self.t.title (sname)

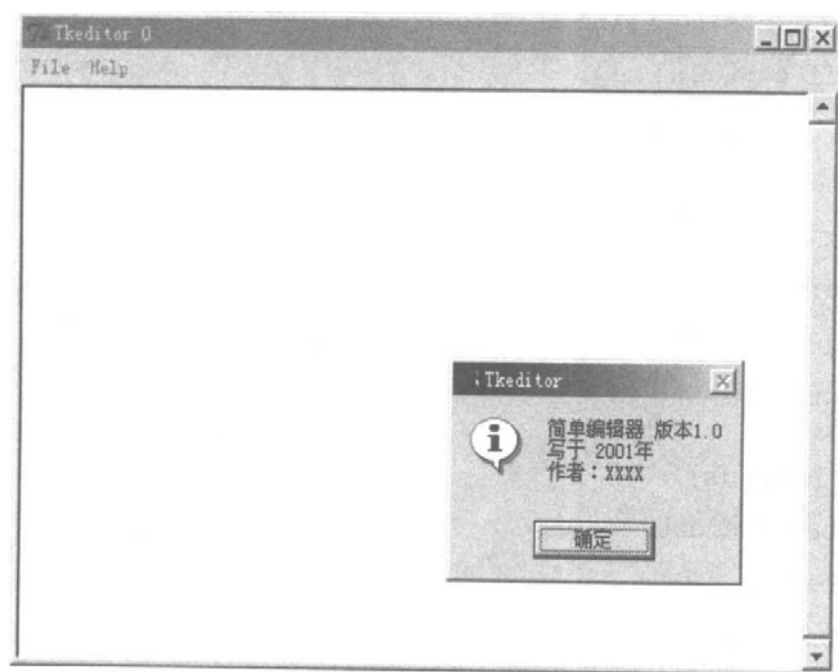
    def neweditor () :
        global root
        tl.append (editor (root))

if __name__ == "__main__":
    root = None
    tl.append (editor (root))
    root = tl[0].t
    root.mainloop ()
```

执行结果截图:



(1)



(2)

11.15 TK 图形

例1: 这是一个查看图片的程序, 在菜单中列出目录下的所有图片, 选择后在窗口中显示所选择的图片。

```
tkcascadingmenu.py:
#!/usr/local/bin/python

from Tkinter import *
import os
import string

img = None

def die():
    sys.exit(0)

def listgifs(d=".") :
    l = os.listdir(d)
    rl = []
    for i in l:
        t = string.lower(i)
```

```
        g = string.rfind (t, ".gif")
        if g >= 0:
            rl.append (i)
    if len (rl) <1:
        rl = None
    else:
        rl.sort ()
    return rl

def setimage (s) :
    global elements
    global lb
    global img
    if s in elements:
        img = PhotoImage (file=s)
        lb["image"] = img

def main () :
    global elements
    global lb
    elements = listgifs ()
    if not elements:
        print "No gifs"
    n = len (elements)
    nm = n / 10
    no = n % 10
    if no:
        nm = nm + 1
    print "For %d files, I'll make %d menus" % ( n, nm )
    root = Tk ()
    mb = Menu (root)
    cb = Menu (mb)
    cb.add_command (label="Exit", command=die)

    gm = Menu (mb)
```

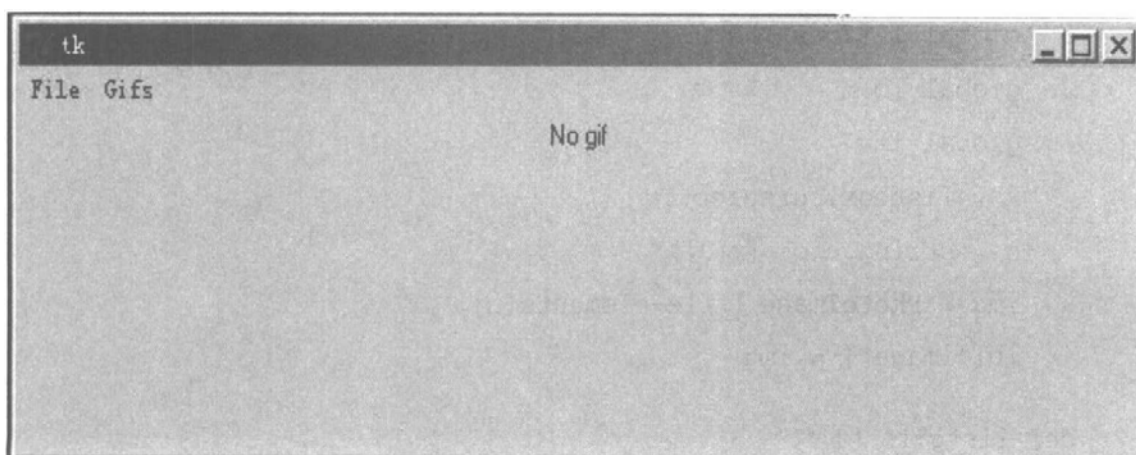
```
for i in range (nm) :
    tm = Menu (gm)
    if i == nm - 1 and no != 0:
        lim = no
    else:
        lim = 10
    for j in range (lim) :
        ne = (10 * i) + j
        tm.add_command (label=elements[ne],
                        command=lambda m=elements[ne]:setimage (m))
    gm.add_cascade (label="List gifs %d" % (i), menu=tm)

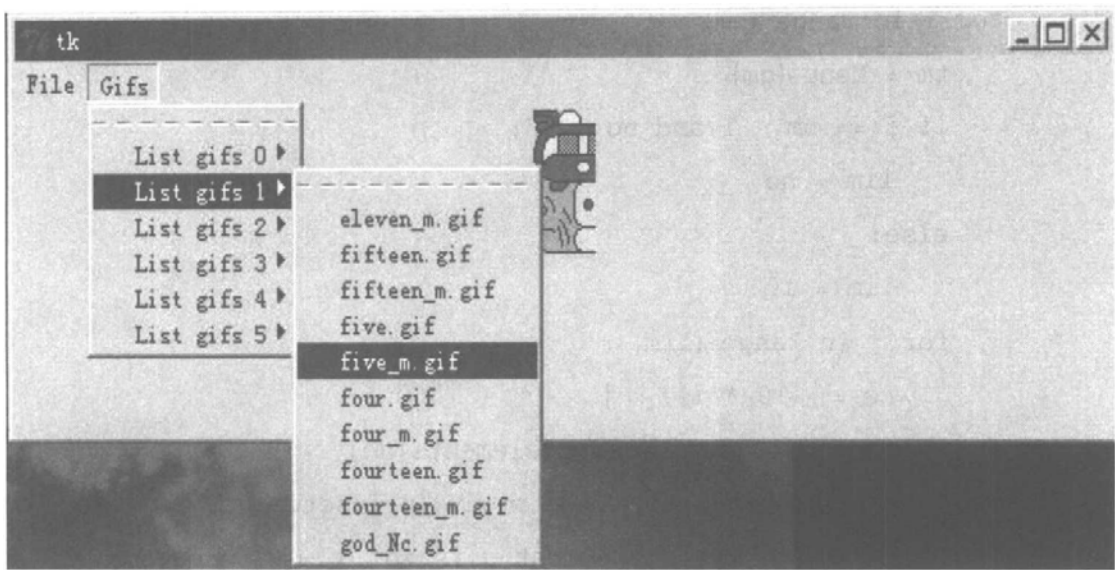
mb.add_cascade (label="File", menu=cb)
mb.add_cascade (label="Gifs", menu=gm)

lb = Label (root, text="No gif")
lb.pack ()
root.config (menu=mb)
root.mainloop ()

if __name__ == "__main__":
    main ()
```

执行结果截图：





(2)

例 2:

```
tkscrolledlistbox.py:
#!/usr/local/bin/python

from Tkinter import *
import sys
import os
import string

def setn (event) :
    global elements
    global listbox
    global lb
    global img
    x = listbox.curselection ()
    n = string.atoi (x[0])
    img = PhotoImage (file=elements[n])
    lb["image"] = img

def listgifs (d=".") :
    l = os.listdir (d)
    rl = []
```

```
    for i in l:
        t = string.lower (i)
        g = string.rfind (t, ".gif")
        if g >= 0:
            rl.append (i)
    if len (rl) <1:
        rl = None
    else:
        rl.sort ()
    return rl

def die (event) :
    sys.exit (0)

root = Tk ()
button = Button (root)
button["text"] = "Quit"
button.bind ("<Button>", die)
button.pack ()
labelx = Label (root)
labelx["height"] = 1
labelx.pack ()

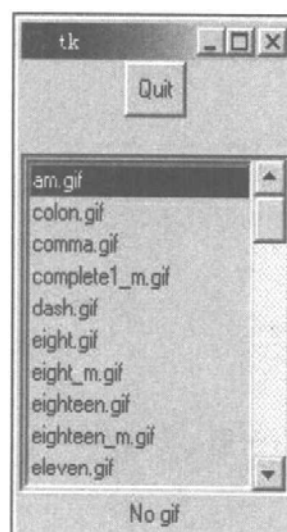
elements = listgifs ()

frame = Frame (root, bd=2, relief=SUNKEN)
frame.pack (expand=1, fill=BOTH)
scrollbar = Scrollbar (frame, orient=VERTICAL)
listbox = Listbox (frame, exportselection=0, height=10,
    yscrollcommand=scrollbar.set)
listbox.bind ("<Double-Button-1>", setn)
for i in elements :
    listbox.insert (END, i)
scrollbar.config (command=listbox.yview)
listbox.pack (side=LEFT)
```

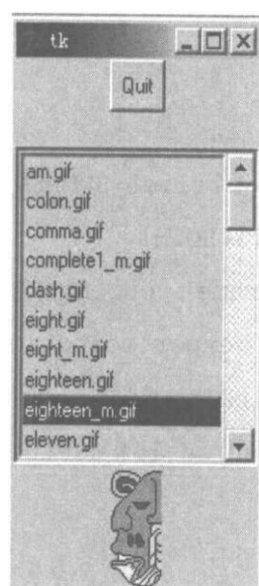
```
scrollbar.pack (side=LEFT, fill=Y)
listbox.select_set (0)
listbox.see (0)
lb = Label (root, text="No gif")
lb.pack ()

root.mainloop ()
```

执行结果截图:



(1)



(2)

11.16 TK图形2

例1: ccube.py

```
#!/usr/local/bin/python

from Tkinter import *
from Canvas import Rectangle, Polygon, Window, CanvasText

def GreenCyan () :
    global bll, cv
    brl = [0, 0xFF, 0]
    tmp = bll[:]
    for i in range (ncolors) :
        cm = "#%02X%02X%02X" % (brl[0], brl[1], brl[2])
        Rectangle (cv, tmp[0], tmp[1], tmp[0]+dg, tmp[1]+4,
                    fill=cm, outline="", tag="colorcube")
        tmp[0] = tmp[0] + dg
        brl[2] = (brl[2] + hr) % 0x100

def GreenYellow () :
    global bll, cv
    brl = [0, 0xFF, 0]
    tmp = bll[:]
    for i in range (ncolors) :
        cm = "#%02X%02X%02X" % (brl[0], brl[1], brl[2])
        Rectangle (cv, tmp[0], tmp[1], tmp[0]+4, tmp[1]-dg,
                    fill=cm, outline="", tag="colorcube")
        tmp[1] = tmp[1] - dg
        brl[0] = (brl[0] + hr) % 0x100

def YellowWhite () :
    global bul, cv
    brl = [0xFF, 0xFF, 0]
    tmp = bul[:]
```

```
    for i in range (ncolors) :
        cm = "%02X%02X%02X" % (brl[0], brl[1], brl[2])
        Rectangle (cv, tmp[0], tmp[1], tmp[0]+dg, tmp[1]+4,
            fill=cm, outline="", tag="colorcube")
        tmp[0] = tmp[0] + dg
        brl[2] = (brl[2] + hr) % 0x100

def WhiteCyan () :
    global bur, cv
    brl = [0xFF, 0xFF, 0xFF]
    tmp = bur[:]
    for i in range (ncolors) :
        cm = "%02X%02X%02X" % (brl[0], brl[1], brl[2])
        Rectangle (cv, tmp[0], tmp[1], tmp[0]+4, tmp[1]+dg,
            fill=cm, outline="", tag="colorcube")
        tmp[1] = tmp[1] + dg
        brl[0] = (brl[0] - hr) % 0x100

def BlackBlue () :
    global fl1, cv
    brl = [0, 0, 0]
    tmp = fl1[:]
    for i in range (ncolors) :
        cm = "%02X%02X%02X" % (brl[0], brl[1], brl[2])
        Rectangle (cv, tmp[0], tmp[1], tmp[0]+dg, tmp[1]+4,
            fill=cm, outline="", tag="colorcube")
        tmp[0] = tmp[0] + dg
        brl[2] = (brl[2] + hr) % 0x100

def BlackRed () :
    global fl1, cv
    brl = [0, 0, 0]
    tmp = fl1[:]
    for i in range (ncolors) :
        cm = "%02X%02X%02X" % (brl[0], brl[1], brl[2])
```

```

        Rectangle (cv, tmp[0], tmp[1], tmp[0]+4, tmp[1]-dg,
                    fill=cm, outline="", tag="colorcube")
        tmp[1] = tmp[1] - dg
        brl[0] = (brl[0] + hr) % 0x100

def RedMagenta () :
    global ful, cv
    brl = [0xFF, 0, 0]
    tmp = ful[:]
    for i in range (ncolors) :
        cm = "%02X%02X%02X" % (brl[0], brl[1], brl[2])
        Rectangle (cv, tmp[0], tmp[1], tmp[0]+dg, tmp[1]+4,
                    fill=cm, outline="", tag="colorcube")
        tmp[0] = tmp[0] + dg
        brl[2] = (brl[2] + hr) % 0x100

def MagentaBlue () :
    global fur, cv
    brl = [0xFF, 0, 0xFF]
    tmp = fur[:]
    for i in range (ncolors) :
        cm = "%02X%02X%02X" % (brl[0], brl[1], brl[2])
        Rectangle (cv, tmp[0], tmp[1], tmp[0]+4, tmp[1]+dg,
                    fill=cm, outline="", tag="colorcube")
        tmp[1] = tmp[1] + dg
        brl[0] = (brl[0] - hr) % 0x100

def BlackGreen () :
    global fl1, cv
    brl = [0, 0, 0]
    tmp = fl1[:]
    eg = dg / 2
    for i in range (ncolors) :
        cm = "%02X%02X%02X" % (brl[0], brl[1], brl[2])
        Polygon (cv,

```

```
        tmp[0], tmp[1],
        tmp[0]+eg, tmp[1]-eg,
        tmp[0]+eg+3, tmp[1]-eg+3,
        tmp[0]+3, tmp[1]+3,
        fill=cm, outline="", tag="colorcube")
    tmp[0] = tmp[0] + eg
    tmp[1] = tmp[1] - eg
    brl[1] = (brl[1] + hr) % 0x100

def BlueCyan () :
    global flr, cv
    brl = [0, 0, 0xFF]
    eg = dg / 2
    tmp = flr[:]
    for i in range (ncolors) :
        cm = "%02X%02X%02X" % (brl[0], brl[1], brl[2])
        Polygon (cv,
            tmp[0], tmp[1],
            tmp[0]+eg, tmp[1]-eg,
            tmp[0]+eg+3, tmp[1]-eg+3,
            tmp[0]+3, tmp[1]+3,
            fill=cm, outline="", tag="colorcube")
        tmp[0] = tmp[0] + eg
        tmp[1] = tmp[1] - eg
        brl[1] = (brl[1] + hr) % 0x100

def RedYellow () :
    global ful, cv
    tmp = ful[:]
    brl = [0xFF, 0, 0]
    eg = dg / 2
    for i in range (ncolors) :
        cm = "%02X%02X%02X" % (brl[0], brl[1], brl[2])
        Polygon (cv,
```

```

        tmp[0], tmp[1],
        tmp[0]+eg, tmp[1]-eg,
        tmp[0]+eg+3, tmp[1]-eg+3,
        tmp[0]+3, tmp[1]+3,
        fill=cm, outline="", tag="colorcube")
    tmp[0] = tmp[0] + eg
    tmp[1] = tmp[1] - eg
    brl[1] = (brl[1] + hr) % 0x100

def MagentaWhite () :
    global fur, cv
    tmp = fur[:]
    brl = [0xFF, 0, 0xFF]
    eg = dg / 2
    for i in range (ncolors) :
        cm = "%02X%02X%02X" % (brl[0], brl[1], brl[2])
        Polygon (cv,
            tmp[0], tmp[1],
            tmp[0]+eg, tmp[1]-eg,
            tmp[0]+eg+3, tmp[1]-eg+3,
            tmp[0]+3, tmp[1]+3,
            fill=cm, outline="", tag="colorcube")
        tmp[0] = tmp[0] + eg
        tmp[1] = tmp[1] - eg
        brl[1] = (brl[1] + hr) % 0x100

def LabelVertices () :
    global cv, fl1, flr, blr, bl1, ful, fur, bul, bur
    CanvasText (cv, fl1[0] - 17, fl1[1], text="Black", tag="colorcube")
    CanvasText (cv, flr[0] + 25, flr[1], text="Blue", tag="colorcube")
    CanvasText (cv, blr[0] + 20, blr[1], text="Cyan", tag="colorcube")
    CanvasText (cv, bl1[0] - 30, bl1[1], text="Green", tag="colorcube")
    CanvasText (cv, ful[0] - 15, ful[1], text="Red", tag="colorcube")
    CanvasText (cv, fur[0] + 40, fur[1], text="Magenta", tag="colorcube")

```



```
    CanvasText (cv, bul[0] - 25, bul[1], text="Yellow", tag="colorcube")
    CanvasText (cv, bur[0] + 25, bur[1], text="White", tag="colorcube")

def ColorCube () :
    # Back rectangle first:
    GreenCyan ()
    GreenYellow ()
    YellowWhite ()
    WhiteCyan ()
    # Diagonals:
    RedYellow ()
    BlackGreen ()
    MagentaWhite ()
    BlueCyan ()
    # Front rectangle:
    BlackBlue ()
    BlackRed ()
    RedMagenta ()
    MagentaBlue ()
    LabelVertices ()

if __name__ == "__main__":
    # Vertices of the rgb cube
    # fl1 = front lower left,
    # bur = back upper right, etc.
    # The various functions draw
    # the necessary edges.
    fl1 = [32, 468]
    flr = [286, 468]
    ful = [32, 212]
    fur = [286, 212]
    bl1 = [160, 340]
    blr = [412, 340]
    bul = [160, 84]
```

```
bur = [412, 84]

def die (event=0) :
    sys.exit (0)

root = Tk ()

va = root.winfo_depth ()
if va < 16:
    ncolors = 16
else:
    ncolors = 32

hw = 256

dg = hw/ncolors
hr = 0x100 / ncolors

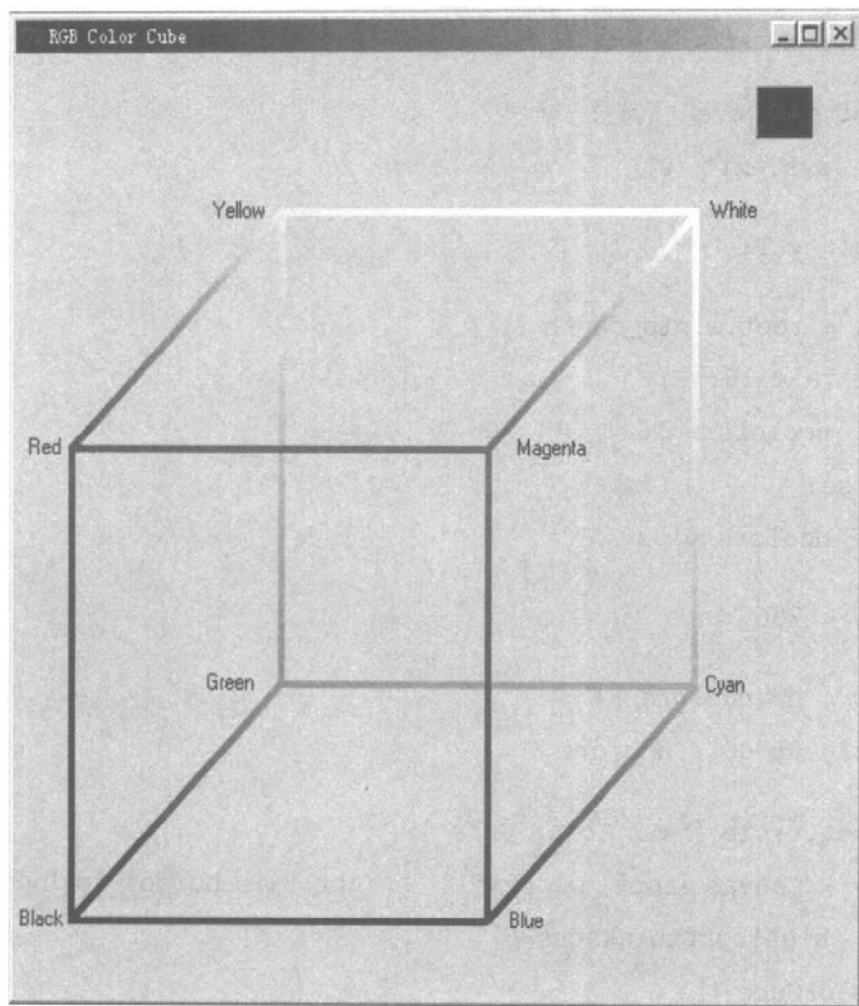
root.title ("RGB Color Cube")
cv = Canvas (root, width=512, height=512, borderwidth=0,
    highlightthickness=0)
ColorCube ()

button = Button (cv, text="Quit", background="black",
    foreground="red", command=die)
Window (cv, 468, 32, window=button)

cv.pack ()

root.mainloop ()
```

执行结果:



11.17 TK 图形3

例 1: colormap.py

```
#!/usr/local/bin/python

import sys
import string

from hls2rgb import *

def SetupColormap0 (ncolors) :
    cmap = []
    if ncolors < 3:
        cmap.append ("#FFFFFF")
```

```
        cmap.append("#000000")
        return cmap
    for i in range(0, ncolors):
        dd = (i * 359.0) / float(ncolors)
        r, g, b = hls2rgb(dd, 0.5, 0.9)
        r = r * 0xFF
        g = g * 0xFF
        b = b * 0xFF
        cmap.append("#%02X%02X%02X" % (r, g, b))
    cmap.append("#000000")
    return cmap

def SetupColormap(ncolors):
    return SetupColormap0(ncolors)

def Graymap(ncolors):
    return SetupColormap6(ncolors)

def SetupColormap1(ncolors):
    cmap = []
    for i in range(0, ncolors):
        dd = (i * 359.0) / float(ncolors)
        r, g, b = hls2rgb(dd, .6, .99)
        r = r * 0xFF
        g = g * 0xFF
        b = b * 0xFF
        cmap.append("#%02X%02X%02X" % (r, g, b))
    cmap.append("#000000")
    return cmap

def SetupColormap2(ncolors):
    cmap = []
    for i in range(0, ncolors):
        dd = (i * 359.0) / float(ncolors)
        r, g, b = hls2rgb(dd, .30, .90)
```

```
        r = r * 0xFF
        g = g * 0xFF
        b = b * 0xFF
        cmap.append ("%02X%02X%02X" % (r, g, b))
    cmap.append ("%000000")
    return cmap

def SetupColormap3 (ncolors) :
    cmap = []
    for i in range (0, ncolors) :
        dd = (i * 359.0) /float (ncolors)
        if (i < 6) :
            r, g, b = hls2rgb (dd, .5, .70)
        else:
            r, g, b = hls2rgb (dd, .6, .90)
        r = r * 0xFF
        g = g * 0xFF
        b = b * 0xFF
        cmap.append ("%02X%02X%02X" % (r, g, b))
    cmap.append ("%000000")
    return cmap

def SetupColormap4 (ncolors) :
    cmap = []
    for i in range (0, ncolors) :
        dd = (i * 359.0) /float (ncolors)
        if (i%2) :
            r, g, b = hls2rgb (dd, .45, .99)
        else:
            r, g, b = hls2rgb (dd, .55, .99)
        r = r * 0xFF
        g = g * 0xFF
        b = b * 0xFF
        cmap.append ("%02X%02X%02X" % (r, g, b))
```

```
cmap.append("#000000")
return cmap

def SetupColormap5 (ncolors) :
    cmap = []
    if ncolors < 3:
        cmap.append("#FFFFFF")
        cmap.append("#000000")
        return cmap
    for i in range (0, ncolors) :
        dd = (i * 359.0) /float (ncolors)
        r, g, b = hls2rgb (dd, 0.5, 0.9)
        r = 0xFF - (r * 0xFF)
        g = 0xFF - (g * 0xFF)
        b = 0xFF - (b * 0xFF)
        cmap.append ("%02X%02X%02X" % (r, g, b))
    cmap.append("#000000")
    return cmap

def SetupColormap6 (ncolors) :
    cmap = []
    xd = 1.0/ncolors
    for i in range (0, ncolors) :
        r = g = b = 0xFF - ((i * xd) * 0xFF)
        cmap.append ("%02X%02X%02X" % (r, g, b))
    cmap.append("#000000")
    return cmap

if __name__ == "__main__":
    from Tkinter import *
    from Canvas import Oval, Arc
    from math import *
    ncolors=360
    cwidth = cheight = 401
    cmapcommands=[SetupColormap0,
```

