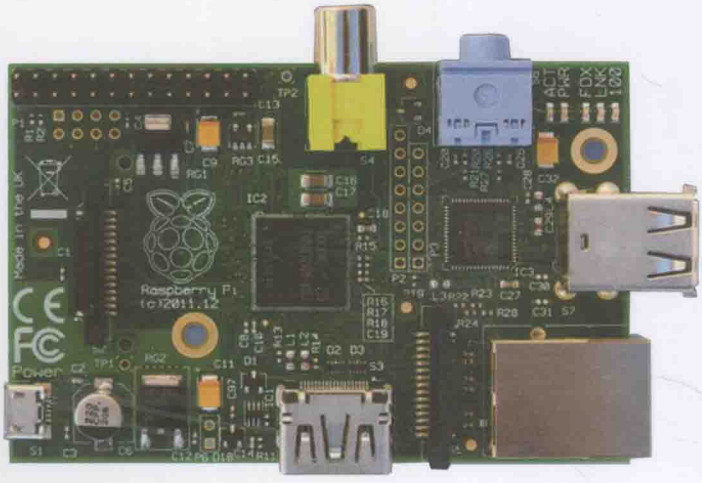
HZ BOOKS
华章科技

树莓派基金会资深软件开发工程师亲笔撰写，系统阐释在树莓派上使用Python开发游戏、多媒体等的实用工具、方法和最佳实践

深入剖析Python常见开发问题，包含大量实践案例，可操作性强，能为用户使用树莓派高效编写Python程序提供有效指导

数字匠人

WILEY



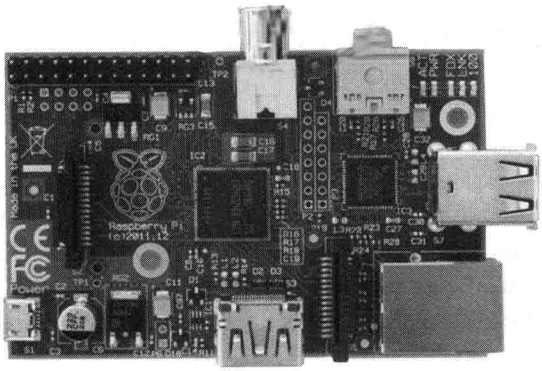
Learning Python
with Raspberry Pi

树莓派
Python编程指南

[美] Alex Bradbury　Ben Everard 著　王文峰 译

机械工业出版社
China Machine Press

数字匠人



Learning Python
with Raspberry Pi

树莓派
Python编程指南

[美] Alex Bradbury　Ben Everard 著　王文峰 译

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

树莓派 Python 编程指南 /（美）布拉德伯里（Bradbury，A.）等著；王文峰译 . —北京：机械工业出版社，2015.1
（数字匠人）
书名原文：Learning Python with Raspberry Pi

ISBN 978-7-111-48986-3

I. 树… II. ①布… ②王… III. 软件工具 – 程序设计 – 指南 IV. TP311.56-62

中国版本图书馆 CIP 数据核字（2014）第 303346 号

本书版权登记号：图字：01-2014-3575

Copyright © 2014 Alex Bradbury and Ben Everard

All Rights Reserved. This translation published under license. Authorized translation from the English language edition, entitled Learning Python with Raspberry Pi, ISBN 978-1-118-71705-9, by Alex Bradbury and Ben Everard, Published by John Wiley & Sons. No part of this book may be reproduced in any form without the written permission of the original copyrights holder.

本书中文简体字版由约翰 - 威利父子公司授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

本书封底贴有 Wiley 防伪标签，无标签者不得销售。

树莓派 Python 编程指南

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：秦　健　　责任校对：殷　虹
印　　刷：北京市荣盛彩色印刷有限公司　　版　　次：2015 年 1 月第 1 版第 1 次印刷
开　　本：186mm × 240mm　1/16　　印　　张：14
书　　号：ISBN 978-7-111-48986-3　　定　　价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