```
        SetupColormap1.  
        SetupColormap2.  
        SetupColormap3.  
        SetupColormap4.  
        SetupColormap5.  
        SetupColormap6.  
    ]  
    cm = 0  
    cmapcom=cmapcommands[cm]  
  
    def die (event=0) :  
        sys.exit (0)  
  
    def drawcmap (wdg) :  
        global ncolors, cmap  
        ar = Oval (wdg, 0, 0, 400, 400)  
        dg = 360.0 / ncolors  
        for i in range (ncolors) :  
            e = i * dg  
            e = 90.0 + e  
            if e > 360.0:  
                e = e - 360.0  
            ps = Arc (cv, 0, 0, 400, 400, start=e, extent=dg, fill=cmap[i],  
outline="")  
  
    def next (event=0) :  
        global cmap, cm, cv, cmapcom, cmapcommands, ncolors, lbl  
        cm= (cm+1) %len (cmapcommands)  
        lbl["text"] = "SetupColormap%d () " % (cm)  
        cmapcom=cmapcommands[cm]  
        cmap = cmapcom (ncolors)  
        cv.delete (ALL)  
        drawcmap (cv)  
  
    def prev (event=0) :
```

```

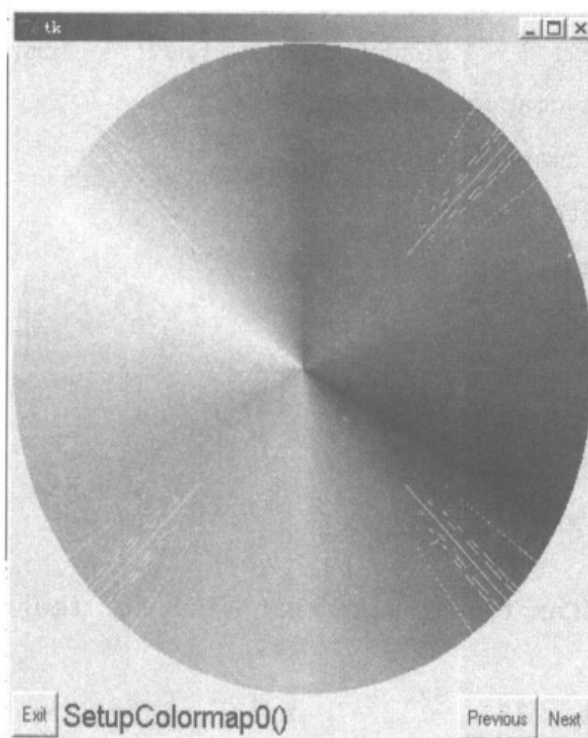
global cmap, cm, cv, cmapcom, cmapcommands, ncolors, lbl
cm= (cm-1) %len (cmapcommands)
lbl["text"] = "SetupColormap%d () " % (cm)
cmapcom=cmapcommands[cm]
cmap = cmapcom (ncolors)
cv.delete (ALL)
drawcmap (cv)

if len (sys.argv) >1:
    ncolors=string.atoi (sys.argv[1])
    if 2 < ncolors < 1101:
        pass
    else:
        print "Can't deal with that many (or few) colors. Resetting to
360."
        ncolors = 360
    cmap=cmapcom (ncolors)
    root=Tk ()
    cv = Canvas (root, width=cwidth, height=cheight, borderwidth=0,
        highlightthickness=0)
    frame=Frame (root)
    qbutton=Button (frame, text="Exit", command=die)
    lbl=Label (frame, text="SetupColormap0 () ", font="Helvetica 14")
    nextbutton=Button (frame, text="Next", command=next)
    prevbutton=Button (frame, text="Previous", command=prev)
    cv.pack (side=TOP)
    frame.pack (expand=1, fill=BOTH)
    qbutton.pack (side=LEFT)
    lbl.pack (side=LEFT)
    nextbutton.pack (side=RIGHT)
    prevbutton.pack (side=RIGHT)

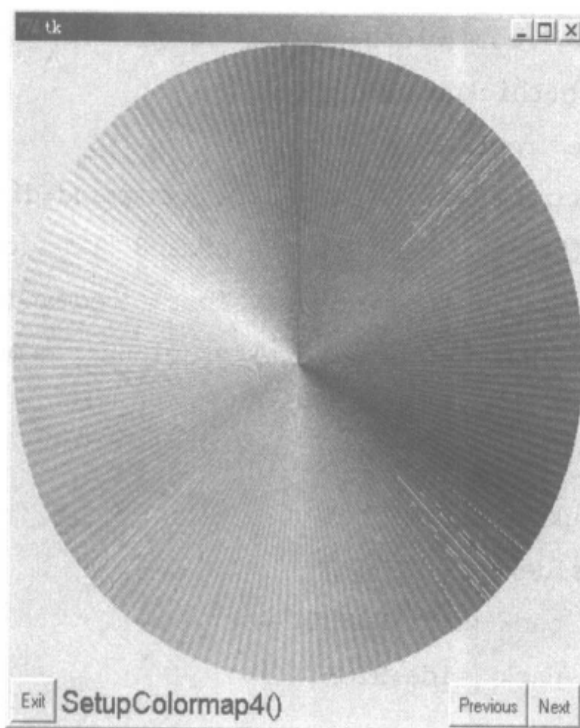
    drawcmap (cv)
root.mainloop ()

```


执行结果:



(1)



(2)

11.18 CGI 编程

例1: 这是一个 Python 用于 CGI 的编程, 主要是利用 SMTP 将注册信息通过邮件发送给制造厂商。

```
SMTP.py:
import time, string, select
from socket import *

msg_pattern = """From: %s\r
To: %s\r
Subject: %s\r
Date: %s\r
X-Mailer: Python SMTP class v 1.2\r
\r
%s
"""

class SMTP:
    # display a debug message
    def say (self, msg) :
        s = time.strftime ('%c', time.localtime (time.time
        ())) + ': ' + msg
        print s

    #
    def _init_ (self, host='localhost', lhost='localhost') :
        self.error = 'SMTP error'
        try:
            self.conn = socket (AF_INET, SOCK_STREAM)
            self.file = self.conn.makefile ()
            self.conn.connect (host, 25)
            self.check ('220')
            self.send ('helo ' + lhost)
            self.check ('250')
        except socket.error:
```

```
        self.conn = None

#
def readline (self) :
    r, w, e = select.select ([self.file], [], [], 30)
    if r == []: raise self.error, "Read from remote host timed out"
    return r[0].readline ()
def check (self, lev) :
    dat = self.readline ()
    if (lev != '') : # means don't care about the result
code
                                if dat[:3] != lev:
                                    raise self.error,
"Unexpected result code!: "+dat
                                # what to do here?
Need to exit gracefully, or retry? TBD
#
def send (self, line) :
    self.conn.send (line + '\n')
#
def send_body (self, mfrom, mto, subj, body) :
    self.conn.send (msg_pattern% (mfrom, mto, subj,
time.ctime (time.time ()), body) + '\n' )
#
def send_end_of_data (self) :
    #self.say ('Sending end of data marker')
    self.conn.send ("\n.\n")

def send_message (self, mfrom, mto, subj, body) :
    self.send ("mail from:<%s>"%mfrom)
    self.check ('250')
    self.send ("rcpt to:<%s>"%mto)
    self.check ('250')
    self.send ("data")
    self.check ('354')
```

```

        self.send_body (mfrom, mto, subj, body)
        self.send_end_of_data ()
        self.check ('250')

    #
    def close (self) :
        self.send ("quit")
        self.check ('221')
        self.conn.close ()

    #
#

if __name__ == '__main__':
    test_msg = """
This is a test of mr. Moukhine :)
"""

    import pdb
    #s = SMTP ("folwell.com")
    #s = SMTP ("65-37.lgcit.com")
    s = SMTP ("mail.callware.com")
    #pdb.run ('s.send ("gandalf@rosmail.com", "gandalf@rosmail.com",
    "Test", "This is a test message.") ')
    #s.send_message ("fred@acme.com", "richard@folwell.com", "Test",
    test_msg)
    #s.send_message ("gandalf@lgcit.com", "gandalf@lgcit.com",
    "Test", test_msg)
    s.send_message ("ivanlan@callware.com", "ivanlan@callware.com",
    "Test", test_msg)
    s.close ()

#

register.py:
#!c:\Python\python.exe
import os
import sys

```

```
import cgi
import SMTP

thisscript="http://www.pauhtun.org/cgi-bin/register.py"
mailhost="mail.callware.com"
mailfrom="ivanlan@callware.com"
mailto="ivanlan@callware.com"
cccto="ivanlan@cailware.com"

def print_content () :
    print "Content-type: text/html"
    print

def print_header () :
    print """<html><title>Pythonic Registration Form</title>
    <body bgcolor=white text=black>"""

def print_footer () :
    print "</body></html>"

def print_form () :
    print """<h1 align=center>Please Register!</h1>
    <p>Please register software before downloading it. Thank you,
    %s
    <hr>
    <p>
    <form name=form action=%s method=post>
    Enter your name:
    <input type="edit" name="name"><br>
    Enter your email address:
    <input type="edit" name="email"><br>
    Enter the name of the software you are downloading:
    <input type="edit" name="software"><br>
    Enter the amount you are willing to pay in dollars for the software:
    <input type="edit" name="pay"><br>
    <hr>
```

```

    <input type="submit">
    <input type="reset">
</form>
""" % (os.environ["SERVER_NAME"], thisscript)

print_content ()
print_header ()

form = cgi.FieldStorage ()
if form.has_key ("name") and form.has_key ("email") :
    nm = form["name"].value
    em = form["email"].value
    sw = form["software"].value
    mn = form["pay"].value
    print """<p align=center><font size=7>Thank you!
    </font><font size=3><br>
    <hr>
    """

    print """<p align=left>Thank you %s. Your email address, %s,
    will shortly receive an invoice for %s dollars, for downloading
    the %s package. If you do not pay up within 5 business days,
    all your files will be crased.<br>
    <hr>
    <p align=right>Have a nice day
    """ % (nm, em, mn, sw)

    msg="""Hello, %s (%s) :
    You recently downloaded %s from %s.
    Please send %s dollars immediately to:

    Ransom
    PO Box 6969
    Washington DC 55512

    If you do not send money, we have a catapult.
    Thank you for your attention to this matter.

```

Anonymous

```
""" % (nm, em, sw, os.environ["SERVER_NAME"], mn)

    s=SMTP.SMTP (mailhost)
    s.send_message (mailfrom, em,
                    "Software Registration", msg)
    s.send_message (mailfrom, ccto,
                    "Software Registration", msg)
    s.close ()
else:
    print_form ()
print_footer ()
```

第二部分 wxPython 程序设计



第 12 章 wxPython 在 Win32 下编程

一种可供 Python 使用的 GUI 工具包叫做 wxPython。目前这个工具对于 Python 环境来说还是陌生的，但正在 Python 开发者中间快速地流行起来。wxPython 是 Python 扩展模块，它封装了 wxWindows C++ 类库。

wxPython 是一个为 Python 提供的交叉平台 GUI 框架工具，它在 Windows 平台上相当成熟。它是基于流行的 wxWindows C++ 框架的 Python，为 GUI 开发者提供了一种有吸引力的替代工具。

12.1 wxPython 简介

12.1.1 wxWindows

wxWindows 是一个自由 C++ 编程框架，被设计用来实现跨平台编程。wxWindows 2.0 支持 Windows 3.1/95/98/NT，Unix 下支持 GTK/Motif/Lesstif，还有正在开发中的 Mac 版本。其他系统正在考虑中。

wxWindows 是一套库函数，允许 C++ 应用程序只需要微量的源代码改动，就可在几种不同类型的机构上编译和运行。每一种支持的 GUI 都有一个对应的库（像 Motif 或 Windows）。同时为了实现 GUI 功能提供了通用的 API，还为了处理一些通常要用到的操作系统设备提供的功能，应用程序可以根据需要使用或替换，这样就会节省大量的编码工作。并且基本数据结构，像字符串，链表和哈希表也提供了。

控件的本地版本，通用对话框和其他的窗口类型被用在支持它们的平台上。对于其他的平台，相适应的替代品使用 wxWindows 自身来生成。例如，在 Win32 平台上，使用了本地的列表控件，但是在 GTK 中，具有相似功能的通用列表控件则是使用 wxWindows 类库创建的。

有经验的 Windows 程序员对于 wxWindows 对象模型会感到像是在家一样。类和原则的许多地方都很相似。例如，多文档界面，用 GDI 对象，如刷子、笔、在上下文设置上绘图，等等。

12.1.2 wxWindows + Python = wxPython

wxPython 是一个 Python 扩展模块，它提供了一套从 wxWindows 库到 Python 语言的绑定。换句话说，扩展模块允许 Python 程序员创建 wxWindows 类的实例，并且调用这些类的方法。

wxPython 扩展模块试图尽可能的将 wxWindows 的类的层次也镜像下来。这就是说在 wxPython 中有一个 wxFrame 的类，看上去，各种功能几乎同 C++ 版本中的 wxFrame 一样。

wxPython 与 C++ 版本如此接近，这样 wxPython 文档的大多数实际上是对 C++ 文档中 wxPython 与之不同地方的注解。其中还包含了一系列的例子程序，和一系列帮助程序员开始使用 wxPython 的文档页。

12.2 初识wxPython

12.2.1 哪里可以得到 wxPython

wxPython 的最新版本可以在 <http://alldunn.com/wxPython/> 上找到。你可以从这个站点下载一个 Win32 系统的自安装软件，其中包含一个已经生成好的扩展模块，HTML 帮助格式文档和一组示例程序。

你可以从这个站点获得 Linux RPM，wxPython 源码，原始的 HTML 文档和其他站点的链接，邮件列表，wxPython FAQ，等等。如果你想自己从源代码创建 wxPython，你也需要 wxWindows 源代码，可以从 <http://www.wxwindows.org/> 得到。在本书的配套光盘中你也可以找到这些软件。

实践证明，学习的最好方法就是动手，接着做实验，观察得到的结果。所以你应该下载和安装 wxPython，启动你常用的文本编辑器，准备执行在下面几节所读到的东西。

12.2.2 一个简单的例子

应该对下面的小 wxPython 程序进行熟悉，当读到跟着的解释时，可以回过头来进行参考：

```
from wxPython.wx import *  
  
class MyApp (wxApp) :  
    def OnInit (self) :
```

```
        frame = wxFrame (NULL, -1, "Hello from wxPython")
        frame.Show (true)
        self.SetTopWindow (frame)
    return true

app = MyApp (0)
app.MainLoop ()
```

当运行这个程序时，你应该看到像图 12.1 的窗口显示出来。

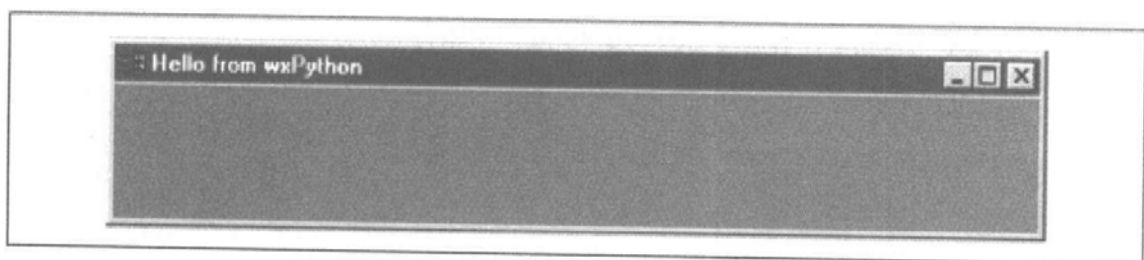


图 12.1 一个基本的 wxPython 程序

要做的第一件事情就是导入整个 wxPython 库，使用 `from wxPython import *` 语句。这是编 wxPython 程序的通常习惯，但是你可以根据需要进行更明确的导入。

每一个 wxPython 应用程序需要从 `wxApp` 派生出一个类，并且为其提供一个 `OnInit` 方法。框架（即窗口）会调用这个方法作为自身初始化序列的一部分。`OnInit` 通常是用来创建窗口，和对于程序开始运行完全必需的操作。在例子中，你创建了一个没有父亲的框架，标题是“Hello from wxPython”，然后显示它。我们也可以在它的构造函数中为框架指定位置和大小，但是因为这里没有，所以这里使用了缺省值。`OnInit` 方法的最后两行可能对于所有应用程序来说都是一样的。`SetTopWindow` 方法告诉 `wxWindows`，这个框架是应用程序主框架其中之一（在这个例子中只有一个），并且你返回 `true` 来表明成功。当所有顶层窗口关闭，应用程序结束。

脚本的最后两行可能又是对于所有的 wxPython 应用程序是一样的。你创建了一个应用程序类的实例，并且调用它的 `MainLoop` 方法。`MainLoop` 是应用程序的心脏：在这里事情被处理，并且被发送到各个窗口，当最后一个窗口关闭后，它返回。幸运的是，`wxWindows` 对你屏蔽了各种 GUI 工具包在事件处理上的不同。

大多数情况下，你会想要定制应用程序的主框架，所以使用普通的 `wxFrame` 是不够的。你可能希望，从 `wxFrame` 派生出自己的类进行定制。下一个例子定义了一个框架类，并且在

应用程序中的 `OnInit` 方法中创建了一个实例。注意除了在 `OnInit` 中创建的类的名字, `MyApp` 代码的其他部分同以前的例子是一样的。这个代码的显示结果如图 12.2。

```
from wxPython.wx import *
ID_ABOUT = 101
ID_EXIT = 102
class MyFrame (wxFrame) :
    def __init_ (self, parent, ID, title) :
        wxFrame.__init_ (self, parent, ID, title,
            wxDefaultPosition, wxSize (200, 150))
        self.CreateStatusBar ()
        self.SetStatusText ("This is the statusbar")
        menu = wxMenu ()
        menu.Append (ID_ABOUT, "&About",
            "More information about this program")
        menu.AppendSeparator ()
        menu.Append (ID_EXIT, "E&xit", "Terminate the program")
        menuBar = wxMenuBar ()
        menuBar.Append (menu, "&File");
        self.SetMenuBar (menuBar)
class MyApp (wxApp) :
    def OnInit (self) :
        frame = MyFrame (NULL, -1, "Hello from wxPython")
        frame.Show (true)
        self.SetTopWindow (frame)
        return true
app = MyApp (0)
```

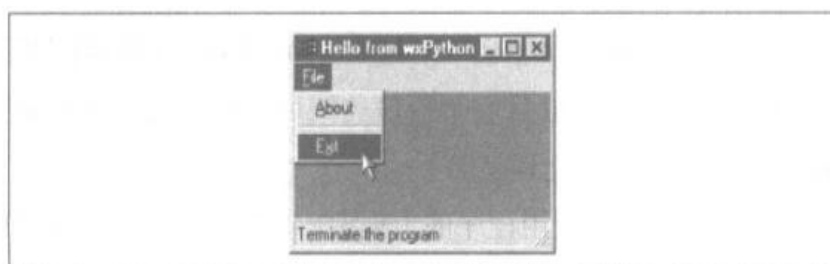


图 12.2 一个带菜单的 wxPython 程序

这个例子显示了一些 `wxFrame` 内建的功能。例如，为框架创建一个状态条只要简单地调用一个方法。框架本身会自动地管理它的位置、大小和绘制。另一方面，如果你想定制状态条，从你自己的 `wxStatusBar` 派生类创建实例，并将其附加到框架上。

在这个例子中也演示了创建一个简单的菜单条和一个下拉菜单。期待的菜单功能全部都支持：层叠子菜单、可核选的项和弹出菜单等等；你要做的只是创建一个菜单对象，向它追加菜单项。菜单项可以是像这里显示的文本，或其他的菜单。每一项你可以有选择地指定一些简单的帮助性文本，就像我们所做的。当菜单项被选中，这些文本会自动地显示在状态条上。

12.2.3 在 wxPython 中的事件

上一个例子没有做的一件事情就是：展示如何让菜单自动做一些事情。如果你运行这个例子，并且从菜单中选中 `Exit`，什么都没有发生。下一个例子改正了这个小问题。

为了在 `wxPython` 中处理事件，任何方法（或同样方式的独立函数）都可以使用工具包中的帮助性函数与任意事件相连。`wxPython` 也提供了一个 `wxEvent` 类和一整串派生类，包含了事件的细节。每次当发生一个事件，一个方法被调用，一个从 `wxEvent` 派生的对象被作为一参数传递，事件对象的实际类型要依赖于事件的类型。`wxSizeEvent` 用于当窗口改变大小，`wxCommandEvent` 用于菜单选择和按钮点击，`wxMouseEvent` 用于（可以猜到）鼠标的移动，等等。

为了解决上一个例子中的小问题，你要做的就是，在 `MyFrame` 构造函数中增加两行，并且增加处理事件的一些方法。我们也演示了一个通用对话框，`wxMessageDialog`。下面就是代码，黑体部分表示新的部分，运行结果见图 12.3。

```
from wxPython.wx import *

ID_ABOUT = 101
ID_EXIT = 102

class MyFrame (wxFrame) :
    def __init__ (self, parent, ID, title) :
        wxFrame.__init__ (self, parent, ID, title,
            wxDefaultPosition, wxSize (200, 150))
        self.CreateStatusBar ()
        self.SetStatusText ("This is the statusbar")
        menu = wxMenu ()
        menu.Append (ID_ABOUT, "&About",
```

```
"More information about this program")
menu.AppendSeparator ()
menu.Append (ID_EXIT, "E&xit", "Terminate the program")
menuBar = wxMenuBar ()
menuBar.Append (menu, "&File");
self.SetMenuBar (menuBar)
EVT_MENU (self, ID_ABOUT, self.OnAbout)
EVT_MENU (self, ID_EXIT, self.TimeToQuit)
def OnAbout (self, event) :
    dlg = wxMessageDialog (self, "This sample program shows off\n"
    "frames, menus, statusbars, and this\n"
    "message dialog.",
    "About Me", wxOK | wxICON_INFORMATION)
    dlg.ShowModal ()
    dlg.Destroy ()
def TimeToQuit (self, event) :
    self.Close (true)
class MyApp (wxApp) :
    def OnInit (self) :
        frame = MyFrame (NULL, -1, "Hello from wxPython")
        frame.Show (true)
        self.SetTopWindow (frame)
        return true

app = MyApp (0)
app.MainLoop ()
```

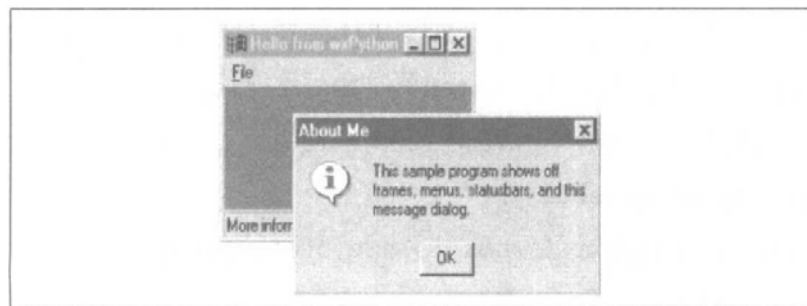


图 12.3 带有一个 About 对话框的程序

在这里调用的 EVT_MENU 函数是一个帮助性函数，是为了将事件与方法连在一起。有时候，如果你将函数调用翻译成英语，它可以帮助我们理解发生了什么。第一个说的是，“对于任一个菜单项选中事件，发送窗口本身和一个 ID_ABOUT 的 ID 号，调用 self.OnAbout 方法。”

有很多 EVT_* 帮助性函数，所有的都对应某个事件类型，或事件。下面将常用的一些事件列出，见表 1。要了解更多的细节请参阅 wxPython。

12.1 常见 wxPython 事件函数

事件函数	事件描述
EVT_SIZE	由于用户干预或由程序实现，当一个窗口大小发生改变时发送给窗口
EVT_MOVE	由于用户干预或由程序实现，当一个窗口被移动时发送给窗口
EVT_CLOSE	当一个框架被要求关闭时发送给框架。除非关闭是强制性的，否则可以调用 event.Veto(true) 来取消关闭
EVT_PAINT	无论何时当窗口的一部分需要重绘时发送给窗口
EVT_CHAR	当窗口拥有输入焦点时，每产生非修改性（Shift 键等等）按键时发送
EVT_IDLE	这个事件会当系统没有处理其他事件时定期的发送
EVT_LEFT_DOWN	鼠标左键按下
EVT_LEFT_UP	鼠标左键抬起
EVT_LEFT_DCLICK	鼠标左键双击
EVT_MOTION	鼠标在移动
EVT_SCROLL	滚动条被操作。这个事件其实是一组事件的集合，如果需要可以被单独捕捉
EVT_BUTTON	按钮被点击
EVT_MENU	菜单被选中

12.3 用Python创建一个Doubletalk浏览器

现在让我们做些有用的东西，用这种方法学习更多关于 wxPython 框架的知识。就像其他的 GUI 工具包所展示的，我们将创建一个小型的应用程序，围绕着 Doubletalk 类库（它允许浏览和编辑交易）。

12.3.1 MDI 框架

我们打算实现一个多文档界面（即 MDI），子框架除了为独立的“文档”外，其中还包含着交易数据的不同视图。如同前面的例子，要做的第一件事就是创建一个应用程序类，并且在它的 `OnInit` 方法中创建一个主框架：

```
from wxPython.wx import *

class DoubleTalkBrowserApp (wxApp) :
    def OnInit (self) :
        frame = MainFrame (NULL)
        frame.Show (true)
        self.SetTopWindow (frame)
        return true

app = DoubleTalkBrowserApp (0)
app.MainLoop ()
```

因为我们正在使用 MDI，所以需要有一个特别的类用来作为框架的基本类。这里的给出主程序框架的初始化方法的代码：

```
class MainFrame (wxMDIParentFrame) :
    title = "Doubletalk Browser - wxPython Edition"
    def __init__ (self, parent) :
        wxMDIParentFrame.__init__ (self, parent, -1, self.title)
        self.hookset = None
        self.views = []
        if wxPlatform == '_WXMSW_':
            self.icon = wxIcon ('chart7.ico', wxBITMAP_TYPE_ICO)
            self.SetIcon (self.icon)
            # 创建一个状态条，在右边显示时间和日期
            sb = self.CreateStatusBar (2)
            sb.SetStatusWidths ([-1, 150])
            self.timer = wxPyTimer (self.Notify)
            self.timer.Start (1000)
            self.Notify ()
```