版权所有 • 侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所　韩光 / 邹晓东

Preface 前　言

计算机已经不再只是用来工作的方盒子。任何拥有一个可编程处理中心的东西都可以称作计算机。游戏终端、智能手机、全球卫星定位系统（GPS）单元、平板电脑以及其他许多令人难以置信的设备都说明了这一点。它们都是计算机，并且它们已经接管了这个世界。我们用它们来工作、通信，以及娱乐。事实上，很难找到有哪个领域还从未使用过计算机。

销售人员喜欢用智能来描述嵌入式计算机设备（智能手机、智能电视、智能手表等），事实上它们却并不智能。处理单元只是块执行指令的硅片。智能手机的“智能”并非来自计算机芯片，而是来自对它们编程的人。

计算机是迄今为止人类发明的最强大的工具，但是由于很少有人知道如何开发它的潜能，目前我们只是使用了它的一小部分功能。在充满计算机的世界中，最重要的就是那些可以发挥计算机全部性能的编程人员。编程，是一项基本技能，并且在未来会变得更重要。

什么是编程

我们已经指出，计算机并不智能。它只是个按照清单一步步执行指令的单元。这个指令清单就是程序。编程，就是接受任务，将其分解成多个步骤，然后把它们用计算机可以理解的语言写下来。

树莓派可以理解多种语言，在本书中，你将学习到 Python3——一种非常强大易学的语言。

本书适用于拥有树莓派并希望学些计算机编程的读者。学习本书不要求读者具有编程经验或者其他类似的技术。即便你只爱看漫画和科幻小说，都没关系，只要你具备前两个基本条件，这本书就是为你准备的。

读完本书，你将会对 Python3 有很深的理解，并且熟悉许多非常有用的模块（Python 附加模块）。通过 Python3 和这些模块，你将能控制树莓派的各个功能。你可以通过控制基本输入输出口（GPIO）使它和外界交互，或者用它连通互联网。拥有一个摄像头，你可以用树莓派拍照片，也可以制作游戏，操纵三维世界。简而言之，这是一本介绍如何发掘你的树莓派的全部潜力的图书。

什么是树莓派

树莓派是个用来学习编程的优秀设备。第一，它很便宜。价格差不多只有低端计算机的十分之一，作为你的主计算机的一个附件它非常便宜。由于程序员往往需要调试开发设备，而调试就有可能破坏某些东西，这就使树莓派显得很有用。通常来说，调试并不会破坏到设备本身，但是可能需要重装系统，这就有可能损失数据并且在几个小时内你都无法使用计算机。如果你有个树莓派，用它来编程，这些就不是问题。如果你的计算机需要和别人共享，使用树莓派就显得更重要。

第二，树莓派是个裸设备。它没有藏在盒子里或者在一个完整的系统中。这意味着你可以自己决定将其做成一个什么系统。你可以将其放到盒子里，也可以就让它裸着运行。你也可以使用 GPIO，这在其他机器上是无法做到的。大多数计算机的用途都已经预先设计好了（如用来网上冲浪或者玩游戏的平板电脑，用来看电影或玩游戏的游戏终端，用来工作或玩游戏的笔记本电脑等）。而只要一点技术手段树莓派就可以做任何事情。

第三，树莓派使用 Linux——一个类似于 Windows 或 Mac OS X 的操作系统。它提供了视窗系统和一个用来操控树莓派的基于文本的命令行接口。如果你之前没有使用过 Linux，会发现它和你使用过的系统有点不同。对于崭露头角的程序员来说，最重要的区别就是 Linux 比其他系统更灵活。正如树莓派的物理设计崇尚体验一样，这个操作系统也是如此。

如何阅读本书

前 3 章介绍了如何在树莓派上使用 Python。读完这 3 章后，你将对 Python 编程有个很好的认识。本书剩余部分将分章来介绍不同的应用，如游戏和多媒体。这些章节涉及 Python 的不同领域。因此前一章没有读完不会影响你对后一章的理解（有些时候，我们可能会引用前面的某些概念，但我们会在引用时标注清楚）。

这意味着你可以在读本书第二部分时自己决定阅读顺序。例如，如果你对多媒体很感兴趣，可以直接跳到这一章，之后再去读其他章节。

学习编程必须多动手实践。也就是说，仅仅坐下来读完本书是不够的。你必须去实践这些学到的东西。贯穿本书，我们设计了很多练习让你来实践所学到的知识。有时通过特定的练习来培养你的技能，有时你需要给我们介绍过的程序添加特性。编程的一个重要部分就是确定程序要完成什么的创造力。因此你不需要完全听从我们的建议。事实上，我们鼓励你把我们的建议和代码作为一个起点——一个开启你的数字艺术征程的起点。