```

menu = self.MakeMenu (false)
self.SetMenuBar (menu)
menu.EnableTop (1, false)
EVT_MENU (self, ID_OPEN, self.OnMenuOpen)
EVT_MENU (self, ID_CLOSE, self.OnMenuClose)
EVT_MENU (self, ID_SAVE, self.OnMenuSave)
EVT_MENU (self, ID_SAVEAS, self.OnMenuSaveAs)
EVT_MENU (self, ID_EXIT, self.OnMenuExit)
EVT_MENU (self, ID_ABOUT, self.OnMenuAbout)
EVT_MENU (self, ID_ADD, self.OnAddTrans)
EVT_MENU (self, ID_JRNL, self.OnViewJournal)
EVT_MENU (self, ID_DTAIL, self.OnViewDetail)
EVT_CLOSE (self, self.OnCloseWindow)

```

图 12.4 显示了到现在应用程序的状态。

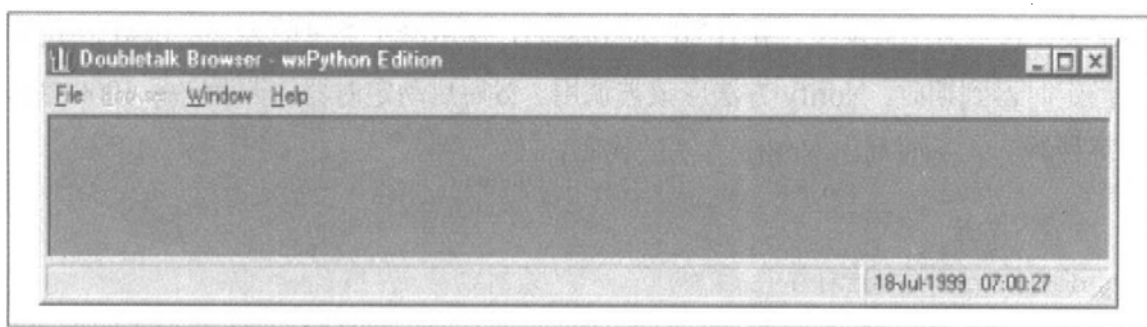


图 12.4 第一个 MDI wxPython 程序

很明显，我们没有展示全部的代码，但是随着我们一点一点地学习，我们将最终将其变得完整。

注意，使用 `wxMDIParentFrame` 作为 `MainFrame` 的基类。通过使用这个类，你会自动地获得为实现 MDI 应用程序的所有需要的东西，不需要关心在外表后面发生了什么。`wxMDIParentFrame` 类有着与 `wxFrame` 类同样的接口，只是拥有一些额外的方法。通常将一个单文档接口程序改为一个多文档程序，只是像改变应用程序派生类的基类一样容易。对于 `wxMDIParentFrame` 类，存在相对应的 `wxMDIChildFrame` 类，用于作文档窗口，在后面就会看到。如果你需要处理 MDI 父窗口的客户区域（或背景区域），你可以使用 `wxMDIClientWindow` 类。你可以使用它来在所有子窗口的后面放置一个背景图像。

12.3.2 图标

前面的代码做的下一件事情是创建一个图标，并将其与框架相连。通常的 Windows 应用程序从资源文件中装入像图标一样的资源项，资源文件与执行码链在一起。因为 wxPython 程序没有二进制执行文件，你需要通过指定全路径的 .ico 文件来创建图标。将图标指定给框架只要调用框架的 `SetIcon` 方法。

12.3.3 时间

你可能已经注意到在图 12.4 中，状态条有两段，在第二段中显示着日期和时间。下面在初始化方法中的几行代码实现了这个功能。框架的 `CreateStatusBar` 方法使用一个可选参数，这个参数指明了要创建的段的值，并且可以将一个整数列表传给 `SetStatusWidths`，用来指明对于每一段应保留多少像素。-1 意味着第一段应该占满剩余的空间。

为了更新日期和时间，你创建了一个 `wxPyTimer` 对象。在 wxPython 中有两种定时器类。第一个就是这里所用的 `wxPyTimer`，它接收一个函数或方法作为一个回调函数。另一个是 `wxTimer` 类，它是为了被继承，并且当时间到期时，调用派生类里的要求的函数。在例子中，你指明当定时器到期时，`Notify` 方法应该被调用。然后启动定时器，告诉它每 1000 毫秒激活（即，每秒钟）。下面列出 `Notify` 方法的代码：

```
# 激活处理
def Notify (self) :
    t = time.localtime (time.time ())
    st = time.strftime ("%d-%b-%Y %I:%M:%S", t)
    self.SetStatusText (st, 1)
```

首先使用 Python 的 `time` 模块来得到当前时间，并且将日期格式化为一种好看的，人类可读的格式串。然后通过调用框架的 `SetStatusText` 方法，你可以将这个字符串放进状态条，在这个例子中为第一段中。

12.3.4 主菜单

在下面的几行代码，如你所见，我们将菜单的创建放入一个独立的方法中。这样做主要有两个原因。第一个就是减少在 `_init_` 方法中的混乱，从而更好的组织类的功能。第二个原因由于 MDI，需要这样做。对于所有的 MDI 应用程序，每一个子框架可以拥有它自己的菜单

条，当这个框架被选中时会自动更新。

我们的例子所使用的方法既可以从 BookSet 菜单中增加也可以删除一个单独的菜单项，要看一个视图能否选择交易进行编辑。下面列出 MakeMenu 方法的代码。注意参数是怎样控制是否将 Edit Transaction 菜单项加入到菜单中的。根据需要，这个项只是允许或禁止可能会更好理解一些，但是那样你将不能看到，当活动窗口改变时，wxPython 是如何自动改变菜单的。还要注意，你没有创建 Window 菜单。wxMDIParentFrame 已经为你想到了：

```
def MakeMenu (self, withEdit) :
    fmenu = wxMenu ()
    fmenu.Append (ID_OPEN, "&Open BookSet", "Open a BookSet file")
    fmenu.Append (ID_CLOSE, "&Close BookSet",
        "Close the current BookSet")
    fmenu.Append (ID_SAVE, "&Save", "Save the current BookSet")
    fmenu.Append (ID_SAVEAS, "Save &As", "Save the current BookSet")
    fmenu.AppendSeparator ()
    fmenu.Append (ID_EXIT, "E&xit", "Terminate the program")
    dtmenu = wxMenu ()
    dtmenu.Append (ID_ADD, "&Add Transaction",
        "Add a new transaction")
    if withEdit:
        dtmenu.Append (ID_EDIT, "&Edit Transaction",
            "Edit selected transaction in current view")
        dtmenu.Append (ID_JRNL, "&Journal view",
            "Open or raise the journal view")
        dtmenu.Append (ID_DTAIL, "&Detail view",
            "Open or raise the detail view")
    hmenu = wxMenu ()
    hmenu.Append (ID_ABOUT, "&About",
        "More information about this program")
    main = wxMenuBar ()
    main.Append (fmenu, "&File")
    main.Append (dtmenu, "&Bookset")
    main.Append (hmenu, "&Help")
```

```
return main
```

如果你跳回到 `_init_` 方法, 注意在你创建菜单然后将它连到窗口之后, 菜单条的 `EnableTop` 方法被调用了。这就是如何禁止整个 `BookSet` 子菜单。(因为还没有 `BookSet` 文件打开, 你还不能真正地对它做任何事。) 使用 `Enable` 方法可以让你通过 ID 号对独立的菜单项允许或禁止。

`_init_` 方法的最后几行代码将事件处理器同各个菜单项连接。当我们解开在这些选项之后的功能时, 将一个 一个地查看它们。但是首先, 这里有简单一些的:

```
def OnMenuExit (self, event) :
    self.Close ()
def OnCloseWindow (self, event) :
    self.timer.Stop ()
    del self.timer
    del self.icon
    self.Destroy ()
def OnMenuAbout (self, event) :
    dlg = wxMessageDialog (self,
        "This program uses the Doubletalk package to\n"
        "demonstrate the wxPython toolkit.\n\n"
        "by Robin Dunn",
        "About", wxOK | wxICON_INFORMATION)
    dlg.ShowModal ()
    dlg.Destroy ()
```

用户从 `File` 菜单中选择 `Exit`, 然后 `OnMenuExit` 方法被调用, 这个方法要求窗口将自己关闭。无论何时当窗口需要关闭时, 不管是因为 `Close` 方法被调用, 还是因为用户点击了标题条上的关闭按钮, `OnCloseWindow` 方法被调用。如果你想用像“你确实想退出吗?”这样的信息提示用户, 就在这里实现它。如果他决定不退出, 只要调用方法 `event.Veto (true)`。

大部分的程序需要一个比 `wxMessageDialog` 所提供的更富有想象的 `About` 对话框, 但是出于我们的目的, 这样就很好了。不要忘了调用对话框的 `Destroy` 方法, 否则你可能泄漏内存。

12.3.5 wxFileDialog

在对 BookSet 做事情之前，你需要先打开一个文件。为了打开文件，使用 wxFileDialog 通用对话框。这个对话框同所有你看到的 Windows 应用程序的**文件→打开**对话框一样，全部都封装在了一个同 wxPython 兼容良好的类接口中了。

这里是捕捉文件→打开事件的事件处理器，图 5 显示了执行后的对话框：

```
def OnMenuOpen (self, event) :  
    # This should be checking if another is already open,  
    # but is left as an exercise for the reader...  
    dlg = wxFileDialog (self)  
    dlg.SetStyle (wxOPEN)  
    dlg.SetWildcard ("*.dtj")  
    if dlg.ShowModal () == wxID_OK:  
        self.path = dlg.GetPath ()  
        self.SetTitle (self.title + ' - ' + self.path)  
        self.bookset = BookSet ()  
        self.bookset.load (self.path)  
        self.GetMenuBar ().EnableTop (1, true)  
        win = JournalView (self, self.bookset, ID_EDIT)  
        self.views.append ((win, ID_JRNL))  
    dlg.Destroy ()
```

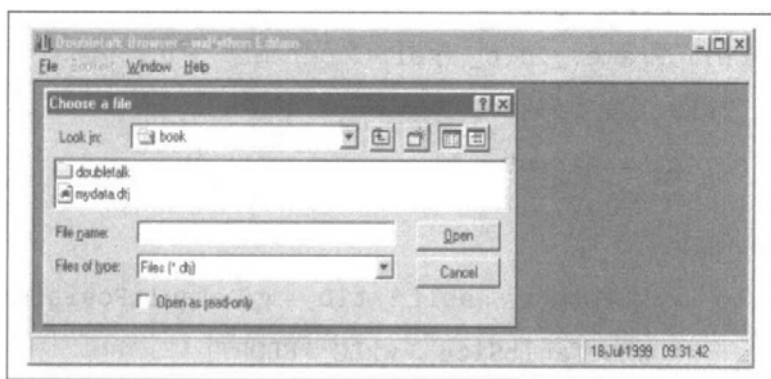


图 12.5 wxPython 浏览一个 Doubletalk 交易文件

开始是创建文件对话框，并且告诉它如何动作。接着显示对话框，给用户一个机会，选择一个 BookSet 文件。注意，这一次你检查了 ShowModal 方法的返回值。这就是对话框如何

告诉你结果是什么。缺省的，对话框理解其中的与按钮相连的 `wxID_OK` 和 `wxID_CANCEL` 消息识别 (ID) 号，并且当它们被点击时做正确的事情。对于你所创建的对话框，如果愿意也可以指定其他的返回值。

在文件对话框成功完成之后，要做的第一件事就是询问对话框被选中的路径名是什么，然后使用这个路径来修改框架的标题，然后打开一个 **BookSet** 文件。

看一下后面一行。它重新使 **BookSet** 菜单有效，因为现在已经存在一个打开文件了。它实际是两行语句合成一句，相当于这两行：

```
menu = self.GetMenuBar ()
menu.EnableTop (1, true)
```

因为当用户打开一个文件时，让他们确实地看到一些东西是有意义的，你应该创建并且显示其中一个视图，用上面的 `OnMenuOpen` 处理函数中最后几行代码。在下面，我们会看到的。

12.3.6 wxListCtrl

日志视图是由 `wxListCtrl` 组成，其中每一个交易都有一个单行的小计。这个控件放置在 `wxMDIChildFrame` 中，并且因为它是框架中唯一的東西，所以不用担心设置或维护大小，框架会自动维护它。（不幸地是，因为某些平台在不同的时间发送第一个改变大小 (`resize`) 事件，有时候窗口显示时，它的子窗口大小会不正确。）

```
class JournalView (wxMDIChildFrame) :
    def __init__ (self, parent, bookset, editID) :
        wxMDIChildFrame.__init__ (self, parent, -1, "")
        self.bookset = bookset
        self.parent = parent

        tID = wxNewId ()
        self.lc = wxListCtrl (self, tID, wxDefaultPosition,
                               wxDefaultSize, wxLC_REPORT)
        ## Forces a resize event to get around a minor bug...
        self.SetSize (self.GetSize ())
        self.lc.InsertColumn (0, "Date")
        self.lc.InsertColumn (1, "Comment")
```

```

self.lc.InsertColumn (2, "Amount")

self.currentItem = 0

EVT_LIST_ITEM_SELECTED (self, tID, self.OnItemSelected)
EVT_LEFT_DCLICK (self.lc, self.OnDoubleClick)

menu = parent.MakeMenu (true)
self.SetMenuBar (menu)

EVT_MENU (self, editID, self.OnEdit)
EVT_CLOSE (self, self.OnCloseWindow)

self.UpdateView ()

```

图 13.6 显示出应用程序处理的很好，并且看上去像一个正规的 Windows 应用程序。

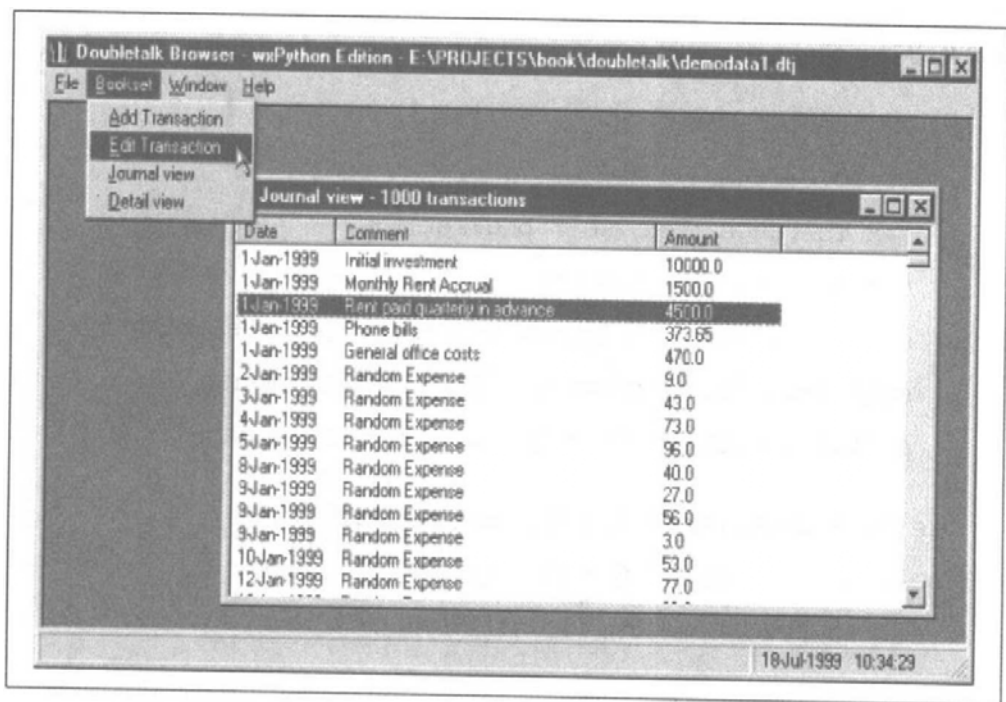


图 13.6 Doubletalk 交易清单

wxListCtrl 有很多的特性，但是对你来说都应该熟悉。在 wxPython 包装的下面，它同在 Windows 资源浏览器右边的面板用的是同样的控件。所有相同的选项都是可用的：大图标、小图标、列表模式和报告模式。你利用它们的列头（header）来定义列，然后为列表控件设置一些事件。当双击时，需要能够对交易进行编辑，那么为什么需要两个事件处理呢？当列表控件的一条被选中时，它会发送一个事件，但是它并不留意双击。另一方面，wxWindow

基类报告双击，但它并不知道列表控件。所以通过截获两个事件，你可以简单地实现你需要的功能。这里给出事件处理代码：

```
def OnItemSelected (self, event) :  
    self.currentItem = event.m_itemIndex  
  
def OnDoubleClick (self, event) :  
    self.OnEdit ()
```

在创建和建立列表控件之后，为这个框架创建了一个菜单条。这里你调用了在父框架中的生成菜单的方法，要求它增加 **Edit Transaction** 菜单项。

`_init_`方法所做的最后一件事是调用一个方法从 **BookSet** 中填充这个列表框。我们将其划分为一个单独的方法，这样不管什么时候 **BookSet** 数据发生改变，都可以被单独调用。下面是 **UpdateView** 方法：

```
def UpdateView (self) :  
    self.lc.DeleteAllItems ()  
    for x in range (len (self.bookset)) :  
        trans = self.bookset[x]  
        self.lc.InsertStringItem (x, trans.getDateString ())  
        self.lc.SetStringItem (x, 1, trans.comment)  
        self.lc.SetStringItem (x, 2, str (trans.magnitude ()))  
  
    self.lc.SetColumnWidth (0, wxLIST_AUTOSIZE)  
    self.lc.SetColumnWidth (1, wxLIST_AUTOSIZE)  
    self.lc.SetColumnWidth (2, wxLIST_AUTOSIZE)  
  
    self.SetTitle ("Journal view - %d transactions" %  
        len (self.bookset))
```

将数据放入一个列表中相当容易：只要插入每个列表项。在报告模式下，你插入一个列表项作为第一列，然后设置剩下列的值。对于例子中的每一列，只要从交易中取得一些数据，然后将其发送到列表控件中。如果你使用图标或图标和文本，要用不同的方法来处理它们。

现在在列表控件中已经有数据了，你应该重新设置列的大小。或者可以指定实际的像素宽度，或者让列表根据数据的宽度自动调整列的宽度。

JournalView 类要做的最后一件事是允许交易的编辑。在前面我们看到，当一个列表项被双击，一个名为 OnEdit 的方法被调用。代码为：

```
def OnEdit (self, *event) :
    if self.currentItem:
        trans = self.bookset[self.currentItem]
        dlg = EditTransDlg (self, trans,
                             self.bookset.getAccountList ())
        if dlg.ShowModal () == wxID_OK:
            trans = dlg.GetTrans ()
            self.bookset.edit (self.currentItem, trans)
            self.parent.UpdateViews ()
        dlg.Destroy ()
```

这个看上去像我们在主框架中对文件对话框的操作，并且的确你将会发现，在使用对话框时会经常使用这种模式。在这里需要注意的事情是调用在父窗口中的 UpdateViews ()。这就是如何管理，让 BookSet 的所有的视图保持更新。只要一个交易更新了，这个方法就会被调用，然后在所有的视图中循环，通知视图用它们的 UpdateView () 方法更新自身。

12.4 xPython窗口布局

wxPython 包含了许多强大的技术，用于控制你的窗口和控件的布局。它提供了几种可以替换的机制和几种有效的方法来完成同一件事情。允许程序员在特别的环境下使用可以工作最好或最习惯的机制。

12.4.1 约束

有一个叫做 wxLayoutConstraints 的类，它允许一个窗口位置和大小说明是相对于它的“兄弟”（同级控件）和它的“父亲”（父控件）。每一个 wxLayoutConstraints 对象是由八个 wxIndividualLayoutConstraint 对象组成，这些对象定义了不同类型的关系，就像哪一个窗口在这个窗口的上面，这个窗口的相对宽度是什么，等等。你通常需要指出八个中的四个约束条件，以便窗口被完全限定。例如，这个按钮将定位于它的父控件的中间，并且将总是占距父控件宽度的 50%：

```
b = wxButton (self.panelA, 100, ' Panel A `')
lc = wxLayoutConstraints {}
lc.centreX.SameAs (self.panelA, wxCentreX)
lc.centreY.SameAs (self.panelA, wxCentreY)
lc.height.AsIs ()
lc.width.PercentOf (self.panelA, wxWidth, 50)
b.SetConstraints (lc);
```

12.4.2 布局算法

名为 `wxLayoutAlgorithm` 的类实现了在 MDI 或 SDI 框架中子窗口的布局。它向框架的子控件发送 `wxCalculateLayoutEvent` 事件，向它们查询它们的大小的信息。因为使用了事件系统，因此这个技术可以应用于任何窗口，甚至那些不需要知道布局的类。然而你可能希望将 `wxSashLayoutWindow` 类用在你的子窗口上，因为这个类提供了用于请求事件的处理器，和

12.5.1 资源

wxWindows 库有一个可用的简单对话框编辑器，它可以帮助你安排一个对话框中控件的布局，并且生一个可在交叉平台上移植的资源文件。这个文件可以在运行时装入到程序中，并且可以立即转化成一个带有特别控件在上面的窗口。这个方法唯一的缺陷是你没有机会实现所生成窗口的子类化，但是如果用存在的控件类型和事件处理能够做你需要的任何事情，它应该执行的很好。最后，将会出现一个为 wxPython 特别设计的应用程序生成工具，它将为你或者生成资源文件或者生成实际的 Python 源代码。

12.5.2 强制力

最后，有一种强制力（brute force）机制，用来通过编程来指明每一个组件的精确位置。有时候一个窗口的布局需要不能适应任何一种 sizer，或者不能保证约束的复杂性，或者布局算法。对于这些情况，你可以重新回到手工方式去处理，但是对于比 Edit Transaction 对话框复杂多的东西，你可能不想去尝试它。

12.6 wxDialog and friends

下一步是建立一个用来编辑交易的对话框。对象由日期，说明和不定数目的交易记录组成，每条记录都有一个账户名和一个余额。我们知道，所有的记录加起来应该为 0，并且日期应该为一个有效日期。另外，为了编辑日期和说明，你需要能够增加，编辑和删除记录。图 12.7 显示了对于这个对话框的一个可能的布局，并且是这个例子中所使用的。

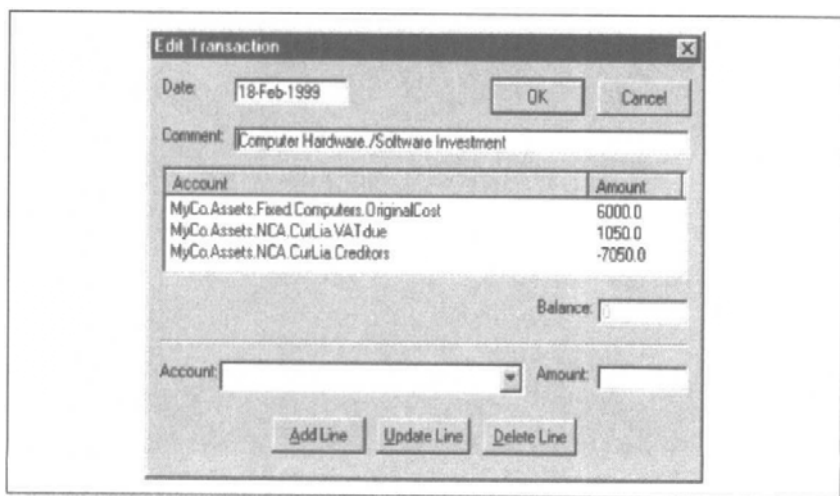


图 12.7 wxPython Doubletalk 交易编辑器

因为这里有很多的代码，让我们一步步地仔细检查这个类的初始化过程。下面是第一部分：

```
class EditTransDlg (wxDialog) :
    def __init__ (self, parent, trans, accountList) :
        wxDialog.__init__ (self, parent, -1, "")
        self.item = -1
        if trans:
            self.trans = copy.deepcopy (trans)
            self.SetTitle ("Edit Transaction")
        else:
            self.trans = Transaction ()
            self.trans.setDateString (dates.ddmmmyyyy (self.trans.date))
            self.SetTitle ("Add Transaction")
```

这是想当简单的东西。只是调用了父类的 `__init__` 方法，做一些初始化工作，并且判断是否你正在编辑一个存在的交易或创建一个新的交易。如果正在编辑一个存在的交易，使用 Python 拷贝模块来生成对象的拷贝。这样做因为你可能正好在编辑交易，并不想让被编辑的交易的任何部分留在 `BookSet` 中。如果对话框被用来增加一条新交易，则创建一条，然后修改它的日期，通过从日期中截去时间。在交易中的缺省日期包括了当前的时间，但是这个对话框只具备了处理日期部分。

第三部分 Python 的高级应用



第13章 Python 和 XML

XML 代表 Extensible Markup Language (eXtensible Markup Language 的缩写, 意为可扩展的标记语言)。XML 是一套定义语义标记的规则, 这些标记将文档分成许多部件并对这些部件加以标识。它也是元标记语言, 即定义了用于定义其他与特定领域有关的、语义的、结构化的标记语言的句法语言。由 W3C (World Wide Web Consortium, 互联网联合组织) 于 1998 年 2 月发布的一种标准, 同 HTML 一样是 SGML (Standard Generalized Markup Language, 标准通用标记语言) 的一个简化子集。由于它将 SGML 的丰富功能与 HTML 的易用性结合到了 Web 的应用中, 自推出以来, 迅速得到软件开发商的支持和程序开发人员的喜爱, 显示出强大的生命力。

让我们来回看一下可扩展标记语言 XML (eXtensible Markup Language) 的发展简史。

13.1 XML的发展历史

XML 有两个先驱——SGML 和 HTML, 这两个语言都是非常成功的标记语言, 但是它们都在某些方面存在着与生俱来的缺陷。SGML (Standard Generalized Markup Language) 的全称是标准通用标记语言, 它为语法标记提供了异常强大的工具, 同时具有极好的扩展性, 因此在分类和索引数据中非常有用。但是, SGML 非常复杂, 并且价格昂贵, 几个主要的浏览器厂商都明确拒绝支持 SGML, 使 SGML 在网上传播遇到了很大障碍。

相反, 超文本标记语言 HTML (HyperText Markup Language) 免费、简单, 在世界范围内得到了广泛的应用。它侧重于主页表现形式的描述, 大大丰富了主页的视觉、听觉效果, 为推动 WWW 的蓬勃发展、推动信息和知识的网上交流发挥了不可取代的作用。可是, HTML 也有如下几个致命的弱点, 这些弱点逐渐成为 HTML 继续发展应用的障碍。

HTML 是专门为描述主页的表现形式而设计的, 它疏于对信息语义及其内部结构的描述, 不能适应日益增多的信息检索要求和存档要求。

HTML 对表现形式的描述能力实际上也还非常不够, 它无法描述矢量图形、科技符号和一些其他的特殊显示效果。

HTML 的标记集日益臃肿, 而其松散的语法要求使得文档结构混乱而缺乏条理, 导致浏

览器的设计越来越复杂，降低了浏览的时间效率与空间效率。

正因为如此，1996 年人们开始致力于描述一种标记语言，它既具有 SGML 的强大功能和可扩展性，同时又具有 HTML 的简单性。XML 就是这样诞生的。

13.2 XML 的优点

XML 的优势之一是它允许各个组织、个人建立适合自己需要的标记集合，并且这些标记可以迅速地投入使用。这一特征使得 XML 可以在电子商务、政府文档、司法、出版、CAD/CAM、保险机构、厂商和中介组织信息交换等领域中一展身手，针对不同的系统、厂商提供各具特色的独立解决方案。

XML 的最大优点在于它的数据存储格式不受显示格式的制约。一般来说，一篇文档包括三个要素：数据、结构以及显示方式。对于 HTML 来说，显示方式内嵌在数据中，这样在创建文本时，要时时考虑输出格式，如果因为需求不同而需要对同样的内容进行不同风格的显示时，要从头创建一个全新的文档，重复工作量很大。此外 HTML 缺乏对数据结构的描述，对于应用程序理解文档内容、抽取语义信息都有诸多不便。

XML 把文档的三要素独立开来，分别处理。首先把显示格式从数据内容中独立出来，保存在样式单文件（Style Sheet）中，这样如果需要改变文档的显示方式，只要修改样式单文件就行了。XML 的自我描述性质能够很好地表现许多复杂的数据关系，使得基于 XML 的应用程序可以在 XML 文件中准确高效地搜索相关的数据内容，忽略其他不相关部分。XML 还有其他许多优点，比如它有利于不同系统之间的信息交流，完全可以充当网络语言，并希望成为数据和文档交换的标准机制。

当然，XML 作为一个新建立的标准，还有许多不足之处：它在强调了数据结构的同时，语义表达能力上略显不足，例如定义了<地址>这样一个标记，如果不是在文档中实际定义内容，我们就无法知道是要表达家庭住址还是 E-mail 地址。另外，XML 的有些技术尚未形成统一的标准，充分支持 XML 的应用处理程序很少，甚至浏览器对 XML 的支持也是有限的。

所以，XML 还并不能完全取代 HTML，毕竟 HTML 是最为方便、快捷的网上信息发布方式。况且 HTML 是描述数据显示的语言，而 XML 是描述数据及其结构的语言，二者在功能上也是截然不同的。

正像 SGML 和 HTML 一样，可扩展标记语言 XML 也是一种标记语言，它通过在数据中加入附加信息的方式来描述结构化数据。不过，XML 并非像 HTML 那样，只提供一组事先

已经定义好的标记。准确地说，它是一种元标记语言，允许程序开发人员根据它所提供的规则，制定各种各样的标记语言。在 XML 中，标记的语法是通过文档类型定义 DTD (Document Type Definition) 来描述的，也就是说，通过 DTD 来描述什么是有效的标记，并进一步定义标记语言的结构。除了定义标记的语法外，为了明确各个标记的含义，XML 还使用与之相连的样式单 (style sheet) 来向应用程序，比如浏览器，提供如何处理显示的指示说明。一言以蔽之，XML 是通过数据文档、DTD、样式单三个分离的部分来描述数据的。

虽然 XML 貌似复杂，但它有一些突出的优点：

1. 良好的可扩展性。

XML 允许各个不同的行业根据自己独特的需要制定自己的一套标记，同时，它并不要求所有浏览器都能处理这成千上万个标记，同样也不要求一个标记语言能够适合各个行业各个领域的应用，这种具体问题具体分析的方法更有助于标记语言的发展。

2. 内容与形式的分离。

正如前面所说，XML 中信息的显示方式已经从信息本身中抽取出来，放在了“样式单”中。这样做便于信息表现方式的修改，便于数据的搜索，也使得 XML 具有良好的自描述性，能够描述信息本身的含义甚至它们之间的关系。

3. 遵循严格的语法要求。

XML 不但要求标记配对、嵌套，而且还要求严格遵守 DTD 的规定。这增加了网页文档的可读性和可维护性，也大大减轻了浏览器开发人员的负担，提高了浏览器的时间空间效率。

4. 便于不同系统之间信息的传输。

不同企业、不同部门中往往存在着许多不同的系统，XML 可以用作各种不同系统之间的交流媒介，是一种非常理想的网络语言。

5. 具有较好的保值性。

XML 的保值性来自它的先驱之一 SGML 语言，可以为文档提供 50 年以上的寿命。正是基于这些优点，国际标准化组织_万维网联盟 W3C 推荐 XML 作为第二代网页发布语言。

13.3 XML的技术实现

在熟悉 XML 之前，我们至少应该了解一下这个技术到底是如何具体实现的。就目前的趋势来看（因为 XML 的有关标准改动都多达十七八次的，所以先不管它最后会怎样，先就目前的实现方式来看），要使得用户最后能够在客户端看到使用 XML 技术做出来的东西（如

果要是用行业术语来解释的话，就是说，如何使用 XSL 级联样表转换 XML 的文档成其他的格式例如 HTML。）），主要有下面的三种实现手段：

1. 第一种方式为：

让 XML 文档和与其关联的 XSL 级联样表同时被传送到客户端（通常使用的是浏览器），然后在客户端让 XML 文档根据 XSL 定义的显示格式显示其内容。

2. 第二种方式为：

在服务器端就使用 XSL 级联样表转换 XML 文档为其他的格式（通常为 HTML 格式）然后在把转换过的文档传送给客户端（一般使用浏览器）

3. 第三种方式为：

使用第三方的产品，在将 XML 文档放到服务器端之前就将该文档转换成其他的格式（一般为 HTML 格式）。然后服务器端和客户端就和平常处理 HTML 一样来处理了。

正是由于有这么三种不同的解决方法，也就决定了目前世界上处理 XML 文档而产生的不同的解决方案。对于开发人员来说，也必须了解有这么三种方式，才能够针对相应的问题而采取相应的解决方法。每种技术路线都产生了不同的软件，虽然从原理上来说，它们都是运用了同样的 XML 和 XSL 文件。例如：如果一个 Web 服务器直接把 XML 文档传递给 IE5 就是使用了第一种方式；如果在服务器端安装 IBM 的 alphaWorks' XML 使能软件就是使用了第二种方式；而如果使用命令行的 XT 程序直接将 XML 转换成 HTML 文档，然后把 HTML 文档放到服务器上就是使用了第三种方式。

但是这三种方式的共同点都是使用了同样的 XML 和 XSL 文件。

13.4 XML 的相关技术

XML 有很多相关的技术，将这些技术结合起来，才能充分发挥 XML 的强大功能。这些技术包括：Xlink 与 Xpointer（设置 XML 的超链接）、DOM（Document Object Model：文件对象模型，存取、操作文件的内容）、Namespaces（解决不同元素有相同名称的问题）、XHTML（下一代的 HTML）等。

13.4.1 Xlink 与 Xpointer

在 XML 的规范中，我们看到它并没有规定有关文件链接的问题。为了使 XML 文件也能够有类似 HTML 文件超链接的功能，W3C 制定了 Xlink 和 Xpointer 两种规范，其中 Xlink 是规定 XML 文件之间的链接规范（和 HTML 中的外链接相似），Xpointer 是规定 XML 文件

中不同位置之间的链接规范（类似 HTML 中的内链接）。

Xlink: Xlink 所设定的链接分为 Simple Link 和 Extended Link。其中，Simple Link 的链接功能和 HTML 的超链接基本上一样，而 Extended Link 则超出了 HTML 超链接的功能，它链接的对象可以一次设定多个，由多个标记来共同制定该链接。

在 XML 文件中使用 Xlink 元素的时候，必须要在 DTD 中声明这个元素。完整的声明样本如下（本样本声明了一个名为 simple 的 Simple Link 类型的 Xlink 元素）：

```
<!ELEMENT simple ANY>
<!ATTLIST simple
xml:link CDATA #FIXED"simple"
href CDATA #REQUIRED
role CDATA #IMPLIED
title CDATA #IMPLIED
inline {true|false} "true"
content-role CDATA #IMPLIED
content-title CDATA #IMPLIED
show {embed|replace|new} #IMPLIED
actuate {auto|user} #IMPLIED
behavior CDATA #IMPLIED>
```

可以看到，Xlink 元素有多种属性，通过对这些属性赋值，可以编制出多种多样的链接方式。下面我们对这些属性作些解释。

Xml:link: 指明链接类型是 Simple Link 还是 Extended Link。

href: 用来设定链接的地址，与 HTML 中 A 标记中的 href 属性一样。

role: 叙述该链接功能，提供给应用程序读取。

title: 叙述该链接功能，提供给用户读取，与 HTML 中 A 标记的 alt 属性相似。

inline: 有“true”和“false”两种取值，声明建立的链接是否以嵌入方式链接，缺省为“true”。

content-role 和 **content-title:** 和 role、title 类似，但它们叙述的是指向的内容，而不是链接的内容。

show: 有三种取值，replace 表示将链接的内容取代当前的内容，new 表示将链接的内容在一个新的窗口打开，embed 表示将链接的内容加入到当前的内容中。

actuate: 设置该链接是如何被激活。auto 表示 XML 文件被解读后，链接自动被激活。

而 user 表示, 该链接必须被用户手动激活, 也就是用户必须要用鼠标点击一下该链接。

behavior: 设置该链接被激活后, 将自动引发一些动作, 可用一些指令来设置链接激活后应用程序要作的事情。

当我们在 DTD 中声明 Xlink 元素后, 就可以在 XML 文件中使用这个元素。例如:

```
<simple href="http://www.cbnews.com/xml.htm" title="这是一篇介绍XML的文章"
role="XML article" content-role="good" cont-title="first" show="new"
actuate="user"
behavior="goto zero"/>
```

另外一种 Xlink 链接方式是 Extended Link, 它的特点是可以一次设定多个链接对象。同样, 在使用 Extended Link 类型的 Xlink 元素前必须在 DTD 中声明这个元素。声明方式和 Simple Link 类型的 Xlink 元素类似, 不同之处有两点, 第一, 声明 xml:link 属性时, 语句变为: xml:link CDATA #FIXED "extended"; 第二, 没有任何 href 属性和任何目标描述, 声明 Extended Link 类型的 Xlink 元素, 必须包括一套包含 href 定位的子元素。即在声明了 Extended Link 类型的 Xlink 元素之后, 还必须声明一个 xml:link 属性值为 locator 的子元素。例如:

```
<!ELEMENT aaa ANY>
<!ATTLIST aaa
xml:link CDATA #FIXED "extended"
inline (true|false) "true"
content-role CDATA #IMPLIED
content-title CDATA #IMPLIED>
<!ELEMENT bbb ANY>
<!ATTLIST bbb
xml:link CDATA #FIXED "locator"
role CDATA #IMPLIED
href CDATA #REQUIRED
title CDATA #IMPLIED
show (embed|replace|new) "replace"
actuate (auto|user) "user"
behavior CDATA #IMPLIED>
```

这样，我们就可以在 XML 文件中使用定义过的 Extended Link 的 Xlink 元素。如：

```
<aaa> 文章资料
<bbb href="http://www.cbinews.com/XML1.htm title="XML入门"/>
<bbb href="http://www.cbinews.com/XML2.htm title="XML进阶"/>
<bbb href="http://www.cbinews.com/XML3.htm title="XML应用"/>
</aaa>
```

当我们通过 CSS 或 XSL 将这个 XML 文件在浏览器中显示出来后，用户点击“文章资料”这个 Extended Link 的 Xlink 链接，将会出现一个选单，列出所有子元素中的标题，并将用户带到相应的位置。

13.4.2 Xpointer

Xpointer 用来设定 XML 文件内不同位置的链接和 HTML 中的内链接类似，但是，Xpointer 提供了 5 种不同的在 XML 文件内定位的方法，可将地址定位到相应的地方，功能上比 HTML 中的内链接更为强大。

绝对定位：root()——将地址定位到 XML 文件中的根元素位置。

属性名(x)——将地址定位到属性值为 x 的特定属性位置。

相对定位：child(x)——将地址定位到当前地址下的第 x 个子标记处。

child(x, y)——将地址定位到当前地址下第 x 个标记名为 y 的标记处。

child(x).child(y, z)——将地址定位到当前地址下的第 x 个标记处，然后将这个标记作为当前地址，再找出该标记下的第 y 个标记名为 Z 的控制标记。

范围定位：span(Xpointer1, Xpointer2)——选择所有的在第一个 Xpointer 开始和第二个 Xpointer 结束之间的内容。其中 Xpointer1、Xpointer2 表示其他的定位方法。

属性定位：attr(x)——找出第一个具有 x 属性的标记。

字符串定位：搜寻特定的字符串，然后将地址定位到特定的字符串处。

合并定位：将上述的定位方法进行组合，产生更多的功能。各种定位方法之间用“.”符号分隔。

13.4.3 DOM (Document Object Model)

早在 HTML 中，DOM 就有应用了。DOM 可以看作是一种 ActiveX 对象，它绑定封装了一部分文件存取 API（应用程序编程接口），使用户能够使用脚本语言（VBScript、JavaScript

等)来调用 DOM 对象,达到存取、操作文件内容的目的。以前在 HTML 中,我们利用 DOM 来创建动态网页,在 XML 文件中,我们同样可以利用 DOM 来创建动态网页,并且 DOM 可以用来加载 XML 文件,并加以解析、截取和操作 XML 文件中的信息。

IE 5 支持 XML 和 DOM 的结合应用,提供了四种 DOM 对象:XMLDOMDocument、XMLDOMNode、XMLDOMNodeList、XMLDOMNameNodeMap。这些 DOM 对象提供了很多方法和属性,用法同一般的 ActiveX 对象也没有什么区别。具体的属性、方法可以参阅微软的 Web 站点。

在下面的例子中,这个 HTML 文件用到了 XMLDOM 对象,在这个 HTML 文件中使用 JavaScript 创建了 DOM 对象,然后调用 DOM 对象,将我们前几期讲 XSL 时的 XML 文件、XSL 文件的例子读入,然后将 XML 文件依照 XSL 样式表的设定显示在浏览器中。

```
<html>
<head>
<title> DOM应用举例 </title>
<SCRIPT LANGUAGE="JavaScript" FOR="window" EVENT="onload">
var xmlfile=new ActiveXObject ("Microsoft.xmlDOM");
xmlfile.load ("2.xml");
var xslfile=new ActiveXObject ("Microsoft.xmlDOM");
xslfile.load ("first.xsl");
document.all.item ( "DOM 应用 " ) .innerHTML=xmlfile.transformNode
(xslfile.documentElement);
</SCRIPT>
</head>
<body>
<DIV id="DOM应用"> </DIV>
</body>
</html>
```

13.4.4 Namespaces

考虑一下,当我们建立 XML 应用的时候,会为具体的行业应用创建特定的 DTD,规定可用的元素。有时会出现下面情况——两个同名的元素在不同的地方可能会有不同的含义。例如,我们定义 <title> 这个标记,在书店应用中,这个标记中的字符含义是书的名称,而

在人事部门中，这个标记中的字符含义却是人的称谓、头衔。如果我们写一个包含书名、作者、作者身份等信息的 XML 文件，将在书名和作者身份两个地方用到同样的〈title〉标记，但它们有不同的含义，计算机程序无法分辨哪一个是书名，哪一个是作者身份。这样就给我们的自动化处理带来了问题，这就是 Namespaces 要解决的问题。Namespaces 的概念非常直接——对于每一套特定应用的 DTD，给它一个独一无二的标志来代表，如果在 XML 文件中使用 DTD 中定义的元素，需将 DTD 的标志和元素名、属性连在一起使用，相当于指明了元素来自什么地方，这就不会和其他同名元素混淆（像我们的电话号码，两个城市可能存在相同的号码，但是我们在前面用区号将它限定了，一个地方的区号在一个国家中是独一无二的）。在 XML 中，采用现成的、在全球范围唯一的“域名”作为 Namespaces，即用 URL 作为 XML 的 Namespaces。前面我们学习 XSL 的时候，实际上就接触到了 Namespaces，XSL 文件中的标记名称前面都有一个“xsl:”，实际上这就是 XSL 中元素的 Namespaces。声明语句就是：<xsl:stylesheet xmlns:xsl=“http://www.w3.org/TR.WD-xsl”>。其中，声明中的 xsl 称为前置字符串，在文件中引用元素时要加上前置字符串，如：〈xsl:templd〉。

```
<?xml version="1.0" encoding="GB2312"?>
<c:客户名单 xmlns:c="http://www.aaa.com/custom.dtd">
xmlns:职工="http://www.aaa.com/employee.dtd">
  <c:客户>
    <c:姓名>张三</c:姓名>
    <c:电话>028-6666666</电话>
    <c:接待人>
      <职工:姓名>李四</职工:姓名>
      <职工:电话>5555555</职工:电话>
    </c:接待人>
  </c:客户>
  .....
```

在这个例子的前面定义了两个 Namespaces——c:和职工:。在应用元素时，前面都加了特定的 Namespaces。那么应用程序在读到同名元素，如：〈姓名〉、〈电话〉时，就能够区分哪一个是客户姓名、电话，而哪一个又是本单位职工的姓名、电话了。

利用 Namespaces，我们还可以在 XML 文件中直接利用 HTML 的标记，不使用 Xlink 或 Xpionter，也让 XML 文件具有超级链接、显示图片的功能。在使用 HTML 标记之前，必须

声明它的 Namespaces。下面这个例子就是 XML 和 HTML 的混合使用。

```
<?xml version="1.0" encoding="GB2312" ?>
<?xml-stylesheet href="first.css" type="text/css" ?>
<data xmlns:HTML="http://www.w3.org/TR/XHTML">
  <book>
    <title> XML入门精解 </title>
    <HTML:a href="mailto:lionliao@yeah.net">
    <author> 作者: 张三 </author>
  </HTML:a>
  <picture>
    <HTML:img src="zhangsan.jpg" width="80" height="80"> </HTML:img>
  </picture>
  <price unit="人民币"> 价格: $20.00 </price>
  <content>
    <HTML:a href="http://www.cbi.com"> 点击查看主要内容 </HTML:a>
  </content>
</book>
</data>
```

注意：在 XML 中使用 HTML 时要严格遵守 XML 的语法规则，元素必须正确关闭。

通过 CSS 或 XSL 可将这个 XML 文件显示出来，如图 1 所示。

13.4.5 TML

XHTML 最早叫 HTML in XML，就是把过去用 SGML 定义的 HTML，用 XML 来重新定义（不要忘了，XML 是一种定义语言的语言）。实际上，XHTML 中的标记基本上还是 HTML 4.0 中的那些标记，各种标记、属性的用法基本不变。只不过因为它是通过 XML 定义的，所以必须严格遵守 XML 的规定，不像过去那样随便。那为什么要发展 XHTML 呢？

现在，手机上网、信息家电等炒得如火如荼，好象什么都应该连到因特网上。XHTML 正是适应这种潮流而出现的。大家知道，现在的 HTML 越来越复杂，而且存在大量的不规范 HTML 网页（并不是能够在 IE、Netscape 正确显示就算是规范的网页），而浏览器为了能够适应这种情况，包容了大量五花八门的 HTML 网页，已经变得非常臃肿（这就是虽然你的 HTML 语法错误，有时还是能够正确显示出来的原因，设计浏览器的工程师绞尽脑汁来适应、

包容错误)。对于 PC 来说,这不算什么,因为 PC 的性能已经得到了很大程度的提升,而对于那些掌中设备、信息家电,可没有这么多存储空间可用。XML 标准简单但是非常严格,主要目的就是减轻解析器、浏览器的开发负担以及这些软件的体积。一般的 XML 解析器的体积不过几百 KB。根据 XML 标准定义出来的 XHTML 当然继承了 XML 的特性,同时也保留了 HTML 在表现形式上的优越性,它摒弃了 HTML 中的“不干净”代码,提供了良好的可伸缩性,可大可小。

XHTML 通过将过去的 HTML 功能,按照用户的需要和浏览器的能力,划分为多个模块,每一组模块仅支持部分 HTML 标记。针对于特定模块开发的解析器、浏览器显然比大而全的解析器、浏览器体积要小得多。这就达到了“可小”的目的。比如说:掌上电脑屏幕本身很小,显然它浏览网页的时候不需要使用 HTML 中的“Frame”功能,那么针对这种设备,有它专门的一套适用的、不包含“Frame”的标记。当然,每个模块都需要有它们的专用 DTD 来声明可以使用哪些标记。现在 W3C 已经定义好了几种专用的 DTD。

“可大”呢?实际上就是利用 XML 的特点,XML 是什么?可扩展标记语言呀!以前,HTML 中的标记是已经定义好了的,是不能够改变,只能拿来使用,而 XHTML 就可以像 XML 一样,能够自定义标记。

使用 XHTML 和使用 HTML 基本上一样,不过要注意的是,我们再不能偷懒了。XHTML 也像 XML 一样,首先必须是 Well-Formed。规则如下:

1) 标记必许正确结束。

2) 标记与标记之间不允许交叉嵌套。像以前的“<i>斜体粗斜体</i>”这样的语句是不行的,而必须写成“<i>斜体粗斜体</i>”才合格。

3) 空元素(开始标记与结束标记间无内容)必须按 XML 的规定写成<元素名/>。

4) 属性值必须用“ ”号括起来,像以前的“<td width=100>”的语句必须改写成“<td width=“100”>”。

5) 属性都要赋值。以前有些标记的属性如果不赋值,它就取缺省值,在 XHTML 中,必须明确地给它赋值。

6) 标记名称、属性名称都用小写字母。

7) 使用正确的根元素加上 Namespaces。

8) <head><body>不能够省略。<title>必须是<head>中出现的第一个子元素。

9) 来 HTML 网页中的 VBScript、JavaScript、样式表区域,必须像 XML 的 CDATA 区一样包装起来,如:

```

<script language="JavaScript">
  <![CDATA[
    .....
    if (i < 3 &&&& .....
  )) ]
</script>

```

满足了上述条件，并不表示 XHTML 就是最好的，而仅仅是格式正确的 XHTML 文件，需要正确地引用 DTD，才成为真正正确的 XHTML。W3C 已经定义了三种 XHTML 的 DTD 供大家使用，它们分别是 Strict、Transitional、Frameset。从现在开始，写 HTML 网页时就需要按照 XHTML 的规定来写，并且最好不要用 font 之类的标记，尽量使用 CSS 来表现你的 HTML 网页。等到 XHTML、XML 普及了，你就会暗笑了，因为你实现网页转换是非常容易的。

13.5 XML DOM

1. 文档对象模型 (DOM)

DOM 是 HTML 和 XML 文档的编程基础，它定义了处理执行文档的途径。编程者可以使用 DOM 增加文档、定位文档结构、添加修改删除文档元素。W3C 的重要目标是把利用 DOM 提供一个使用于多个平台的编程接口。W3C DOM 被设计成适合多个平台，可使用任意编程语言实现的方法。

2. 节点接口

XML parser 用来装载 XML 文档到缓存中，文档装载时，可以使用 DOM 进行检索和处理。DOM 采用树形结构表示 XML 文档，文档元素是树的最高阶层，该元素有一个或多个孩子节点用来表示树的分枝。

节点接口程序通常用来读和写 XML 节点树中的个别元素，文档元素的孩子节点属性可以用来构造个别元素节点。XML parser 用来证明 Web 中的 DOM 支持遍历节点树的所有函数，并可通过它们访问节点及其属性、插入删除节点、转换节点树到 XML 中。所有 Microsoft XML parser 函数得到 W3C XML DOM 的正式推荐，除了 load 和 loadXML 函数（正式的 DOM 不包括标准函数 loading XML 文档）。有 13 个节点类型被 Microsoft XML parser 支持，下面列出常用节点类型：

```
Document type <!DOCTYPE food SYSTEM "food.dtd">
```

```
Processing instruction <?xml version="1.0"?>
Element <drink type="beer">Carlsberg</drink>
Attribute type="beer"
Text Carlsberg
```

3. 使用 XML parser

为了更加熟练的处理 XML 文档，必须使用 XML parser。Microsoft XML parser 是 IIS5.0 所带的一个 COM 组件，一旦安装了 IIS5.0，parser 可以利用 HTML 文档和 ASP 文件中的脚本。

Microsoft XMLDOM parser 支持以下编程模式：

- 支持 JavaScript, VBScript, Perl, VB, Java, C++等等
- 支持 W3C XML 1.0 和 XML DOM
- 支持 DTD 和 validation

如果使用 IE5.0 中的 JavaScript，可以使用下面的 XML 文档对象：

```
var xmlDoc = new ActiveXObject( "Microsoft.XMLDOM" )
```

如果使用 VBScript，可以使用下面的 XML 文档对象：

```
set xmlDoc = CreateObject( "Microsoft.XMLDOM" )
```

如果使用 ASP，可以使用下面的 XML 文档对象：

```
set xmlDoc = Server.CreateObject( "Microsoft.XMLDOM" )
```

4. 装载一个 XML 文件到 parser 中

下面的代码装载存在的 XML 文档进入 XML parser:

```
<script language="JavaScript">
var xmlDoc = new ActiveXObject ( "Microsoft.XMLDOM" )
xmlDoc.async="false"
xmlDoc.load ( "note.xml" )
// ..... processing the document goes here
</script>
```

第一行脚本增加了一个 Microsoft XML parser 实例，第三行装载名为“note.xml”的 XML 文档进入 parser 中。第二行保证文档装载完成以后 parser 进行下一步工作。

5. parseError 对象

打开 XML 文档时, XML Parser 产生错误代码, 并存在 parseError 对象中, 包括错误代码、错误文本和错误行号, 等信息。

6. 文件错误

下面的例子将试图装载一个不存在的文件, 然后产生相应的错误代码:

```
var xmlDoc = new ActiveXObject ("Microsoft.XMLDOM")
xmlDoc.async="false"
xmlDoc.load ("ksdjf.xml")

document.write ("<br>Error Code: ")
document.write (xmlDoc.parseError.errorCode)
document.write ("<br>Error Reason: ")
document.write (xmlDoc.parseError.reason)
document.write ("<br>Error Line: ")
document.write (xmlDoc.parseError.line)
```

7. XML 错误

下面使用不正确的格式装载 XML 文档,

```
var xmlDoc = new ActiveXObject ("Microsoft.XMLDOM")
xmlDoc.async="false"
xmlDoc.load ("note_error.xml")

document.write ("<br>Error Code: ")
document.write (xmlDoc.parseError.errorCode)
document.write ("<br>Error Reason: ")
document.write (xmlDoc.parseError.reason)
document.write ("<br>Error Line: ")
document.write (xmlDoc.parseError.line)
```

8. parseError 属性

属性描述:

errorCode——返回长整型错误代码

reason——返回字符串型错误原因

line——返回长整型错误行号
 linePos——返回长整型错误行号位置
 srcText——返回字符串型产生错误原因
 url——返回 url 装载文档指针
 filePos——返回长整型错误文件位置

9. 遍历节点树

一种通用的析取 XML 文档的方法是遍历节点树和它的元素值。下面是使用 VBScript 写的遍历节点树的程序代码：