致　谢 *Acknowledgements*

感谢大家帮助我完成此书。在 Wiley 公司，Kezia Ednsley 和 Craig smith 从本书开始写作起就帮忙审稿。感谢 Erin Zeltner，是他让本书的文字看起来更优美，也让排版更合理。

还有许多人需要感谢。没有编程环境就不会有这本关于编程的书。基于树莓派的 Python 包含了数以千计的程序员的工作，其中有很多人还是免费贡献。这些人都需要感谢，受限于篇幅，我们仅列出三位：Guido van Rossum、Linux Torvalds 和 Richard Stallman。

当然，软件需要运行在硬件之上，我们也要感谢 Eben Upton 和树莓派基金会。

本书中的任何错误都由作者独自承担。

Contents 目　录

前言

致谢

第 1 章　起航（启动和运行） …… 1

1.1　组装好你的树莓派 …… 1

1.2　可能遇到的问题 …… 2

1.3　树莓派快速指南 …… 3

1.3.1　使用 LXDE（轻量级 X11 桌面环境） …… 3

1.3.2　使用终端 …… 4

1.3.3　通过 Raspi-Config 改变配置 …… 6

1.3.4　安装软件 …… 6

1.4　Python3 …… 6

1.4.1　Python 解释器 …… 7

1.4.2　运行 Python 程序 …… 7

1.5　小结 …… 8

第 2 章　Python 简介 …… 9

2.1　使用 Turtles 绘画 …… 9

2.1.1　使用循环 …… 12

2.1.2　条件处理：if、elif 和 else …… 14

2.1.3　使用函数和方法组织代码 …… 15

2.2　一个 Python 游戏：猫和老鼠 …… 16

2.2.1　理解变量 …… 19

2.2.2　定义函数 …… 19

2.2.3　在游戏中循环 …… 19

2.3　小结 …… 20

第 3 章　Python 基础 …… 22

3.1　变量、值和类型 …… 22

3.1.1　值和类型 …… 23

3.1.2　数字排序 …… 24

3.1.3　使用 Strings 保存文字 …… 25

3.1.4　布尔值：真或假 …… 25

3.1.5　数据类型转换 …… 26

3.1.6　知识测试 …… 26

3.2　在结构体中存储值 …… 27

3.2.1　字典和集合中的非序列元素 …… 30

3.2.2　知识测试 …… 31

3.3　控制程序流程 …… 32

3.3.1　用循环遍历数据 …… 32

3.3.2　深入理解循环嵌套 …… 33

3.3.3　使用 if 语句控制程序分支 …… 34

3.3.4 捕获异常 …… 35
3.4 使用函数复用代码 …… 36
3.5 组合装配 …… 38
3.6 使用类来构建对象 …… 40
3.7 使用模块获得附加特性 …… 45
3.8 小结 …… 46
3.9 习题答案 …… 47

第 4 章 图形编程 …… 48

4.1 图形用户界面（GUI）编程 …… 48
4.2 添加控制 …… 50
4.3 创建 Web 浏览器 …… 52
4.4 添加窗口菜单 …… 60
4.5 小结 …… 62
4.6 习题答案 …… 62

第 5 章 搭建游戏 …… 65

5.1 构建游戏 …… 66
5.2 初始化 PyGame …… 69
5.3 为角色创建世界 …… 73
5.3.1 检测冲突 …… 74
5.3.2 左右移动 …… 76
5.3.3 达到目标 …… 78
5.3.4 制造挑战 …… 79
5.4 在游戏中加入自己的风格 …… 83
5.5 添加音乐 …… 83
5.6 添加布景 …… 84
5.7 让游戏更上一层楼 …… 87
5.8 逼真的游戏物理 …… 87
5.9 小结 …… 91