```
set xmlDoc=CreateObject ("Microsoft.XMLDOM")
xmlDoc.async="false"
xmlDoc.load ("note.xml")

for each x in xmlDoc.documentElement.childNodes
document.write (x.nodeName)
document.write (" : ")
document.write (x.text)
next
```

10. 为 XML 文件提供 HTML 格式

XML 的一个优点是把 HTML 文档和它的数据分离开。通过使用浏览器中的 XML parser, HTML 页面可以被构造成静态文档,通过 JavaScript 提供动态数据。下面的例子使用 JavaScript 读取 XML 文档,写 XML 数据成 HTML 元素:

```
var xmlDoc = new ActiveXObject ("Microsoft.XMLDOM")
xmlDoc.async="false"
xmlDoc.load ("note.xml")

nodes = xmlDoc.documentElement.childNodes

to.innerText = nodes.item (0) .text
from .innerText = nodes.item (1) .text
header.innerText = nodes.item (2) .text
body.innerText = nodes.item (3) .text
```

11. 通过名称访问 XML 元素

下面的例子使用 JavaScript 读取 XML 文档，写 XML 数据成 HTML 元素：

```
var xmlDoc = new ActiveXObject ("Microsoft.XMLDOM")
xmlDoc.async="false"
xmlDoc.load ("note.xml")

document.write (xmlDoc.getElementsByTagName ("from") .item (0) .text)
```

12. 装载纯 XML 文本进入 parser

下面的代码装载文本字符串进入 XML parser：

```
<script language="JavaScript">
var text="<note>"
text=text+"<to>Tove</to><from>Jani</from>"
text=text+"<heading>Reminder</heading>"
text=text+"<body>Don't forget me this weekend!</body>"
text=text+"</note>"
var xmlDoc = new ActiveXObject ("Microsoft.XMLDOM")
xmlDoc.async="false"
xmlDoc.loadXML (text)
// ..... processing the document goes here
</script>
```

13. 装载 XML 进入 Parser

```
<html>
<body>

<script language="javascript">
var xmlDoc = new ActiveXObject ("Microsoft.XMLDOM")
xmlDoc.async="false"
xmlDoc.load ("note.xml")

document.write ("The first XML element in the file contains: ")

document.write (xmlDoc.documentElement.childNodes.item (0) .text)
```

```
</script>
```

```
</body>
```

```
</html>
```

遍历XML节点树:

```
<html>
```

```
<body>
```

```
<script language="VBScript">
```

```
txt="<h1>Traversing the node tree</h1>"
```

```
document.write (txt)
```

```
set xmlDoc=CreateObject ("Microsoft.XMLDOM")
```

```
xmlDoc.async="false"
```

```
xmlDoc.load ("note.xml")
```

```
for each x in xmlDoc.documentElement.childNodes
```

```
document.write ("<b>" & x.nodeName & "</b>")
```

```
document.write (": ")
```

```
document.write (x.text)
```

```
document.write ("<br>")
```

```
next
```

```
</script>
```

```
</body>
```

```
</html>
```

装载XML 进入 HTML

```
<html>
```

```
<head>
```

```
<script language="JavaScript"
```

```
for="window" event="onload">
```

```
var xmlDoc = new ActiveXObject ("Microsoft.XMLDOM")
```

```
xmlDoc.async="false"
```

```
xmlDoc.load ("note.xml")
```

```
nodes = xmlDoc.documentElement.childNodes
```



```
to.innerText = nodes.item (0) .text
from.innerText = nodes.item (1) .text
header.innerText = nodes.item (2) .text
body.innerText = nodes.item (3) .text

</script>

<title>HTML using XML data</title>
</head>

<body bgcolor="yellow">
<h1>Refsnes Data Internal Note</h1>

<b>To: </b><span id="to"></span>

<br>
<b>From: </b><span id="from"></span>

<hr>
<b><span id="header"></span></b>

<hr>
<span id="body"></span>

</body>
</html>
```

13.6 thon和XML

在许多情况下,Python 是使用 XML 文档的理想语言。像 Perl、REBOL、REXX 和 TCL 一样,它是一种灵活的脚本语言,并且有强大的文本操作能力。而且,XML 文档除了编码大多数类型的文本文件(或流文件),通常还编码大量复杂的数据结构。文本处理中常见的“读取几行,并将它们与一些规则表达式比较”样式通常不能很好地适合对 XML 进行彻底语法分析和处理。幸好,Python(与大多数其他语言相比)不仅有直接处理复杂数据结构的方法(通常使用类和属性),还有许多 XML 相关的模块可以帮助语法分析、处理和生成 XML。

关于 XML,要记住一个总体概念:可以验证或非验证方式处理 XML 文档。在以前的

处理类型中，读取 XML 文档之前，必须先读取“文档类型定义”（DTD）。这种情况下，处理将总体计算 XML 文档的简单句型规则，还将计算 DTD 的特定语法约束。大多数情况下，使用非验证处理就可以了（通常运行更快，更适合程序）——我们相信文档创建者遵循文档范围的规则。在下面讨论的大多数模块都是非验证型；如果存在验证选项，则描述将指出。

13.6.1 主要模块和包

中心资源库（Vaults of Parnassus）最近已成为查找 Python 资源的标准方法。用户可以在那个站点上找到一些很有用的资源。特别地，可以在资源库中找到 PyXML 发行版，它是 tar 文件和 Win32 形式的安装程序。

XML-SIG 的成员执行了许多或大部分维护 Python 一部分 XML 工具的任务。与其他 Python SIG 一样，XML-SIG 要维护邮件发送列表、列表档案、有用的参考人权、文档、标准包和其他资源。阅读了本书中的概述后，最好从 XML-SIG Web 页面入手。

根据本书讲述的特定重点，XML-SIG 维护了 PyXML 发行版。这个包包含了许多本书中阐述的模块，一些“入门”文档，一些演示代码和其他一些 XML-SIG 决定放入该发行版的东西。给定的包也许不会总是包含每个独立模块或工具的最新版本，但下载 PyXML 发行版是个好主意。以后，可以随时添加任何未包含的模块，或者已包含模块的新版本（以及许多 PyXML 发行版提供的服务所未包含的模块）。

1. 模块：xmllib 模块（标准）

Python 1.5.* 带有模块 [xmllib]。Python 1.6 也许结合了更多 XML-SIG 的成就，但它仍是测试版。[xmllib] 是一个非验证的低级语法分析器。[xmllib] 的工作方式是用应用程序覆盖 XMLParser 类，并提供处理文档元素（如特定或类属标记，或字符实体）的方法。

作为正在使用的 [xmllib] 示例，PyXML 发行版包括一个叫做“quotations.dtd”的 DTD，以及这个 DTD 的文档“sample.xml”（请参阅参考资料，以获取本文中提到的文件的档案文件）。以下的代码显示了“sample.xml”中每段引言的前几行，并生成了非常简单的未知标记和实体的 ASCII 指示符。经过分析的文本作为连续流来处理，所使用的任何累加器都由程序员负责（如标记中的字符串（#PCDATA），或所遇到的标记的列表/词典）。

例 1: xmllib 的代码

```
#----- try_xmllib.py -----#
import xmllib, string

class QuotationParser (xmllib.XMLParser):
```

```
        """Crude xmllob extractor for quotations.dtd
document"""

        def _init_ (self) :
            xmllob.XMLParser._init_ (self)
            self.thisquote = ''          #
quotation accumulator

        def handle_data (self, data) :
            self.thisquote = self.thisquote +
data

        def syntax_error (self, message) : pass

        def start_quotations (self, attrs) : # top level tag
            print '--- Begin Document ---'

        def start_quotation (self, attrs) :
            print 'QUOTATION:'

        def end_quotation (self) :
            print string.join (string.split (se
lf.thisquote[:230])) + '...',
            print ' {'+str (len (self.thisquo
te)) + ' bytes) \n'

            self.thisquote = ''

        def unknown_starttag (self, tag, attrs) :
            self.thisquote = self.thisquote + '{'

        def unknown_endtag (self, tag) :
            self.thisquote = self.thisquote + '}'

        def unknown_charref (self, ref) :
            self.thisquote = self.thisquote + '?'

        def unknown_entityref (self, ref) :
            self.thisquote = self.thisquote + '#'
```

```

if __name__ == '__main__':
    parser = QuotationParser ()
    for c in open ("sample.xml") .read () :
        parser.feed (c)
    parser.close ()

```

其他语法分析模块。PyXML 发行版包含了几个具有各种功能的附加语法分析模块。提供这些模块是为了对基本 [xmllib] 模块做一些改进。

[pyexpat] 是 GPL 方式的 XML 语法分析器工具箱“expat”的封装程序。“expat”是用 C 语言写的库，这就意味着任何想要利用它的语言都可以使用它。“expat”是非验证型，因此它比原来的 Python 语法分析器快很多。[sgmlop] 的目的与 [pyexpat] 相同。它也是非验证型，而且也用 C 语言编写。[pyexpat] 可以作为 MacOS 二进制使用，[sgmlop] 可以当作 Win32 二进制使用；但如果您需要使用不同的平台，那么就要用 C 编译器为您自己的平台构建模块。

[xmlproc] 是 python 原有的语法分析器，它执行几乎完整的验证。如果需要验证型语法分析器，[xmlproc] 是 Python 当前唯一的选择。同样，[xmlproc] 提供其他语法分析器所不具备的各种高级和测试接口。

如果决定使用 XML 的简单 API (SAX)，它应该用于复杂的事物，因为其他大部分工具都构建在它之上，将为您完成许多语法分析器的分类工作。在 PyXML 发行版中，[xml.sax.drivers] 包含许多语法分析器的瘦封装程序，包括所有那些已讨论过的、名称形式为“drv_*.py”的语法分析器。但是，一般使用高级 SAX 设施访问驱动器，该设施自动选择系统上“最佳”的可用语法分析器：例选择语法分析器

```

#----- selecting the best parser -----#
from xml.sax.saxext import *
parser = XMLParserFactory.make_parser ()

```

2. 包：SAX

以上，我们已提到 SAX 会自动选择要使用的语法分析器；但 SAX 是什么？一个较好的答案是：“SAX (XML 的简单 API) 是 XML 语法分析器的公用语法分析器接口。它允许应用程序作者编写使用 XML 语法分析器的应用程序，但是它却独立于所使用的语法分析器。（将它看作 XML 的 JDBC。）”

SAX 如同它提供的语法分析器模块的 API，基本上是一个 XML 文档的顺序处理器。使用它的方法与 [xmllib] 示例极其相似，但更加抽象。定义语法分析器类，应用程序员将定义一个 “handler” 类，该类将注册所使用的语法分析器。必须定义四个 SAX 接口（每个接口都有几个方法）：DocumentHandler、DTDHandler、EntityResolver 和 ErrorHandler。已提供了所有这些接口的基类，但大多数情况下，最简单的方法是继承 “HandlerBase”，因为这个类继承了所有四个接口。可以不用考虑想要做什么。某些代码将帮助解释这一点；该样本执行与 [xmllib] 示例相同的任务。

例 2: SAX 的样本代码

```
#----- try_sax.py -----#
import string
from xml.sax import saxlib, saxexts

class QuotationHandler (saxlib.HandlerBase) :
    """Crude sax extractor for quotations.dtd
document"""

    def _init_ (self) :
        self.in_quote = 0
        self.thisquote = ''

    def startDocument (self) :
        print '--- Begin Document ---'

    def startElement (self, name, attrs) :
        if name == 'quotation':
            print 'QUOTATION:'
            self.in_quote = 1
        else:
            self.thisquote = self.thisquote + '{'

    def endElement (self, name) :
        if name == 'quotation':
            print string.join (string.split (sel
```

```

f.thisquote[:230])) + '...',

print ' ('+str (len (self.thisquote))

+' bytes) \n'

self.thisquote = ''
self.in_quote = 0
else:
self.thisquote = self.thisquote + '}'

def characters (self, ch, start, length) :
    if self.in_quote:
        self.thisquote = self.thisquote +
ch[start:start+length]

if __name__ == '__main__':
    parser = saxexts.XMLParserFactory.make_parser()
    handler = QuotationHandler ()
    parser.setDocumentHandler (handler)
    parser.parseFile (open ("sample.xml"))
    parser.close ()

```

与 [xmllib] 相比, 关于示例要注意两件小事: “`parseFile()`” “`/`” `parse()`” 方法处理整个流 / 字符串, 所以不必为语法分析器创建循环; 向 “`characters()`” 提供了大量数据, 自变量会指出数据的大小和位置以及传递的字符串。不要假设变量 “`ch`” 将以什么形式传送给 “`characters()`”。

3. 包: DOM

DOM 是一种 XML 文档的高级树型表示。该模型并非特定于 Python, 而是一种普通 XML 模型 (请参阅参考资料以获取进一步信息)。Python 的 DOM 包是针对 SAX 构建的, 并且包括在 PyXML 发行版中。由于篇幅关系, 没有将代码样本加到本文中, 但在 XML-SIG 的 “Python/XML HOWTO” 中给出了一个极好的总体描述。

“文档对象模型” (DOM) 为 XML 文档指定了树型表示。顶级文档实例是树的根, 它只有一个子代, 即顶级元素实例; 这个元素有表示内容和子元素的子节点, 他们也可以有子代。定义的函数允许随意遍历结果树, 访问元素和属性值, 插入和删除节点, 以及将树转换回 XML。

DOM 可以用于修改 XML 文档，因为可以创建一棵 DOM 树，通过添加新节点和来回移动子树来修改这棵树，然后生成一个新的 XML 文档作为输出。您也可以自己构造一棵 DOM 树，然后将它转换成 XML；用这种方法生成 XML 输出比仅将 `<tag1>...</tag1>` 写入文件的方法更灵活。

4. 包：Pyxie

[pyxie] 模块从 XML-SIG 构建到 PyXML 发行版之上，它为 XML 文档提供了附加的高级接口。[pyxie] 将完成两项基本操作：它将 XML 文档转换成一种更易于进行语法分析的基于行的格式；并且它提供了将 XML 文档当作可操作树处理的方法。[pyxie] 所使用的基于行的 PYX 格式是独立于语言的，其工具适用于几种语言。总之，文档的 PYX 表示与其 XML 表示相比，更易于使用常见的基于行的文本处理工具进行处理，如 `grep`、`sed`、`awk`、`bash`、`perl`，或标准 python 模块，如 [string] 和 [re]。根据结果，从 XML 转换到 PYX 可能节省许多工作。

[pyxie] 将 XML 文档当作树处理的概念与 DOM 中的思路相似。由于 DOM 标准得到许多编程语言的广泛支持，那么如果 XML 文档的树型表示是必需的，大多数程序员会使用 DOM 标准而非 [pyxie]。

5. 模块：XML 语法分析器

“XML 语法分析器”这个叫法太笼统，也许还不太确切，实际上它是一种比较旧的工具，用于检查 XML 文档是否符合句法以及其结构是否完好（但对于 DTD 无效）。一个附加的实用程序类在进行检查时会产生一些小麻烦，它会让 HTML 文档通过检查（即使那些文档没有 XML 必需的结束标记）。这个模块的适用范围并不能覆盖 PyXML 发行版中的所有模块。但如果只想验证一些 XML 文档，那么设置和运行 XML 语法分析器还是很容易的。如果从命令行运行，则该模块将在 STDIN 上检查 XML 文档，甚至不用将它导入程序。这是最简单的做法。

6. XML_OBJECTS 0.1

如同其他高级工具，xml_objects 构建在 SAX 之上。构建 xml_objects 的目的是将 XML 文档转换成一个二维网格表示，从而更易于在关系数据库中存储。

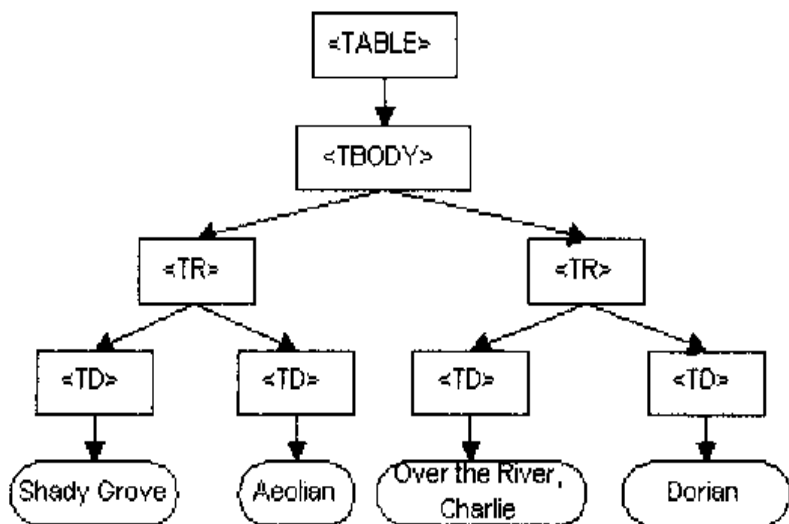
13.6.2 文档对象模型

xml.dom 模块对于 Python 程序员来说，可能是使用 XML 文档时功能最强大的工具。不幸的是，XML-SIG 提供的文档目前来说还比较少。W3C 语言无关的 DOM 规范填补了

这方面的部分空白。但 Python 程序员最好有一个特定于 Python 语言的 DOM 的快速入门指南。本文旨在提供这样一个指南。在上一篇专栏文章中，某些样本中使用了样本 quotations.dtd 文件，并且这些文件可以与本文中的代码样本档案文件一起使用。

有必要了解 DOM 的确切含义。这方面，正式解释非常好：“文档对象模型”是平台无关和语言无关的接口，它允许程序和脚本动态访问和更新文档的内容、结构和样式。可以进一步处理文档，而处理的结果也可以合并到已显示的页面中。

DOM 将 XML 文档转换成树或森林表示。万维网联盟（W3C）规范给出了一个 HTML 表的 DOM 版本作为例子。



如上图所示，DOM 从一个更加抽象的角度定义了一组可以遍历、修剪、改组、输出和操作树的方法，而这种方法要比 XML 文档的线性表示更为便利。

13.6.3 将 HTML 转换成 XML

有效的 HTML 几乎就是有效的 XML，但又不完全相同。这里有两个主要的差异，XML 标记是区分大小写的，并且所有 XML 标记都需要一个显式的结束符号（作为结束标记，而这对于某些 HTML 标记是可选的；例如：）。使用 xml.dom 的一个简单示例就是使用 HtmlBuilder() 类将 HTML 转换成 XML。

例 3:

```

try_dom1.py

"""Convert a valid HTML document to XML

```



```
    USAGE: python try_dom1.py < infile.html > outfile.xml
    """

    import sys
    from xml.dom import core
    from xml.dom.html_builder import HtmlBuilder

    # Construct an HtmlBuilder object and feed the data to it
    b = HtmlBuilder ()
    b.feed (sys.stdin.read ())

    # Get the newly-constructed document object
    doc = b.document

    # Output it as XML
    print doc.toxml ()
```

`HtmlBuilder()` 类很容易实现它继承的部分基本 `xml.dom.builder` 模板的功能，它的源码值得研究。然而，即使我们自己实现了模板功能，DOM 程序的轮廓还是相似的。在一般情况下，我们将用一些方法构建一个 DOM 实例，然后对该实例进行操作。DOM 实例的 `.toxml()` 方法是一种生成 DOM 实例的字符串表示的简单方法（在以上的情况中，只要在生成后将它打印出来）。

13.6.4 将 Python 对象转换成 XML

Python 程序员可以通过将任意 Python 对象导出为 XML 实例来实现相当多的功能和通用性。这就允许我们以习惯的方式来处理 Python 对象，并且可以选择最终是否使用实例属性作为生成 XML 中的标记。只需要几行（从 `building.py` 示例派生出），我们就可以将 Python“原生”对象转换成 DOM 对象，并对包含对象的那些属性执行递归处理。

例 4:

```
try_dom2.py

"""Build a DOM instance from scratch, write it to XML

    USAGE: python try_dom2.py > outfile.xml
    """

import types
```

```

from xml.dom import core
from xml.dom.builder import Builder

# Recursive function to build DOM instance from Python instance
def object_convert (builder, inst) :
    # Put entire object inside an elem w/ same name as the class.
    builder.startElement (inst._class__.__name_)

    for attr in inst._dict_.keys () :
        if attr[0] == '_':      # Skip internal attributes
            continue
        value = getattr (inst, attr)
        if type (value) == types.InstanceType:
            # Recursively process subobjects
            object_convert (builder, value)
        else:
            # Convert anything else to string, put it in an element
            builder.startElement (attr)
            builder.text (str (value))
            builder.endElement (attr)

    builder.endElement (inst._class__.__name_)

if __name__ == '__main__':
    # Create container classes
    class quotations: pass
    class quotation: pass

    # Create an instance, fill it with hierarchy of attributes
    inst = quotations ()
    inst.title = "Quotations file (not quotations.dtd conformant)"
    inst.quot1 = quot1 = quotation ()
    quot1.text = """"is not a quine" is not a quine' is a quine""
    quot1.source = "Joshua Shagam, kuro5hin.org"
    inst.quot2 = quot2 = quotation ()
    quot2.text = "Python is not a democracy. Voting doesn't help. "+\

```

```
        "Crying may..."

    quot2.source = "Guido van Rossum, comp.lang.python"

    # Create the DOM Builder
    builder = Builder ()
    object_convert (builder, inst)

    print builder.document.toxml ()
```

函数 `object_convert()` 有一些限制。例如，不可能用以上的过程生成符合 XML 文档的 `quotations.dtd`：#PCDATA 文本不能直接放到 `quotation` 类中，而只能放到类的属性中（如 `.text`）。一个简单的变通方法就是让 `object_convert()` 以特殊方式处理一个带有名称的属性，例如 `.PCDATA`。可以用各种方法使对 DOM 的转换变得更巧妙，但该方法的妙处在于我们可以从整个 Python 对象开始，以简明的方式将它们转换成 XML 文档。

还应值得注意的是在生成的 XML 文档中，处于同一个级别的元素没有什么明显的顺序关系。例如，在作者的系统中使用特定版本的 Python，源码中定义的第二个 `quotation` 在输出中却第一个出现。但这种顺序关系在不同的版本和系统之间会改变。Python 对象的属性并不是按固定顺序排列的，因此这种特性就具有意义。对于与数据库系统相关的数据，我们希望它们具有这种特性，但是对于标记为 XML 的文章却显然不希望具有这种特性（除非我们想要更新 William Burroughs 的“cut-up”方法）。

13.6.5 将 XML 文档转换成 Python 对象

从 XML 文档生成 Python 对象就像其逆向过程一样简单。在多数情况下，用 `xml.dom` 方法就可以了。但在某些情况下，最好使用与处理所有“类属”Python 对象相同的技术来处理从 XML 文档生成的对象。例如，在以下的代码中，函数 `pyobj_printer()` 也许已经是用来处理任意 Python 对象的函数。

例 5:

```
try_dom3.py

"""Read in a DOM instance, convert it to a Python object
"""

from xml.dom.utils import FileReader
```

```

class PyObject: pass

def pyobj_printer (py_obj, level=0) :
    """Return a "deep" string description of a Python object"""
    from string import join, split
    import types
    descript = ''
    for memname in dir (py_obj) :
        member = getattr (py_obj, memname)
        if type (member) == types.InstanceType:
            descript = descript + (' '*level) + '{'+memname+'}\n'
            descript = descript + pyobj_printer (member, level+3)
        elif type (member) == types.ListType:
            descript = descript + (' '*level) + '['+memname+']\n'
            for i in range (len (member)) :
                descript = descript+ (' '*level) +str (i+1) +': '+ \
                    pyobj_printer (member[i], level+3)
        else:
            descript = descript + memname+'='
            descript = descript + join (split (str (member) [:50])) + '...\n'
    return descript

def pyobj_from_dom (dom_node) :
    """Converts a DOM tree to a "native" Python object"""
    py_obj = PyObject ()
    py_obj.PCDATA = ''
    for node in dom_node.get_childNodes () :
        if node.name == '#text':
            py_obj.PCDATA = py_obj.PCDATA + node.value
        elif hasattr (py_obj, node.name) :
            getattr (py_obj, node.name) .append (pyobj_from_dom (node))
        else:
            setattr (py_obj, node.name, [pyobj_from_dom (node) ])
    return py_obj

```

```
# Main test
dom_obj = FileReader ("quotes.xml").document
py_obj = pyobj_from_dom (dom_obj)
if __name__ == "__main__":
    print pyobj_printer (py_obj)
```

这里的关注焦点应该是函数 `pyobj_from_dom()`，特别是起实际作用的 `xml.dom` 方法 `.get_childNodes()`。在 `pyobj_from_dom()` 中，我们直接抽取标记之间的所有文本，将它放到保留属性 `.PCDATA` 中。对于任何遇到的嵌套标记，我们创建一个新属性，其名称与标记匹配，并将一个列表分配给该属性，这样就可以潜在地包含在在父代块中多次出现的标记。当然，使用列表要维护在 XML 文档中遇到的标记的顺序。

除了使用旧的 `pyobj_printer()` 类属函数（或者更复杂和健壮的函数）之外，我们可以使用正常的属性记号来访问 `py_obj` 的元素。

13.6.6 Python 交互式会话

例 6:

```
>>> from try_dom3 import *
>>> py_obj.quotations[0].quotation[3].source[0].PCDATA
'Guido van Rossum'
```

13.6.7 重新安排 DOM 树

DOM 的一大优点是它可以让程序员以非线性方式对 XML 文档进行操作。由相匹配的开/关标记括起的每一块都只是 DOM 树中的一个“节点”。当以类似于列表的方式维护节点以保留顺序信息时，则顺序并没有什么特殊之处，也并非不可改变。我们可以轻易地剪下某个节点，嫁接到 DOM 树的另一个位置（如果 DTD 允许，甚至嫁接到另一层上）。或者添加新的节点、删除现有节点，等等。

例 7:

```
try_dom4.py
"""Manipulate the arrangement of nodes in a DOM object
```

```

"""
from try_dom3 import *

#-- Var 'doc' will hold the single <quotations> "trunk"
doc = dom_obj.getChildNodes () [0]

#-- Pull off all the nodes into a Python list
# (each node is a <quotation> block, or a whitespace text node)
nodes = []
while 1:
    try: node = doc.removeChild (doc.getChildNodes () [0])
    except: break
    nodes.append (node)

#-- Reverse the order of the quotations using a list method
# (we could also perform more complicated operations on the list:
# delete elements, add new ones, sort on complex criteria, etc.)
nodes.reverse ()

#-- Fill 'doc' back up with our rearranged nodes
for node in nodes:
    # if second arg is None, insert is to end of list
    doc.insertBefore (node, None)

#-- Output the manipulated DOM
print dom_obj.toxml ()

```

如果我们将 XML 文档只看作一个文本文件，或者使用一个面向序列的模块（如 `xml.lib` 或 `xml.sax`），那么在以上几行中执行对 `quotation` 节点的重新安排操作将引出一个值得考虑的问题。然而如果使用 DOM，则问题就如同对 Python 列表执行的任何其他操作一样简单。

13.7 Python和XML的结合

有许多技术和工具可以用 Python 处理 XML 文档。不过，现有的大多数 XML/Python 工具共有的一个问题是，它们主要是以 XML 为中心而不是以 Python 为中心。特定的构造

和编码技术感觉在某个编程语言中很“自然”，而其他往往感觉像是从其他领域导入的。但在理想环境中，所有构造都天生适合各自的领域，并且各个领域无缝地合并在一起。如果是这样，程序员就可以变得非常有创造性，而不只停留在让它工作就可以了。

Python 是一种语言，带有灵活的对象系统和一系列丰富的内置类型。Python 的这种丰富性对于该项目有利有弊。一方面，拥有以 Python 表示的大量本机设施使得大量 XML 结构表示起来更容易。另一方面，Python 的这些本机类型和结构在许多情况下也让人担心能否以 XML 表示本机 Python 对象。由于 XML 和 Python 之间存在不对称性，项目至少在最初时，包含两个单独的模块：xml_pickle 和 xml_objectify，前者用于以 XML 表示的任意 Python 对象，后者用于将 XML 文档“本机”表示为 Python 对象。

下面我们来了解一些 xml_pickle 和 xml_objectify 的一些基本情况。

13.7.1 xml pickle

Python 的标准 pickle 模块已经提供了将 Python 对象串行化的简便方法，这种方法对于持久存储器或在网络上进行传输很有帮助。但在某些情况下，希望对带有一些不由 pickle 拥有的特性的格式执行串行化。即，格式：

- 1) 是人类可读的
- 2) 可以进行语法分析、控制，并且其对象由非 Python 语言导入
- 3) 支持已存储串行化对象的确认

xml_pickle 在维护与 pickle 接口兼容性的同时提供了这些特性。不过，xml_pickle 不是 pickle 通常意义上的替代，因为 pickle 保留了其自身的一些优点，例如较快的操作（特别是通过 cPickle 时）以及更为紧凑的对象表示。尽管 xml_pickle 的接口与 pickle 的接口几乎相同，但还是有必要为那些不熟悉 Python 或 pickle 的人说明一下（非常简单）xml_pickle 的用法。我们来分析以下代码：

```
import xml_pickle # import the module

# declare some classes to hold some attributes
class MyClass1: pass
class MyClass2: pass

# create a class instance, and add some basic data members to it
o = MyClass1 ()
o.num = 37
```

```

o.str = "Hello World"
o.lst = [1, 3.5, 2, 4+7j]

# create an instance of a different class, add some members
o2 = MyClass2 ()
o2.tup = ("x", "y", "z")
o2.num = 2+2j
o2.dct = { "this": "that", "spam": "eggs", 3.14: "about PI" }

# add the second instance to the first instance container
o.obj = o2

# print an XML representation of the container instance
xml_string = xml_pickle.XML_Pickler(o).dumps ()
print xml_string

```

除了第一行和倒数第二行以外的所有代码对于使用对象实例来说都是常规 Python。它可能有些人为化且简单，但基本上您对实例数据成员（包括作为容器数据的嵌套实例，这是大多数复杂结构以 Python 构造的方式）所执行的每个操作都包含在上例中。Python 程序员只需要进行一个方法调用就能将他们的对象作为 XML 编码。

当然，一旦“pickle”了解对象，往往希望以后能操作它们（或者在其他地方使用）。假设上述几行代码已经运行，恢复对象表示很简单：

```
new_object = xml_pickle.XML_Pickler().loads(xml_string)
```

很明显，在实际情况下，您希望对所创建的 XML 文档执行的操作可能比只在运行时将它放在内存中要有趣得多。例如，将 XML 文档保存到磁盘（可能使用 XML_Pickler.dump() 方法），或者在通信通道上发送它。实际上，示例是打印到纸的，这是一种非常好的持久存储形式。

运行上述样本代码将产生一些非常典型的 Python 对象的 xml_pickle 表示法特性。但下面的示例是开发的手工编码的测试方案，具有包含文档类型中允许的每个 XML 结构、标记和属性的优点。这里创造了一些特定数据，但要想象数据所属的应用程序并不困难。同样我们来看看 PyObjects.dtd XML 文档的结构：

例 1: PyObjects.dtd

```
<?xml version="1.0"?>
```



```
<!DOCTYPE PyObject SYSTEM
"PyObjects.dtd">
<PyObject class="Automobile">
    <attr
name="doors" type="numeric" value="4" />
    <attr name="make"
type="string" value="Honda" />
    <attr name="tow_hitch"
type="None" />
    <attr name="prev_owners" type="tuple">
        <item type="string" value="Jane Smith" />
        <item
type="tuple">
            <item type="string" value="John Doe" />
            <item type="string" value="Betty Doe" />
            <item type="string" value="Charles Ng" />
        </item>
    </attr>
    <attr name="repairs" type="list">
        <item type="string"
value="June 1, 1999: Fixed radiator" />
        <item
type="PyObject" class="Swindle">
            <attr name="date"
type="string" value="July 1, 1999" />
            <attr
name="swindler" type="string" value="Ed's Auto" />
            <attr
name="purport" type="string" value="Fix A/C" />
        </item>
    </attr>
```

```
<attr name="options" type="dict">

<entry>
    <key type="string" value="Cup Holders" />

<val type="numeric" value="4" />
</entry>

<entry>
    <key type="string" value="Custom Wheels" />

<val type="string" value="Chrome Spoked" />
</entry>

</attr>

<attr name="engine" type="PyObject"
class="Engine">
    <attr name="cylinders" type="numeric"
value="4" />
    <attr name="manufacturer" type="string"
value="Ford" />
</attr>
</PyObject>
```

不难看出 `PyObjects.dtd` XML 文档的结构。DTD 将消除任何不是非常明显的问题的歧义。在研究样本 XML 文档时，你可以看到满足了三个规定的 `xml_pickle` 设计目标。

XML 表示法可以通过非 `xml_pickle` 的手段控制，无论它们是与 Python/XML 模块无关的、其他编程语言中的 XML 库、XML 增强的编辑器和实用程序，还是只是简单的文本编辑器（如样本创建中使用的那个），Python 对象的 XML 表示可以使用标准 XML 确认程序，使用 `PyObjects.dtd` 来确认符合 DTD 的所有文档并且只有符合 DTD 的文档才是有效 Python 对象的表示。

13.7.2 xml pickle 设计特点

1. 内容模型

Python 和 XML 的内容模型只不过在某些特定方面有所不同。一个重要差异是 XML

文档在格式上生来就是线性的。Python 对象属性,还有 Python 字典,没有明确的顺序(尽管实现细节创建的顺序很随意,例如散列的键)。在这方面,Python 对象模型与关系模型很接近;关系表中的各行没有“天然”的顺序,主键和辅键可以为表提供任何有意义的顺序。键总能够通过比较运算符来定序,但这种顺序与键的语义并没有关系。

XML 文档总是以特定顺序列出它的标记元素。顺序对于某些应用程序没有什么意义,但 XML 文档顺序总是存在的。将 Python 和 XML 中的键顺序重要性进行区分的结果是,由 `xml_pickle` 产生的文档不保证通过“pickle”/“unpickle”循环维护元素顺序。例如,手工准备的 `PyObjects.dtd` XML 文档,例如上面的那个,可以被“unpickle”到一个 Python 对象中。如果产生的对象又被“pickle”,标记则很可能以和原始文档中不同的顺序出现。这是一个特性,不是错误,但您要理解这一现象。

2. 限制

`xml_pickle` 的当前版本(0.2)中有几个已知的限制。一个潜在的严重缺陷是,没有人尝试捕捉复合/容器对象中的循环引用。如果对象属性往回引用到容器对象(或它的某些递归版本),`xml_pickle` 将耗尽 Python 堆栈。循环引用往往表明在开始的对象设计中存在缺陷,但 `xml_pickle` 的更新版本肯定会尝试更智能地处理它们。

另一个限制是 XML 属性值的名称空间(例如 `<attr name="123">` 中的“123”)比有效 Python 变量和实例成员的名称空间更大。在 Python 名称空间以外手工创建的属性在实例的 `__dict__` 属性中存在奇怪的状态,但对于普通属性语法是不可访问的(例如“`obj.123`”是个语法错误)。这只是在使用非 `xml_pickle` 本身的方法来创建或修改 XML 文档时才有的问题。现在我还没有确定处理这个(有些模糊)问题的最好办法。

第三个限制是 `xml_pickle` 不能处理 Python 对象的所有属性。所有“常见”的数据成员(字符串、数、字典等)都能很好地“pickle”。但不处理作为属性的实例方法、类和函数对象。使用 pickle 时,在“pickle”过程中忽略方法。如果类或函数对象作为属性存在,则出现 `XMLPicklingError`。这可能是正确的最终行为,但还不能下最后的结论。

3. 设计选择

XML 文档设计中一个真正的歧义在于要选择何时使用标记属性以及何时使用子元素。关于这一设计问题有不同的意见,XML 程序员常常会强烈地感觉到他们的观点是冲突的。这可能是在决定 `xml_pickle` 文档结构时最大的问题。

已确定的一个常规原则是,如果事物生来就是“复数”的,应该由子元素表示。例如,Python 列表包含的项数没有限制,因此由一连串子元素表示。另一方面,数是单数的事物(值

可能大于 1,但在其中只有一样事物)。在这种情况下,使用称为“value”的 XML 属性似乎更符合逻辑。真正困难的情况是在使用 Python 字符串时。它们基本上是序列对象,就像列表一样。但使用假设的标记表示字符串中每一字符将破坏人类可读性的目标,并产生大量 XML 表示。决定是将字符串放在 XML “value”属性中,如同对数进行的操作一样。不过,从美学的角度看,它们可能无法存在于一个标记容器中,特别在多行字符串的情况下。但由于在规范中没有其他“裸露”的 #PCDATA,所以这个结论似乎更合适。

部分原因是由于字符串是存储在 XML “value”属性中,但主要由于维护 XML 文档的语法性质,Python 字符串需要以“安全”的形式存储。在 Python 中可能出现一些不安全的事物。第一种类型是基本标记字符,例如大于和小于号。第二种类型是确定属性的引号和单引号。第三种类型是有疑问的 ASCII 值,例如 null 字符。考虑的一个可能性是将整个 Python 字符串以类似 base64 编码的格式进行编码。这可以让字符串“安全”,但对于人类是完全不可读的。因此决定使用混合方式。基本 XML 字符以“&”、“>”或“<”样式来转义。有疑问的 ASCII 值以 Python 样式转义,例如“\000”。这种组合产生了人类可读的 XML 表示,但需要某些混合的方法来为已存储的字符串译码。

因为在 XML 和 Python 之间存在不对称性,所以至少在最初时包含两个单独的模块:xml_pickle 和 xml_objectify,前者用于以 XML 表示任意的 Python 对象,后者用于将 XML 文档本机表示为 Python 对象。下面主要来认识 xml_objectify。

13.7.3 xml objectify

在 Python 中,例如 xmllib、xml.sax、pyxie 和 xml.dom 这样的模块和软件包提供了处理 XML 社区中一些公共 XML 文档的方法。您可能熟悉应用于其他编程语言的类似模块和库。实际上,许多模块都基于语言中性的 XML 标准,它们通常实现以 XML 为中心的处理文档和对象的方法。

常规 XML 协议的 Python 实现提供了以不同方法进行编程的灵活性。例如,可以使用如 DOM 这样的可移植标准,这样,使用一种语言的程序员可以方便地对以另一种语言编写的面向 DOM 的代码进行操作。不过,Python 程序员有时可能宁愿以更类似于“正常”Python 的方法进行编码。在许多情况下,XML 概念性框架看起来似乎更接近于 Python,而不是 Python 的一个组成部分。因此,我开发了一系列用于 XML 文档的“Python 化”模块。

1. 如何使用 xml objectify

使用 xml_objectify 很简单,而且在模块 docstring 注释中有详细记载。让我们快速浏览

一下一些样本代码:

例 2: 从 XML 文档创建 Python 对象

```
from xml_objectify import XML_Objectify
xml_obj = XML_Objectify ('address.xml')
py_obj = xml_obj.make_instance ()
```

如你所见, 从常规 XML 文档创建本机 Python 对象有两个步骤。首先创建一个类似于 DOM 的中间工厂对象 (即用于创建其他对象的对象)。然后, 从 XML_Objectify 实例中生成一个或多个 Python 对象实例。请注意, 应该使用 xml_pickler 来处理特殊的 PyObjects.dtd 格式文档。也可以在同一行上执行这两步。

例如: 在一行中创建 XML/Python 对象:

```
py_obj = XML_Objectify ('address.xml') .make_instance ()
```

当然, 在后一种情况中, 不保留工厂对象来产生更多本机对象, 而且也将清除包含它的完整 DOM 实例的 ._dom 数据成员。

为进行比较, 下例显示了使用 Python 创建 DOM 对象有多简单:

例 3: 从 XML 文档创建 DOM 对象

```
from xml.dom.utils import FileReader
dom_obj =
FileReader ().readXml (open ('address.xml'))
```

FileReader ().readXml () 需要实际的文件对象, 而 XML_Objectify () 可以接受文件对象或者普通文件名。在这两种情况下, 创建对象是一个两行的操作。

使用 xml_objectify 模块和 xml.dom 软件包的不同之处在于最后得到的对象类型。Python DOM 对象是真正的 Python 对象, 但它的属性和方法与原始 XML 文档的数据和结构对应程度并不像 XML_Objectify 对象那样接近。Python DOM 对象的属性通常嵌套了 .children 列表, 这些列表在语义上并没有什么太大的帮助。要访问样本文档中同一个 XML 属性, 可以使用 xml_objectify 的第一行, 也可以使用 DOM 的以下四行。

例 4: 使用 [xml.dom] 和 [xml_objectify] Python 对象

```
print py_obj.person[1].address.city
print dom_obj.get_childNodes () [1].get_childNodes () [3].\
```

```
get_childNodes () [3].get_attributes () ['city'].value
print dom_obj._node.children[1].children[3].children[3].\
    attributes['city'].children[0].value
```

DOM 树是按严格定序的节点树组织起来的。枚举这些节点并不困难，但要引用其中特定的节点就非常麻烦了。如果有些节点是空白文本和处理指令节点（你几乎不太关心它们），情况就更糟糕了，因此在节点列表中查找子标记多半要反复试验。在上例中，访问本机属性（例如 `.children`）和 DOM 样式的方法（例如 `.get_childNodes()`）用在不同的 `print` 语句中。使用这两种方法时，要知道引用了 XML 文档中哪些数据都很不容易。

相反，上例中第一个 `print` 语句非常好地文档化了自己。唯一需要注意的一个小问题是必须使用 Python 的基于 0 的列表索引。除此以外，这行只说：“Print the city of the address of the second person in the addressbook.”（“New York” 是每个语句都要打印的。）为进一步帮助你理解，`py_obj._class_` 就是“addressbook”，与 XML 文档的根元素对应。每个不仅包含简单文本的属性都是特定类的实例；这个特定类是根据定义它的 XML 标记命名的。

如你所见，`xml.dom` 使用起来通常比较难，它的语法也很模糊。本机 Python 对象使用起来就简单得多。请注意，`xml_objectify` 在内部广泛利用了 DOM。实际上，每个 `XML_Objectify` 实例都包含了一个 `._dom` 属性，该属性是打开的 XML 文档的 DOM 树。不过，创建的实例 `.make_instance` 不包含任何 DOM，它是根标记的类类型。

13.7.4 `xml_objectify` 的设计特点

1. 代码自测

使用 `xml_objectify`，可以利用所有现有的常规函数。`pyobj_printer()` 是个样本常规函数，其中包括 `xml_objectify` 模块。该函数产生所有 Python 对象可读的递归表示。通过将 XML 文档表示为本机 Python 文档，可以重用现有的、以抽象方式处理 Python 对象的函数。当然，DOM 对象勉强算得上是 Python 对象，但要对这些对象以有用的方式使用常规函数就比较困难。例如，因为 DOM 对象的属性嵌套了 `.children` 列表，所以使用像 `pyobj_printer()` 这样的常规函数将不会产生非常有用的输出。

2. 使用类行为的技巧

`xml_objectify` 提供了一种非常精妙的技巧，使用这种技巧，只有在没有定义类的情况下才动态地为属性值定义类。这可以让您定义具有复杂行为的类，和可以放入特定 XML 文档内容的属性。假设类 `person` 使用各种方法（如果需要，包括 `._init_()` 方法）进行了预定

义。导入到上例中 XML 地址簿中的每个 “person” 都将具有给予它的所有行为，包括对放入实例中的数据的操作方法。当然，如果在对文档运行 XML_Objectify () 之前没有进行预定义，类就只是用于在实际 XML 中定义的属性的容器。

3. 字符标记处理

XML 标记通常是块级别的，但某些也属于字符级别。依我看，自然的 Python 表示在每种情况下各不相同。块级别的子标记易于通过父标记的属性表示，父标记是根据子标记命名的。子标记属性的值是新 Python 对象，它也是根据子标记命名的类型。例如，person 从层次上角度上考虑，可以具有 address 和 misc-info。使用 Python，可以用 person.address 和 person.misc_info 来引用它们。

使用字符级别的标记时，标记的内容是文本数据和这些数据的标记（常常是排版方面的）的混合体，子标记在层次结构上不是父标记中的一部分。例如，misc_info 对象实际上没有 ital 属性。那么，下面的 XML 类型应该如何表示呢？

```
<misc-info>One of the <ital>most</ital> talented  
actresses on TV.</misc-info>
```

xml_objectify 将一个称为 ._XML 的特殊属性添加到看起来包含标记字符数据的对象 / 标记。这个属性在标记中包含文字 XML。例如，如果给定的嵌套对象有 ._XML 属性，pyobj_printer () 函数显示这个文字 XML 而不是递归属性。不过，仍然执行标准递归子标记和对象的创建，因此可以知道哪些属性和结构最适当。

4. 本机 Python 对象只包含根文档

许多 XML 文档都伴随着标记和字符数据内容提供处理指令和 / 或注释。不过，由 XML_Objectify 对象的 .make_instance () 方法创建的本机 Python 对象只包含文档根标记的内容。而且将忽略 XML 注释；只表示标记属性和字符数据。

在上面的从 XML 文档创建 Python 对象示例中，如果保留原始的 XML_Objectify 对象 (xml_obj)，可以访问它的 .processing_instruction 属性，甚至可以访问它的 ._dom 属性来查看本机 Python 对象忽略了什么。

5. 属性类型简化

所有 XML 属性都转换成字符串类型的 Python 对象属性。当前，Python 不表示属性的 XML 枚举或数字类型。在以后的版本中可能加入这样的能力，但这些通常需要 DTD，而 xml_objectify 不具有 DTD。

6. 子标记属性

XML 子标记或者由对象类型的 Python 属性表示，或者由这种对象的列表表示，这要取决于是一个还是几个相同类型的子标记。即由包含相同类型多个子标记的特定标记决定。例如，在上面第一个 `address.xml` 示例中，一个人的联系信息可能包括 1 个家庭电话，而另一个人的联系信息可能包括 0 个或几个。相应地，一些 `contact_info` 对象没有 `.home_phone` 属性，一些具有包含一个 `home_phone` 对象的 `.home_phone` 属性，还有一些具有包含许多 `home_phone` 对象的 `.home_phone` 属性。尽管使用 DTD 的情况下有可能施加更多法则，但我认为，Python 应用程序需要这种动态能力。

7. Python 名称空间限制

要知道 Python 名称空间比 XML 名称空间小。因此，有时要修改标记或属性的 XML 名称。`xml_objectify` 将破折号、冒号和镑值 / 散列标志转换成下划线。该模块不处理任何其他名称空间冲突。例如，如果 XML 文档有标记 `<spam-eggs>`、`<spam_eggs>`、`<spam:eggs>` 和 `<spam#eggs>`，那么 `xml_objectify` 所创建的 Python 对象就不能正确表示 XML 文档。在大多数情况下，这不是问题，因为人们不希望得到的 XML 文档具有这些冲突的标记。

13.7.5 `xml_objectify` 的前景

目前，还不能将本机 Python 对象转回带有与读取的 XML 文档相同结构的 XML 文档。因为 `xml_objectify` 故意舍去 XML 文档中有关顺序的信息来产生更易理解的 Python 对象，所以可能出现问题。Python 属性没有任何预先确定的顺序，但需要 XML 标记和属性以特定的顺序排列。即使不需要 XML 标记来以特定的顺序出现，顺序在语义上还是很重要。（请注意在重复公共子标记的情况下，Python 列表维护顺序。）为了转换回 XML，我们需要或者选择任意顺序，或者保留本机 Python 对象中的顺序信息，使得它看上去不太像 Python。

第 14 章 Python 中的 Curses 编程

相信你在网络上一定用过如 tin, elm 等工具, 这些软件有项共同的特色, 即他们能利用上下左右等方向键来控制光标的位置。除此之外, 这些程序的画面也较为美观。对 Programming 有兴趣的朋友一定对此感到好奇, 也许他能在 PC 上用 Turbo C 轻易地写出类似的程序, 然而, 但当他将相同的程序一字不变地移到工作站上来编译时, 却出现一堆抓也抓不完的错误。其实, 原因很简单, 他使用的函数库可能在 UNIX 上是没有定义的。有些在 TurboC 上被广泛使用的一些函数, 可能在 UNIX 上是不被定义的。为了因应网络上各式各样的终端类型 (terminal), UNIX 上特别发展出一套函数库, 专门用来处理 UNIX 上光标移动及屏幕的显示。

某一类 Python 应用程序最好使用交互式用户界面, 这样可以消除图形环境的系统开销或复杂性。交互式文本模式程序 (在 Linux/UNIX 中), 例如封装在 Python 的标准 curses 模块中的 ncurses 库, 正是你所需要的。在这章中, 我们首先来认识 curses.h 函数库利用这个函数库, 你也可以写出像 elm 般利用方向键来移动光棒位置的程序。然后, 来看在 Python 中 curses 的用法。

14.1 Curses 的历史与版本

curses 最早是由柏克莱大学的 Bill Joy 及 Ken Arnold 所开发出来的。当时开发此一函数库主要原因是为了提高程序对不同终端的兼容性而设计的。因此, 利用 curses 开发出来的程序将和你所使用的终端无关。也就是说, 你不必担心你的程序因为换了一部终端而无法使用。这对程序设计者而言, 尤其是网络上程序的撰写, 是件相当重要的一件事。curses 之所以能对上百种以上的终端工作, 是因为它将所有终端的数据, 存放在一个叫 termcap 的数据库中, (而在第二版的 System V 系统中, 新版的 curses 以 terminfo 取代原来的 termcap)。有了这些记录, 程式就能够知道遇到哪一种终端时, 须送什么字符才能移动光标的位置, 送什么字符才能清除整个屏幕清除。

Python 的标准 curses 提供了“玻璃电传” (glass teletype) (在 20 世纪 70 年代, 原始 curses 库刚创建时, 它叫做 CRT) 的公共特性的基本接口。有许多方法可以让用 Python

编写的交互式文本模式程序变得更巧妙。这些方法分成两类。

一方面，有些 Python 模块支持 `ncurses` (`curses` 的超集) 或 `slang` (相似却独立的控制台库) 的全部功能集合。最值得注意的是，这当中有一个增强库 (由适当的 Python 模块封装) 可以让你将颜色添加到界面上。

另一方面，许多构建在 `curses` (或 `ncurses` /`slang`) 上的高级窗口小部件库添加了诸如按钮、菜单、滚动栏和各种公共界面设备之类的特性。如果你看到过诸如 Borland's TurboWindows (DOS 版) 之类的库开发的应用程序，你就知道在文本模式控制台中，这些特性是多么吸引人。窗口小部件库中的功能单单使用 `curses` 都可以达，但是还可以利用其他程序员在高级界面上取得的成果。

本章只涉及 `curses` 自身的特性。由于 `curses` 模块是标准发行版的一部分，你不必下载支持库或其他 Python 模块就可以找到并使用它 (至少在 Linux 或 UNIX 系统中是这样)。理解 `curses` 提供的基本支持很有用，即使只是作为理解高级模块的基础。并且不使用其他模块，单独使用 `curses` 构建漂亮且实用的 Python 文本模式应用程序也很简单。预先发行的说明提到 Python 2.0 将包括 `curses` 的增强版本，但不管怎样，它应该兼容此处说明的版本。

14.2 认识Curses编程的思路

在介绍用 Python 封装 `curses` 应用程序设计之前我们来认识 Curses 底层的实现原理和过程是很有意义的，下面我们来介绍在一般的程序语言中使用 Curses。

1. C 语言中实现 Curses

在你的 C 程序的头文件中将 `<curses.h>` include 进来。当你引进 `curses.h` 这个函数库后，系统会自动将 `<stdio.h>` 和 `<unistd.h>` 一并 include 进来。另外，在 System V 版本中，`<terminfo.h>` 这个函数库也将一并 include 进来。

```
#include <curses.h>
main ()
{
    :
    :
}
```

当然，你的系统内必须放有 `curses.h` 这个函数库。

2. 如何编译 (compile)

当你编辑好你的程序，在 UNIX 提示符号下键入：

```
% /usr/5bin/cc [file.c] -lcurses
```

```
^^^^^^^
```

引进 curses.h 这个 library

```
或 % /usr/5bin/cc [file.c] -lcurses -ltermib
```

3. 如何编写 curses 程序

在开始使用 curses 的一切命令之前，你必须先利用 `initscr()` 这个函数来开启 curses 模式。相对的，在结束 curses 模式前（通常在你结束程序前）也必须以 `endwin()` 来关闭 curses 模式。

```
#include <curses.h>

main ()
{
    initscr ();
    :
    :
    :
    endwin ();
}
```

这是一般 curses 程序标准的模式。此外，你可以就你程序所需，而做不同的设定。当然，你可以不做设定，而只是调用 `initscr()`。你可以自己写一个函数来存放所有你所需要的设定。平常使用时，只要调用这个函数即可启动 curses 并完成一切设定。

下面的例子，即是笔者将平常较常用的一些设定放在一个叫 `initial()` 的函数内。

```
void initial ()
{
    initscr ();
    cbreak ();
    nonl ();
    noecho ();
    intrflush (stdscr, FALSE);
    keypad (stdscr, TRUE);
```

```
    refresh ();  
}
```

各函数分别介绍如下：

initscr()

initscr() 是一般 **curses** 程序必须先调用的函数，一旦这个函数被调用之后，系统将根据终端的形态并启动 **curses** 模式。

endwin()

curses 通常以调用 **endwin()** 来结束程序。**endwin()** 可用来关闭 **curses** 模式，或是暂时的跳离 **curses** 模式。如果你在程序中须要 call shell (如调用 **system()** 函数) 或是需要做 **system call**, 就必须先以 **endwin()** 暂时跳离 **curses** 模式最后再以 **wrefresh()** **doupdate()** 来重返 **curses** 模式。

cbreak() 和 **nocbreak()**

当 **cbreak** 模式被开启后，除了 **DELETE** 或 **CTRL** 等仍被视为特殊控制字符外一切输入的字符将立刻被一一读取。当处于 **nocbreak** 模式时，从键盘输入的字符将被储存在 **buffer** 里直到输入 **RETURN** 或 **NEWLINE**。在较旧版的 **curses** 须调用 **crmode()**，**nocrmode()** 来取代 **cbreak()**，**nocbreak()**。

nl() 和 **nonl()**

用来决定当输入数据时，按下 **RETURN** 键是否被对应为 **NEWLINE** 字符 (如 **\n**)。而输出数据时，**NEWLINE** 字符是否被对应为 **RETURN** 和 **LINDFEED** 系统预设是开启的。

echo() 和 **noecho()**

此函数用来控制从键盘输入字符时是否将字符显示在终端上。系统预设是开启的。

intrflush(win, bf)

调用 **intrflush** 时须传入两个值：

win 为一 **WINDOW** 类型指针，通常传入标准输出屏幕 **stdscr** **bf** 为 **TRUE** 或 **FALSE**，当 **bf** 为 **true** 时，当输入中断字符 (如 **break**) 时，中断的反应将较为快速。但可能会造成屏幕的错乱。

keypad(win, bf)

调用 keypad 时须传入两个值:

win 为一 WINDOW 类型指针, 通常传入标准输出入屏幕 stdscr, bf 为 TRUE 或 FALSE, 当开启 keypad 后, 可以使用键盘上的一些特殊字符, 如上下左右等方向键, curses 会将这些特殊字符转换成 curses.h 内定义的一些特殊键。这些定义的特殊键通常以 KEY_ 开头。

refresh()

refresh() 为 curses 最常用的调用一个函数。

curses 为了使屏幕输出达最佳化, 当你调用屏幕输出函数企图改变屏幕上的画面时, curses 并不会立刻对屏幕做改变, 而是等到 refresh() 调用后, 才将刚才所做的变动一次完成。其余的数据将维持不变。以尽可能送最少的字符至屏幕上。减少屏幕重绘的时间。如果是 initscr() 第一次调用 refresh(), curses 将做清除屏幕的工作。

4. 光标的控制

move(y, x) 将光标移动至 x, y 的位置

getyx(win, y, x) 得到目前光标的位置 (请注意! 是 y, x 而不是 &y, &x)

5. 有关清除屏幕的函数

clear()

erase() 将整个屏幕清除 (请注意配合 refresh() 使用)

6. 如何在屏幕上显示字符

echochar(ch) 显示某个字符

addch(ch) 显示某个字符

mvaddch(y, x, ch) 在 (x, y) 上显示某个字符, 相当于调用 move(y, x); addch(ch);

addstr(str) 显示一串字符串

mvaddstr(y, x, str) 在 (x, y) 上显示一串字符串, 相当于调用 move(y, x); addstr(str);

printw(format, str) 类似 printf(), 以一定的格式输出至屏幕

mvprintw(y, x, format, str) 在 (x, y) 位置上做 printw 的工作。相当于调用 move(y, x); printw(format, str);

7. 如何从键盘上读取字符

getch() 从键盘读取一个字符 (注意! 传回的是整数值)

`getstr()`

从键盘读取一串字符

`scanw (format, &arg1, &arg2...)` 如同 `scanf`, 从键盘读取一串字符

例:

```

int ch;
char string1[80];    /* 请注意! 不可声明为 char *string1; */
char string2[80];
echo ();             /* 开启 echo 模式, 使输入立刻显示在屏幕上 */
ch=getch ();
string1=getstr ();
scanw ("%s", string2);
mvprintw (10, 10, "String1=%s", string1);
mvprintw (11, 10, "String2=%s", string2);

```

8. 如何利用方向键

`curses` 将一些如方向键等特殊控制字符, 以 `KEY_` 为开头定义在 `curses.h` 这个文件里, 如 `KEY_UP` 即代表方向键的 “↑”。但如果你想使用 `curses.h` 为你定义的这些特殊键的话, 你就必须将 `keypad` 设定为 `TRUE`。否则, 你就必须自己为所有的特殊键定义了。

`curses.h` 为一些特殊键的定义如下:

<code>KEY_UP</code>	0403	↑
<code>KEY_DOWN</code>	0402	↓
<code>KEY_LEFT</code>	0404	←
<code>KEY_RIGHT</code>	0405	→
<code>KEY_HOME</code>	0406	Home key (upward+left arrow)
<code>KEY_BACKSPACE</code>	0407	backspace (unreliable)
<code>KEY_F0</code>	0410	Function keys.
<code>KEY_F (n)</code>	<code>(KEY_F0+ (n))</code>	Formula for f.
<code>KEY_NPAGE</code>	0522	Next page
<code>KEY_PPAGE</code>	0523	Previous page

其他常用的一些特殊字符

```

[TAB]           /t
[ENTER]         /r

```

[ESC]	27
[BACKSPACE]	127

9. 如何改变屏幕显示字符的属性

为了使输出的屏幕画面更为生动美丽, 我们常须要在屏幕上做一些如反白, 闪烁等变化。`curses` 定义了一些特殊的属性, 通过这些定义, 我们也可以在 `curses` 程序中控制屏幕的输出变化。

<code>attron (mod)</code>	开启属性
<code>attroff (mod)</code>	关闭属性

`curses.h` 里头定义了一些属性, 如:

<code>A_UNDERLINE</code>	加底线
<code>A_REVERSE</code>	反白
<code>A_BLINK</code>	闪烁
<code>A_BOLD</code>	高亮度
<code>A_NORMAL</code>	标准模式 (只能配合 <code>attrset()</code> 使用)

当使用 `attron()` 开启某一种特殊属性模式后, 接下来在屏幕的输出都会以该种属性出现。到你调用 `attroff()` 将此模式关闭。请注意, 当你欲 `attron()` 开启另一种属性时, 请记住利用 `attroff()` 先关闭原来的属性, 或直接以 `attrset(A_NORMAL)` 将所有特殊属性关闭。否则, `curses` 会将两种属性做重叠处理。例:

```
attrset (A_NORMAL);           /* 先将属性设定为正常模式      */
attron (A_UNDERLINE);         /* 加下划线                      */
mvaddstr (9, 10, "加底线");    /* 加底线输出一串字符          */
attroff (A_UNDERLINE);        /* 关闭加底线模式, 恢复正常模式 */
attron (A_REVERSE);           /* 开启反白模式                  */
mvaddstr (10, 10, "反白");     /* 输出一串反白字符            */
attroff (A_REVERSE);          /* 关闭反白模式, 恢复正常模式   */
attron (A_BLINK);             /* 开启闪烁模式                  */
mvaddstr (11, 10, "闪烁");     /* 输出一串闪烁字符            */
attroff (A_BLINK);            /* 关闭闪烁模式, 恢复正常模式   */
attron (A_BOLD);              /* 开启高亮度模式                */
mvaddstr (12, 10, "高亮度");   /* 输出一串高亮度字符          */
attroff (A_BOLD);             /* 关闭高亮度模式, 恢复正常模式 */
```


10.应用示例

下面所举的例子，即完全利用刚刚所介绍的函数来完成。这个程序可将从键盘上读取的字符显示在屏幕上，并且可以用上下左右方向键来控制光标的位置，当按下 [ESC] 后，程序即结束。

```
#include <curses.h>                /* 引进curses.h, 并自动引进 stdio.h */
#define StartX 1                    /* 决定光标初始位置 */
#define StartY 1

void initial ();

main ()
{
    int x=StartX;                    /* 声明 x, y 并设定其初值 */
    int y=StartY;
    int ch;                          /* 声明 ch 为整数, 配合 getch () 使用 */
    initial ();                      /* 调用initial (), 启动curses 模式, */
                                    /* 并完成其他设定 */
    box (stdscr, 'l', '-');          /* 画方框 */
    attron (A_REVERSE);              /* 开启反白模式 */
    mvaddstr (0, 20, "Curses Program"); /* 在 (20, 0) 处输出反白字符 */
    attroff (A_REVERSE);             /* 关闭反白模式 */
    move (x, y);                     /* 将光标移至初始位置 */
    do {                             /* 以无限循环不断等待输入 */
        ch=getch ();                 /* 等待自键盘输入字符 */
        switch (ch) {                /* 判断输入字符为何 */
            case KEY_UP: --y;         /* 判断是否"↑"键被按下 */
                break;
            case KEY_DOWN: ++y;       /* 判断是否"↓"键被按下 */
                break;
            case KEY_RIGHT: ++x;      /* 判断是否"→"键被按下 */
                break;
            case KEY_LEFT: --x;       /* 判断是否"←"键被按下 */
                break;
            case '\r':                 /* 判断是否 ENTER 键被按下 */
```

```
        ++Y;
        x=0;
        break;
    case '\t':
        /* 判断是否 TAB 键被按下 */
        x+=7;
        break;
    case 127:
        /* 判断是否 BACKSPACE 键被按下 */
        mvaddch (y, --x, ' '); /* delete 一个字符 */
        break;
    case 27: endwin ();
        /* 判断是否 [ESC] 键被按下 */
        exit (1);
        /* 结束 curses 模式 */
        /* 结束此程序 */
    default:
        addch (ch);
        /* 如果不是特殊字符，将此字符印出 */
        x++;
        break;
    }
    move (y, x);
    /* 移动光标至现在位置 */
} while (1);
}

void initial ()
/* 自定开启 curses 函数 */
{
    initscr ();
    cbreak ();
    nonl ();
    noecho ();
    intrflush (stdscr, FALSE);
    keypad (stdscr, TRUE);
    refresh ();
}
```

14.3 Curses多视窗处理方式

上节我们结合 C 语言介绍了 Curses 编程的一些基本思路, 以及一些常用的函数。我们利用上节介绍的一些基本函数就可以来编辑一些比较实用简单的图形应用程序。在结合 Python 语言之前我们还是要介绍一下 curses 多视窗处理方式, 使读者使用 Python 之前对 Curses 有一个更深的认识。

1. 视窗的建立

视窗的建立, 以 `newwin()` 这个函数来完成。同时, 需声明此视窗为 WINDOW 结构变量。

```
WINDOW*newwin (lines, columns, start_y, start_x);
```

```
WINDOW*win;
```

```
win=newwin (10, 20, 0, 0);
```

如此, 将以(0, 0)为原点, 取一个 10 列 20 行的矩形为一个新的视窗。今后我们只要调用 win 这个变量, 就可以对这新视窗做处理。

如: `wmove (win, 3, 2);`

2. 多视窗处理函数的格式

这一类函数和一般的基本函数极为类似, 几乎每一个基本函数都有一个对应的视窗处理函数。一般将 'w' 加在函数的里作为区别, 'w' 乃 'window' 之意。另外, 因为可同时处理多个视窗, 在调用使用时, 需特别指定欲处理的视窗。当然, 如果你指定对 `stdscr` 做处理, 由于是对标准输出入屏幕处理, 其作用将相当于一般的。例:

`wmove (win, y, x)` 即对 win 这个视窗做 `move()` 动作。

`wmove (stdscr, y, x)` 相当于 `move (y, x)`

下面我们介绍一些较重要的函数。

```
wmove (win, y, x)
```

```
touchwin (win)
```

```
wrefresh (win)
```

```
mvwaddstr (win, y, x, str)
```

```
wattron (attr)
```

```
delwin (win)
```

```
subwin (win, ny, nx, y, x)
```

其他函数多和基本函数互为对应, 故不全部列出, 详细名称可参考 curses 的 online

manual。

3. 视窗内的坐标系

视窗内的坐标系，将以此视窗的起始点为新原点，并以其相对位置作为新的座标。举例来说：

```
win=newwin (10, 20, 5, 5) ;
```

```
wmove (win, 2, 3)
```

将以 (5, 5) 为新原点，y 方向移动 2 单位，x 方向移动 3 单位。因此实际上，光标将移动至 $y=7, x=8$ 的位置上。

4. POP-UP 视窗的建立

利用 curses 所提供的视窗处理函数，我们可以做出像 ONLINE HELP 的 POP-UP 画面。当按下某键后，一个新的视窗将像“跳”出来一般覆盖原来的画面。当关掉此视窗后，又不会影响到原来被覆盖的画面。

下面的例子，我们模拟 ONLINE HELP 的形式，当按下 ‘h’ 键时，视窗即出现

```
#include <curses.h>
main ()
{
    int ch, x, y;
    WINDOW *win;
    initscr ();      ← ↵
    cbreak;          | 启动 curses 模式
    noecho ();        |
    nonl ();          ← ↵

    win=newwin (4, 30, LINES/2-3, COLS/2-15); /* 建立一个新视窗，其中LINES,
COLS
*/

    box (win, '|', '-'); /* 为 curses 内定值，
即屏幕行/列数*/

    mvwaddstr (win, 1, 4, "This is another screen");
    mvwaddstr (win, 2, 2, "Press anykey to continue..");
    for (y=0; y<LINES; ++y) /* 以'@'填满屏幕 */
        for (x=0; x<COLS; ++x)
```

```
        mvprintw (y, x, "@");

for (;;) {
    refresh ();
    ch=getch ();
    switch (ch) {
        case 'q':                /* 按 'q' 键离开 */
                                endwin ();
                                exit (0);

        case '\t':                /* 按 [TAB] 键 调用另一视窗 */
                                touchwin (win);          /* wrefresh () 前需 touchwin () */
                                wrefresh (win);
                                getch ();                /* 按任意键关闭视窗 */
                                touchwin (stdscr);
                                break;

        default:break;
    }
}
}
```

5. 视窗的滚动

视窗的滚动，用来配合视窗的处理，当我们持续对视窗输出直到视窗的光标移动至最后一列时，如果我们再输出一列或是输出一个换行字符时，视窗可整个往上滚动一行。这对我们撰写一个编辑程式时，是尤其重要的，一个画面无法滚动的编辑器，势必无法处理超过一个屏幕大小的文件。视窗的滚动是预设为关闭的，并以 `scrollok ()` 来控制开闭。

`scrollok (win, TRUE)`；开启

`scrollok (win, FALSE)`；关闭

下面的例子因为不断地输出 0, 1, 2……故将以一个 40 * 10 的视窗不停的滚动。

```
#include <curses.h>
main ()
{
```

```

int i;
WINDOW *scrwin, *boxwin;
initscr ();           ← ↵
cbreak;               | 启动 curses 模式
noecho ();            |
nonl ();              ← ↵

scrwin=newwin (10, 40, LINES/2-6, COLS/2-25); /* 设定内 视窗大小 */
boxwin=newwin (12, 42, LINES/2-7, COLS/2-26); /* 设定外框视窗大小 */
scrollok (scrwin, TRUE); /* 开启视窗滚动功能 */

box (boxwin, '|', '-');
refresh ();
wrefresh (boxwin);

for (i=0;; ++i)        /* 不断地在视窗内输出 0-8 的数字，使视窗滚动
*/
{
    wprintw (scrwin, "%d", i%9);
    wrefresh (scrwin);
}
}

```

6. 模拟 joe 分割画面同时编辑两个文件

在下面的例子里，我们应用了多视窗处理的函数，改良上回介绍的编辑器，在这个程式里，我们可以同时编辑两个画面，并以[ESC]做不同视窗间的切换。同时，按下[TAB]键，会出现 POP-UP 的 ONLINE HELP。

```

#include <curses.h>
void initial ();
main ()
{
    WINDOW *win[2], *curwin, *helpwin;
    int nowwin;
    int x, y;
    int i;

```

```
int ch;

initial ();

win[0]=newwin (LINES/2-1, COLS-1, 0, 0);          /* 设定两个视窗的大小 */
win[1]=newwin (LINES/2-1, COLS-1, LINES/2, 0);

helpwin=newwin (3, 30, 2, COLS/2-15 );          /* ONLINE HELP 的大小 */
box (helpwin, '|', '-');
mvwaddstr (helpwin, 0, 10, "ONLINE HELP");        /* ONLINE HELP 的内容 */
mvwaddstr (helpwin, 1, 4, "Hit any key to continue..");

for (i=0; i<COLS-1; ++i)                        /* 画两个视窗间的界限 */
    mvaddch (LINES/2-1, i, '~');

nowwin=0;                                         /* 先指定光标在第一视窗 */
curwin=win[nowwin];
getyx (curwin, y, x);
move (0, 0);
refresh ();
refresh ();

do {
    ch=getch ();
    switch (ch) {

        case KEY_UP: --y;                        /* 判断是否"↑"键被按下 */
            break;
        case KEY_DOWN: ++y;                      /* 判断是否"↓"键被按下 */
            break;
        case KEY_RIGHT: ++x;                    /* 判断是否"→"键被按下 */
            break;
        case KEY_LEFT: --x;                     /* 判断是否"←"键被按下 */
            break;
        case '\r':                               /* 判断是否 ENTER 键被按下 */
            ++y;
            x=0;
    }
```

```

        break;
    case '\t':
        /* 判断是否 TAB 键被按下 */
        touchwin (helpwin);
        wrefresh (helpwin); /* 调用 ONLINE HELP */
        getch ();
        touchwin (win[1-nowwin]); /* 重画第一、二视窗 */
        wrefresh (win[1-nowwin]);
        touchwin (curwin);
        wrefresh (curwin);
        break;

    case 127:
        /* 判断是否 BACKSPACE 键被按下 */
        wmove (curwin, y, --x); /* delete 一个字符 */
        waddch (curwin, ' ');
        break;

    case 27 : nowwin=1-nowwin; /* [ESC] 键切换视窗 */
        curwin=win[nowwin];
        getyx (curwin, y, x);
        break;

    default:
        waddch (curwin, ch);
        x++;
        break;
    }

    wmove (curwin, y, x);
    wrefresh (curwin);
} while (1);
}

void initial ()
{
    initscr ();
    cbreak ();
    nonl ();
    noecho ();

```

```

└─┘
| 启动 curses 模式
|
└─┘

```



```
    intrflush (stdscr, FALSE);  
    keypad (stdscr, TRUE);  
    refresh ();  
}
```

14.4 Python: Curses编程

这一节我们将用一个用 Python 编写的封装器，来详细深入的介绍 Curses 编程。该应用程序是一个文本到 HTML 转换程序 Txt2Html，Txt2Html 有几种运行方式。但为了与本文的目的保持一致，我们将研究从命令行运行的 Txt2Html。操作 Txt2Html 的一种方式是向它提供一组命令行变量（它们说明要执行的转换的各方面），然后将应用程序当作批处理运行。对于偶尔使用的用户，一个更友好的用户界面提供了一个交互式选择屏幕，它可以在执行实际转换之前，引导用户遍历转换选项（提供选中选项的视觉反馈）。

`curses_txt2html` 的界面基于常见的顶栏菜单，它带有下拉和嵌套子菜单。所有菜单相关的功能都在 `curses` 上“从头”开始设计。虽然这些菜单缺少更复杂的 `curses` 封装器的一些特性，但它们的基本功能是由几行只使用 `curses` 的代码实现的。这个界面还带有一个简单的滚动帮助框和几个用户输入字段。以下是显示常规布局 and 样式的应用程序的屏幕快照。

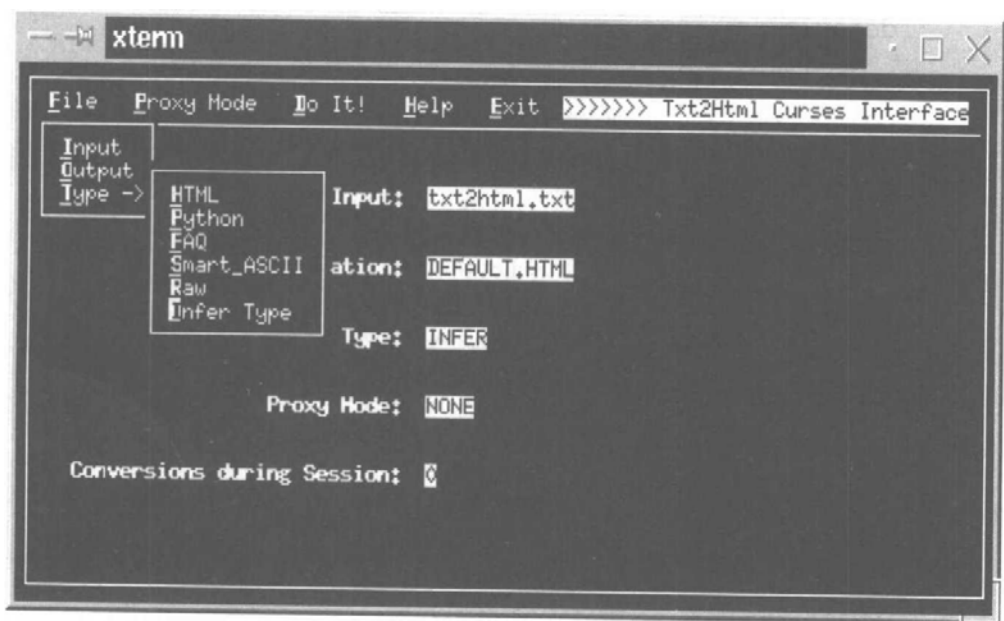


图 14.1 X 终端上的应用程序

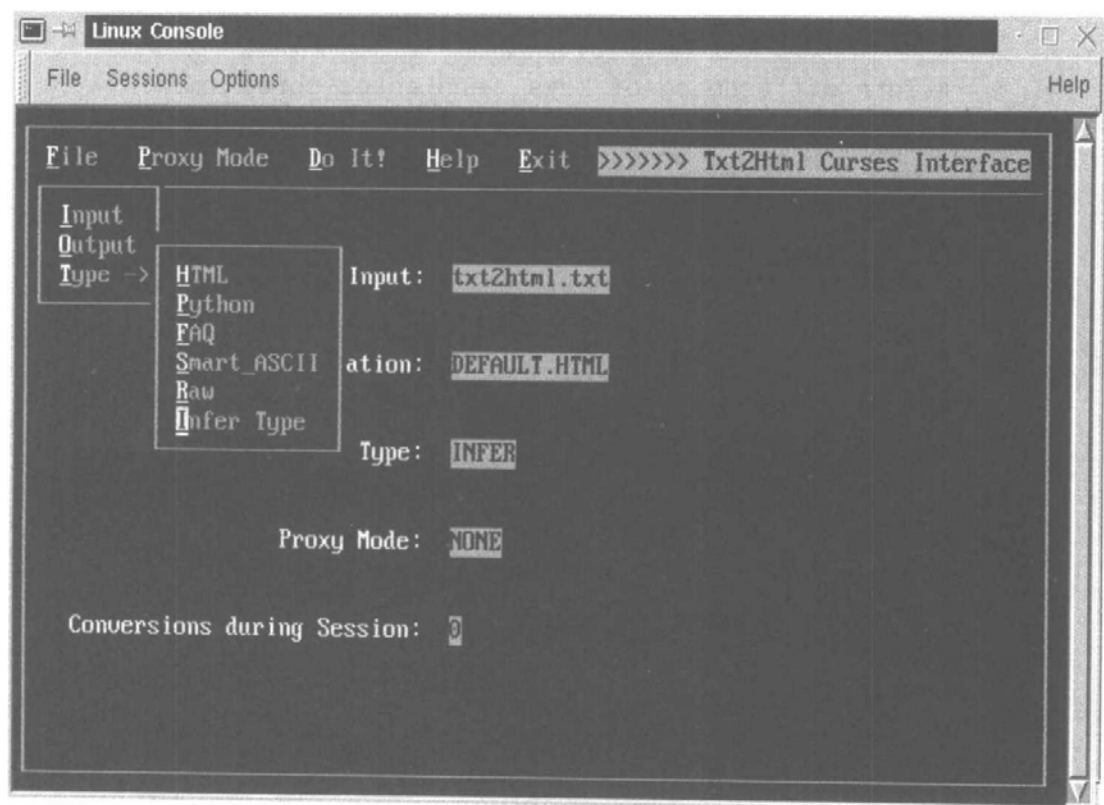


图 14.2 Linux 终端上的应用程序

1. 封装 curses 应用程序

curses 编程的基本元素是窗口对象。窗口是带有一个可寻址光标的实际物理屏幕的区域，光标的坐标与窗口相关。可以到处移动窗口，并且可以创建和删除窗口而不影响其他窗口。在窗口对象中，输入或输出操作发生在光标上，这通常由输入或输出方法明确设置，但也可以分别修改。

在初始化 curses 之后，可以用各种方式修改或完全禁用面向流的控制台输入和输出。这基本上就是使用 curses 的全部重点。可是一旦更改了流式控制台交互，如果程序出错，将不会以正常方式显示 Python 追溯事件。Andrew Kuchling 使用一个很好的 curses 程序顶级框架解决了这个问题。

以下模板（基本上与 Kuchling 的相同）保留在正常命令行 Python 的错误报告功能：

例 1: Python [curses] 程序的顶层设置代码

```
import curses, traceback
if __name__ == '__main__':
    try:
        # Initialize curses
```

```
stdscr=curses.initscr ()
# Turn off echoing of keys, and enter cbreak mode,
# where no buffering is performed on keyboard input
curses.noecho ()
curses.cbreak ()

# In keypad mode, escape sequences for special keys
# (like the cursor keys) will be interpreted and
# a special value like curses.KEY_LEFT will be returned
stdscr.keypad (1)
main (stdscr) # Enter the main loop
# Set everything back to normal
stdscr.keypad (0)
curses.echo ()
curses.nocbreak ()
curses.endwin () # Terminate curses

except:
    # In event of error, restore terminal to sane state.
    stdscr.keypad (0)
    curses.echo ()
    curses.nocbreak ()
    curses.endwin ()
    traceback.print_exc () # Print the exception
```

try 代码块执行一些初始化，调用 `main()` 函数来执行实际工作，然后执行最后的清除。如果出错，`except` 代码块会将控制台恢复成缺省状态，然后报告遇到的异常。

2. `main()` 事件循环

现在，我们研究 `main()` 函数，看看 `curses_txt2html` 做些什么：

`curses_txt2html.py` `main()` 函数和事件循环

```
def main (stdscr) :
    # Frame the interface area at fixed VT100 size
    global screen
    screen = stdscr.subwin (23, 79, 0, 0)
```

```

        screen.box ()
        screen.hline (2, 1, curses.ACS_HLINE, 77)
        screen.refresh ()

# Define the topbar menus
        file_menu = ("File", "file_func () ")
        proxy_menu = ("Proxy Mode", "proxy_func () ")
        doit_menu = ("Do It!", "doit_func () ")
        help_menu = ("Help", "help_func () ")
        exit_menu = ("Exit", "EXIT")

# Add the topbar menus to screen object
        topbar_menu ((file_menu, proxy_menu, doit_menu,
                        help_menu, exit_menu))

# Enter the topbar menu loop
while topbar_key_handler ():
        draw_dict ()

```

根据由空行隔开的三部分，很容易理解 `main()` 函数。

第一部分执行应用程序外观的常规设置。为了建立应用程序元素之间的可预期间隔，交互式区域限制在 80×25 VT100/PC 屏幕大小（即使实际的终端窗口更大）。程序围绕这个子窗口绘制一个框，并使用水平线画出顶栏菜单的视觉偏移量。

第二部分建立应用程序所使用的菜单。函数 `topbar_menu()` 使用一些技巧将热键绑定到应用程序操作并用期望的视觉属性来显示菜单。请获取源码文件以查看所有代码。`topbar_menu()` 应该是非常普通的。（欢迎将它合并到你自己的应用程序中。）非常重要的一旦绑定了热键，它们就 `eval()` 与菜单相关的字节组第二个元素中包含的字符串。例如，激活以上设置中的 "File" 菜单将调用 `eval("file_func()")`。所以就要求应用程序定义叫做 `file_func()` 的函数，要求它返回一个布尔（Boolean）值以表示是否达到应用程序终止状态。

第三部分只有两行，但这正是整个应用程序实际运行的部分。函数 `topbar_key_handler()` 就像它的名称所暗示的：它等待击键，然后处理它们。击键处理程序可以会返回 Boolean false 值。（如果是这样，则应用程序终止。）该应用程序中，键处理程序主要是检查第二段中绑定的键。但即使你的 curses 应用程序绑定键的方式与该应用程序不同，你仍要使用类似的事件循环。处理程序的关键部分很可能使用以下这行代码：

```
c = screen.getch () # read a keypress
```

对 `draw_dict()` 的调用只是事件循环中唯一的代码。此函数绘制了 `screen` 窗口中几处位置中的值。但在应用程序中，你可能想要将以下这行代码：

```
screen.refresh () # redraw the screen w/ any new output
```

加到绘制 / 刷新函数中（或只加到事件循环本身中）。

5. 获取用户输入

`curses` 应用程序以击键事件的形式获取所有用户输入。我们已经看过了 `.getch()` 方法，现在让我们看一下将 `.getch()` 与其他输入方法组合在一起的例子 `.getstr()`。以下就是我们以前提到的 `file_func()` 函数的缩写版本（它由 "File" 菜单激活）。

例 2： `curses_txt2html.py` `file_func()` 函数

```
def file_func () :
    s = curses.newwin (5, 10, 2, 1)
    s.box ()
    s.addstr (1, 2, "I", hotkey_attr)
    s.addstr (1, 3, "nput", menu_attr)
    s.addstr (2, 2, "O", hotkey_attr)
    s.addstr (2, 3, "utput", menu_attr)
    s.addstr (3, 2, "T", hotkey_attr)
    s.addstr (3, 3, "ype", menu_attr)
    s.addstr (1, 2, ".", hotkey_attr)
    s.refresh ()
    c = s.getch ()
    if cin (ord ('I'), ord ('i'), curses.KEY_ENTER, 10) :
        curses.echo ()
        s.erase ()
        screen.addstr (5, 33, " " * 43, curses.A_UNDERLINE)
        cfg_dict['source'] = screen.getstr (5, 33)
        curses.noecho ()
    else:
        curses.beep ()
        s.erase ()
```

```
return CONTINUE
```

此函数组合了几个 `curses` 特性。它做的第一件事就是创建另一个窗口对象。由于这个新窗口对象是“File”选择项的实际下拉菜单，所以程序使用 `box()` 方法围着它绘制了一个框架。在窗口 `s` 中，程序绘制了几个下拉菜单选项。使用了一种稍微费力的方法突出显示了每个选项的热键，这样就与选项描述的其余部分形成了对比。最后的 `addstr()` 调用将光标移到缺省菜单选项。如同主屏幕一样，`s.refresh()` 实际上显示了画到窗口对象上的元素。

绘制了下拉菜单后，程序使用简单的 `s.getch()` 调用来获取用户的选择项。在演示应用程序中，菜单只响应热键，但不响应箭头键或可移动突出显示栏。可以通过捕捉附加键操作并在下拉菜单中设置事件循环来构建这些更复杂的菜单功能。但这个例子已经足够说明这种概念了。

接着，程序将刚读取的击键与各种热键值做比较。在本例中，热键的大小写都可以激活下拉菜单选项，并且可以使用 `ENTER` 键激活缺省选项。（`curses` 特殊键常量看上去并不完全可靠，我发现必须添加实际的 ASCII 值“10”来捕捉 `ENTER` 键。）请注意，如果要执行字符值比较，那么要将字符串封装到 `ord()` 内置 Python 函数中。

当选中“Input”选项时，程序会使用 `getstr()` 方法，该方法提供带有原始编辑能力的字段输入（可以使用退格键）。由 `ENTER` 键终止输入，然后方法返回输入的值。通常会像上例中一样，将这个值分配给一个变量。

为了在视觉上区别输入字段，我使用了一点小技巧，预先向将要发生数据输入的区域添加了下划线。无论如何，这都是必要的，但它添加了一种视觉效果。由以下这行代码画出下划线：

```
screen.addstr(5, 33, " " * 43, curses.A_UNDERLINE)
```

当然，程序还必须除去下划线，这项工作在 `draw_dict()` 刷新函数中由以下这行代码执行：

```
screen.addstr(5, 33, " " * 43, curses.A_NORMAL)
```


第 15 章 Python 中的 TK 编程

本章我们要向读者介绍能想像到的开始 GUI 编程的最简单方法，就是使用 Scriptics 的 TK 和 Tkinter 封装器。我们将与上一节中的“Python 中的 curses 编程”提到的 curses 库进行很多比较。除了 curses 实现文本控制台而 TK 实现 GUI 这一差别之外，这两个库有着惊人相似的接口。在使用任何一个库之前，需要基本了解窗口和事件循环，并参考可用的窗口小部件。

如同关于 curses 的文章，本章仅讨论 Tkinter 本身的特性。既然很多 Python 发行版都带有 Tkinter，因此可能无需下载支持库或其他 Python 模块。本章后面的参考资料指向几个更高级别的用户接口窗口小部件的集合，但是你可以用 Tkinter 本身做许多事，包括构造自己的高级窗口小部件。学习基本 Tkinter 模块将为你引入 TK 的思维方式，即使你继续使用更高级的窗口小部件集合，这种思维方式仍十分重要。

15.1 TK 简要描述

TK 是与 TCL 语言关系最密切、且被广泛使用的图形库，TCL 语言和 TK 都由 John Ousterhout 开发。虽然 TK 于 1991 年作为 X11 库出现，但实际上它从那时起就被移植到每一种流行的 GUI。（它与 Python 逐渐拥有“标准”GUI 的情形相似。）现在，大多数流行语言和很多小型语言都有 TK 绑定（Tkinter 模块）。

在下面的小节中我们将使用 Txt2Html。虽然可以用几种方式运行 Txt2Html，但这里的封装器却要从命令行运行 Txt2Html。该应用程序以批处理进程的形式运行，并带有指出要执行的转换各方面特性的命令行自变量。（以后，最好为用户提供交互式选择屏幕选项，以在执行实际转换之前引导用户逐步选择不同的转换选项并提供所选选项的可视反馈。）tk_txt2html 基于带有下拉菜单和嵌套子菜单的顶部菜单。旁边有详细的实现说明，它看起来与在“Python 中的 Curses 编程”中讨论的 curses 版本很相象。虽然 TK 用较少的代码可以实现更多的功能，但很明显，tk_txt2html 和 curses_txt2html 很相似。例如，在 TK 中，像菜单这样的特性可以依靠内置的 Tkinter 类实现，而无需从头编写。

除了设置配置选项之外，TK 封装器还包括一个与 TK Text 窗口小部件一起构建的滚动

帮助框（一个带有 Message 窗口小部件的“关于”框）和一个进行 TK 动态几何管理的历史窗口。与大多数交互式应用程序一样，封装器用 TK 的 Entry 窗口小部件接受某些用户输入。

15.2 基本知识

15.2.1 最小的 [Tkinter] 程序

实际上，Tkinter 程序只需做三件事：

```
import Tkinter      # import the Tkinter module
root = Tkinter.Tk() # create a root window
root.mainloop()    # create an event loop
```

这是一个完全有效的 Tkinter 程序（不要介意它没有实际用处，因为它甚至不管理“hello world”）。该程序唯一需要做的是创建一些容纳其根窗口的窗口小部件。这样增强之后，无需程序员进一步干涉，该程序的 `root.mainloop()` 方法调用就可以处理所有用户交互。

15.2.2 `main()` 函数

现在，我们看一下 `tk_txt2html.py` 更现实的 `main()` 函数。

例 1: `tk_txt2html main()` 函数

```
def main():
    global root, history_frame, info_line
    root = Tkinter.Tk()
    root.title('Txt2Html TK Shell')
    init_vars()

    #-- Create the menu frame, and menus to the menu frame
    menu_frame = Tkinter.Frame(root)
    menu_frame.pack(fill=Tkinter.X, side=Tkinter.TOP)
    menu_frame.tk_menuBar(file_menu(), action_menu(), help_menu())

    #-- Create the history frame (to be filled in during runtime)
    history_frame = Tkinter.Frame(root)
    history_frame.pack(fill=Tkinter.X, side=Tkinter.BOTTOM, pady=2)
```

```

#-- Create the info frame and fill with initial contents
info_frame = Tkinter.Frame (root)
info_frame.pack (fill=Tkinter.X, side=Tkinter.BOTTOM)

# first put the column labels in a sub-frame
LEFT, Label = Tkinter.LEFT, Tkinter.Label # shortcut names
label_line = Tkinter.Frame (info_frame, relief=Tkinter.RAISED,
borderwidth=1)
label_line.pack (side=Tkinter.TOP, padx=2, pady=1)
Label (label_line, text="Run #", width=5) .pack (side=LEFT)
Label (label_line, text="Source:", width=20) .pack (side=LEFT)
Label (label_line, text="Target:", width=20) .pack (side=LEFT)
Label (label_line, text="Type:", width=20) .pack (side=LEFT)
Label (label_line, text="Proxy Mode:", width=20) .pack (side=LEFT)

# then put the "next run" information in a sub-frame
info_line = Tkinter.Frame (info_frame)
info_line.pack (side=Tkinter.TOP, padx=2, pady=1)
update_specs ()

#-- Finally, let's actually do all that stuff created above
root.mainloop ()

```

在这个简单的 `main()` 函数中有几件事要注意：

每一个窗口小部件都有一个父代。每当创建窗口小部件时，传递给实例创建的第一个自变量是新的窗口小部件的父代。

如果有其他窗口小部件创建自变量，将通过名称传递它们。Python 的这一特性给我们以指定选项或允许它们取缺省值的极大灵活性。

有几个窗口小部件实例 (Frame) 是全局变量。可以通过在函数间传递变量来使它们成为本地变量，以便维护代码范围的理论纯洁性，但是与它的实际用处相比过于麻烦。另外，使这些基本的 UI 元素全局化强调了这样一个事实：它们可以在整个函数中使用。但是，要确保对自己的全局变量使用良好的命名规范。（事先给你一个警告，Python 人员看起来讨厌匈牙利符号）。

创建完窗口小部件之后，我们调用一个几何图形管理器来让 TK 知道在哪里放置窗口小

部件。TK 在计算细节信息时有许多魔力，特别是当调整窗口大小或动态添加窗口小部件时更是如此。

15.2.3 应用几何图形管理器

TK 提供三个几何图形管理器：`.pack()`、`.grid()` 和 `.place()`。虽然 `.place()` 可用于精细（换句话说，非常复杂）的控制，但 `tk_txt2html` 只使用头两个。大多数时候，你将使用 `.pack()`。

当然，可以不带自变量来调用 `.pack()` 方法。但是如果那样做，窗口小部件可能会在显示屏的某处结束，你可能也想为 `.pack()` 提供一些提示。这些提示中最重要的是 `side` 自变量。可能的值是 `LEFT`、`RIGHT`、`TOP` 和 `BOTTOM`（请注意这些是 Tkinter 名称空间中的变量）。

`.pack()` 的许多魔力来自可以将窗口小部件嵌套这一事实。特别地，除了作为其他窗口小部件的容器（它有时显示不同类型的边界）之外，`Frame` 窗口小部件几乎不做什么。这样，就可以很方便地在所希望的方向排列几个框架，然后在每个框架中添加其他窗口小部件。按照调用框架（以及其他窗口小部件）的 `.pack()` 方法的顺序来排列它们。因此，如果两个窗口小部件都请求 `side=TOP`，则满足先进入的请求。

`tk_txt2html` 还偶尔使用 `.grid()`。`grid` 几何图形管理器用可视的坐标线覆盖父代窗口小部件。当窗口小部件调用 `.grid(row=3, column=4)` 时，它请求其父代将它放在第三行第四列上。通过查看父代的所有子代的请求来计算父代的总行数和总列数。

别忘了对自己的窗口小部件应用几何图形管理器，以免在显示屏幕上看不到它们时后悔莫及。

15.2.4 菜单

Tkinter 能轻易生成菜单。虽然我们在这里使用十分简单的示例，但是如果愿意，还可以用不同的字体、图形、复选框和各种别致的子代窗口小部件来填充菜单。在我们的示例中，`tk_txt2html` 的菜单全部用我们在上面所见的行创建。

```
menu_frame.tk_menuBar (file_menu (), action_menu (), help_menu ())
```

这行本身可能有些神秘。大多数必须完成的工作位于名为 `*_menu()` 的函数中。让我们看一下最简单的示例。

例 2: 创建下拉菜单

```
def help_menu():
    help_btn = Tkinter.Menubutton(menu_frame, text='Help', underline=0)
    help_btn.pack(side=Tkinter.LEFT, padx="2m")
    help_btn.menu = Tkinter.Menu(help_btn)
    help_btn.menu.add_command(label="How To", underline=0, command=HowTo)
    help_btn.menu.add_command(label="About", underline=0, command=About)
    help_btn['menu'] = help_btn.menu
    return help_btn
```

下拉菜单是将 Menu 小窗口部件作为子代的 Menubutton 小窗口部件。pack() (或 .grid() 等) 将 Menubutton 排列在适当的位置。Menu 小窗口部件用 .add_command() 方法添加项。

15.2.5 接受用户输入

下面将看到的示例演示 Label 小窗口部件 widget 如何显示输入。字段输入的基本小窗口部件是 Entry。它易于使用，但是如果以前曾使用过 Python 的 raw_input() 或 curses 的 .getstr()，你将发现技巧略有不同。TK 的 Entry 小窗口部件不返回可分配的值。相反，它获取自变量来填充字段对象。例如，下面的函数允许用户指定输入文件。

例 3: 接受用户字段输入

```
def GetSource():
    get_window = Tkinter.Toplevel(root)
    get_window.title('Source File?')
    Tkinter.Entry(get_window, width=30,
                  textvariable=source).pack()

    Tkinter.Button(get_window, text="Change",
                   command=lambda: update_specs()).pack()
```

这里有几件事要注意：我们为这个输入创建了一个新的 Toplevel 小窗口部件和对话框，并且通过创建一个带有 textvariable 自变量的 Entry 小窗口部件指定了输入字段。但是 textvariable 自变量没有指定简单的字符串变量。相反，它引用一个 StringVar 对象。在我们

的示例中，从 `main()` 调用的 `init_vars()` 函数包含三行。

```
source = Tkinter.StringVar ()
source.set ('txt2html.txt')
```

上面创建了一个适用于用户输入的对象并为其分配了初始值。每次在与之相链接的 **Entry** 小窗口部件中进行更改时都立即修改该对象。每次在 **Entry** 小窗口部件中击键、而不是读取终止时，都进行 `raw_input()` 样式的更改。要想获得用户输入的值，我们使用 **StringVar** 实例的 `.get()` 方法。

例如：

```
source_string = source.get ()
```

第四部分

附 录



附录 A 交互式输入编辑及代换过去的内容

有些的 Python 的直译器版本有支持目前所在行的编辑，以及过去输入内容的代换，这跟在 Korn Shell 以及 GNU Bash shell 里面的一些功能有点像。这个功能是用 GNU Readline 的链接库做出来的，提供了类似 Emacs 以及类似 vi 的编辑功能。这个链接库本身有自己的参考文件，所以我在此不再重复，我只是简单的介绍基本的功能。这里所介绍的互动模式输入编辑及代换过去内容的功能，通常在 Unix 以及 CygWin 的直译器版本都可以见得到。

本章没有包含在 Mark Hammond 的 PythonWin package 里的编辑功能，也没有包含在标准 Python distribution 里面的 Tk 之类的环境，或是 IDLE 的编辑功能。在 DOS 以及 NT 及其他类似的环境下，也有命令列过去内容记忆的功能，但是也不在本文的内容之中。

A.1 整行编辑

如果支持整行编辑的话，当直译器印出 primary prompt 或是 secondary prompt 的时候，整行编辑的功能就会被激活起来。你可以用一般的 Emacs 控制字符来编辑目前所在的行。其中常见的有这些：C-A (Control-A) 会移动 cursor 到目前行的最开头的地方，C-E 会移动到最尾端，C-B 会往左移动一个位置，C-F 会往右边一个位置。Backspace 键会消去目前 cursor 所在处左边的一个字符，C-D 会消去右边的一个字符，C-K 会杀掉 (消去) 本行在光标之后的所有字符，C-Y 会把刚刚所杀掉的字符串再次贴回来，C-underscore 会取消刚才所做的最后一个动作，这个功能也可以重复多次的使用。

A.2 代换过去的内容

要取代过去的内容方法如下：所有的非空白行都会被储存在一个储存过去内容的缓冲区 (history buffer) 里面，当 prompt 出现的时候，你所处的位置就是在这个 buffer 的最底下，使用 C-P 会使得在 buffer 里面往上移动一行，C-N 会往下移动一行。任何在 buffer 里面的行都可以被编辑，若是在 prompt 之前出现星号的话就代表这个行已经被修改过了。当你按下 Return 键的时候就是把目前这一行送给直译器了。C-R 会开始一个逐渐往上的搜寻，按下

C-S 会开始往前的搜寻。

A.3 键盘连接

键盘的连接以及一些其他有关 **Readline library** 的参数都可以被修改，其方法是在一个叫做 `~/.inputrc` 的初始化文件中打入一些指令。键盘的连接有以下的几种形式：

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

或是

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

你也可以设定一些选项：

```
set option-name value
```

请看下面的例子：

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

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

# Rebind some keys:
Meta h: backward-kill-word

"\C-u": universal-argument

"\C-x\C-r": re-read-init-file
```

需要注意的是，在 **Python** 里面预设的 **Tab** 键所连接的是输入一个 **Tab** 字符，而非 **Readline** 链接库里面预设的文件名自动完成的功能。如果你坚持的话，你也可以这样子设定来覆盖 **Python** 的预设：

```
Tab: complete
```

这应该在你的 `~/.inputrc` 里面。（当然如果你这样做的话，缩排连续行的时候就费力一些了。）

自动完成变量及 **module** 的名称的功能是可以自由选择要或不要的。如果要在互动模式下激活这个功能的话，可以在激活文件里面加入下面的指令：

```
import rlcompleter, readline
```

```
readline.parse_and_bind('tab: complete')
```

这样做会使TAB 键连接到完成的功能,所以按下TAB 键两次就会建议一些完整的名称。其搜寻的是Python 的语句名称、目前的local 变量,以及可用的module 名称。对于像string.a 这样带有点的expression,这项功能会先是着先evaluate 到最后一个点的意思,然后再从所得的对象中建议可用的变量名称。要注意的是,如果此对象含有的__getattr__() 这个method 且是此expression 的一部分的话,就会执行一些特定的程序代码。

A.4 评注

这个功能跟其他以前的直译器版本比起来是很大的一个进步,但是我们还有很多希望有的功能,如在连续行的时候可以建议适当的缩排距离(分析器(parser)应该知道何时需要缩排);自动完成的功能应该也能使用直译器的symbol table;也要有一个指令可以检查(甚至是建议)何时应该要有结束的括号等等。

附录 B Python 资源

1. Python 官方站点

<http://www.python.org>

这个站点是 Python 的中心，所有的 Python 的资源均可以这个站上找到。包括 Python 的最新发布版、文档，和其他的相关资料。

2. 新闻组，讨论组和 EMAIL 帮助

[comp.lang.python/python-list](comp.lang.python.python-list)

[comp.lang.python.announce/python-list-announce](comp.lang.python.announce.python-list-announce)

python-help@python.org

3. python 专业服务公司

<http://www.pythonpros.com>

4. python 爱好者的国内主页

西奈酒廊之 python 园地: <http://python.3322.net>

python 爱好者: <http://pythonfans.yeah.net>

Python-Chinese 中文支撑站点: <http://jhonywoo.at.china.com>

程序设计实现: <http://python123.yeah.net>

附录 B Python 资源

1. Python 官方站点

<http://www.python.org>

这个站点是 Python 的中心，所有的 Python 的资源均可以这个站上找到。包括 Python 的最新发布版、文档，和其他的相关资料。

2. 新闻组，讨论组和 EMAIL 帮助

[comp.lang.python/python-list](comp.lang.python.python-list)

[comp.lang.python.announce/python-list-announce](comp.lang.python.announce.python-list-announce)

python-help@python.org

3. python 专业服务公司

<http://www.pythonpros.com>

4. python 爱好者的国内主页

西奈酒廊之 python 园地: <http://python.3322.net>

python 爱好者: <http://pythonfans.yeah.net>

Python-Chinese 中文支撑站点: <http://jhonywoo.at.china.com>

程序设计实现: <http://python123.yeah.net>

附录 B Python 资源

1. Python 官方站点

<http://www.python.org>

这个站点是 Python 的中心，所有的 Python 的资源均可以这个站上找到。包括 Python 的最新发布版、文档，和其他的相关资料。

2. 新闻组，讨论组和 EMAIL 帮助

[comp.lang.python/python-list](comp.lang.python.python-list)

[comp.lang.python.announce/python-list-announce](comp.lang.python.announce.python-list-announce)

python-help@python.org

3. python 专业服务公司

<http://www.pythonpros.com>

4. python 爱好者的国内主页

西奈酒廊之 python 园地: <http://python.3322.net>

python 爱好者: <http://pythonfans.yeah.net>

Python-Chinese 中文支撑站点: <http://jhonywoo.at.china.com>

程序设计实现: <http://python123.yeah.net>

附录 B Python 资源

1. Python 官方站点

<http://www.python.org>

这个站点是 Python 的中心，所有的 Python 的资源均可以这个站上找到。包括 Python 的最新发布版、文档，和其他的相关资料。

2. 新闻组，讨论组和 EMAIL 帮助

[comp.lang.python/python-list](comp.lang.python.python-list)

[comp.lang.python.announce/python-list-announce](comp.lang.python.announce.python-list-announce)

python-help@python.org

3. python 专业服务公司

<http://www.pythonpros.com>

4. python 爱好者的国内主页

西奈酒廊之 python 园地: <http://python.3322.net>

python 爱好者: <http://pythonfans.yeah.net>

Python-Chinese 中文支撑站点: <http://jhonywoo.at.china.com>

程序设计实现: <http://python123.yeah.net>

附录 B Python 资源

1. Python 官方站点

<http://www.python.org>

这个站点是 Python 的中心，所有的 Python 的资源均可以这个站上找到。包括 Python 的最新发布版、文档，和其他的相关资料。

2. 新闻组，讨论组和 EMAIL 帮助

[comp.lang.python/python-list](comp.lang.python.python-list)

[comp.lang.python.announce/python-list-announce](comp.lang.python.announce.python-list-announce)

python-help@python.org

3. python 专业服务公司

<http://www.pythonpros.com>

4. python 爱好者的国内主页

西奈酒廊之 python 园地: <http://python.3322.net>

python 爱好者: <http://pythonfans.yeah.net>

Python-Chinese 中文支撑站点: <http://jhonywoo.at.china.com>

程序设计实现: <http://python123.yeah.net>

附录 B Python 资源

1. Python 官方站点

<http://www.python.org>

这个站点是 Python 的中心，所有的 Python 的资源均可以这个站上找到。包括 Python 的最新发布版、文档，和其他的相关资料。

2. 新闻组，讨论组和 EMAIL 帮助

[comp.lang.python/python-list](comp.lang.python.python-list)

[comp.lang.python.announce/python-list-announce](comp.lang.python.announce.python-list-announce)

python-help@python.org

3. python 专业服务公司

<http://www.pythonpros.com>

4. python 爱好者的国内主页

西奈酒廊之 python 园地: <http://python.3322.net>

python 爱好者: <http://pythonfans.yeah.net>

Python-Chinese 中文支撑站点: <http://jhonywoo.at.china.com>

程序设计实现: <http://python123.yeah.net>