第 6 章 使用 OpenGL 创建图形 …… 92

6.1 获取模块 …… 93
6.2 创建旋转立方体 …… 93
6.2.1 向量和矩阵 …… 95
6.2.2 组合包装 …… 98
6.2.3 增加光照 …… 101
6.3 让屏幕起舞 …… 107
6.3.1 建立 3D 模型 …… 109
6.3.2 计算声音强度 …… 110
6.4 继续完善 …… 115
6.5 添加纹理 …… 115
6.6 小结 …… 116

第 7 章 Python 与网络 …… 117

7.1 理解主机、端口和套接字 …… 117
7.1.1 使用 IP 地址定位计算机 …… 117
7.1.2 搭建会话服务器 …… 118
7.1.3 “推”向世界 …… 121
7.1.4 使用 JSON 做天气预报 …… 123
7.2 知识测验 …… 125
7.3 走向网站 …… 125
7.3.1 让网站动起来（动态网站）… 127
7.3.2 使用模板 …… 128
7.3.3 使用表格回传数据 …… 129
7.4 安全 …… 131
7.5 小结 …… 134
7.6 习题答案 …… 134

第 8 章 我的世界 …… 137

8.1 畅游我的世界 …… 138

8.1.1 控制我的世界 …… 138
8.1.2 用 Python 创建我的世界 …… 139
8.1.3 深入探索 …… 142
8.2 制作贪吃蛇游戏 …… 142
8.2.1 移动贪吃蛇 …… 146
8.2.2 增长贪吃蛇 …… 146
8.2.3 添加苹果 …… 146
8.3 深入探索 …… 147
8.4 小结 …… 148

第 9 章 多媒体 …… 149

9.1 使用 PyAudio 让计算机发声 …… 149
9.1.1 录音 …… 151
9.1.2 向树莓派讲话 …… 151
9.1.3 向程序提问 …… 152
9.1.4 组合包装 …… 153
9.1.5 深入探索 …… 155
9.2 制作电影 …… 155
9.2.1 使用 USB 网络摄像头 …… 155
9.2.2 使用 OpenCV 添加计算机图像特性 …… 158
9.2.3 深入探索 …… 160
9.2.4 使用树莓派摄像头模块 …… 160
9.2.5 创建直播视频 …… 162
9.2.6 深入探索 …… 165
9.3 小结 …… 165

第 10 章 脚本 …… 166

10.1 从 Linux 命令行开始 …… 166
10.1.1 使用 subprocess 模块 …… 168
10.1.2 命令行标签 …… 169
10.1.3 正则表达式 …… 170
10.2 知识测验 …… 173
10.3 脚本中的网络 …… 174
10.4 组合包装 …… 175
10.5 在 Python 中操作文件 …… 180
10.6 小结 …… 182

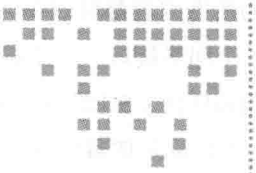
第 11 章 硬件接口 …… 183

11.1 硬件设置选择 …… 183
11.1.1 母转公接头 …… 183
11.1.2 无焊面包板 …… 184
11.1.3 成品板和万能板 …… 185
11.1.4 PCB 加工 …… 185
11.2 辅助工具 …… 185
11.2.1 剪线 / 剥线器 …… 185
11.2.2 万用表 …… 185
11.2.3 电烙铁 …… 185
11.3 本章所需的硬件 …… 186
11.3.1 第一个电路 …… 186
11.3.2 保护树莓派 …… 189
11.3.3 电源限制 …… 190
11.3.4 获得输入 …… 191
11.4 使用 I2C、SPI 和串口扩展 GPIO …… 192
11.4.1 SPI 通信协议 …… 193
11.4.2 I2C 通信协议 …… 196
11.4.3 串口通信协议 …… 196
11.5 深入研究 …… 196
11.5.1 Arduino …… 197
11.5.2 PiFace …… 197
11.5.3 Gertboard …… 197

11.5.4 Wireless Inventor's Kit …… 198
11.6 尝试一些流行工程 …… 198
11.6.1 机器人 …… 198
11.6.2 家庭自动化 …… 198
11.6.3 防盗报警器 …… 199
11.6.4 数字艺术 …… 199
11.7 小结 …… 199

第 12 章 测试与调试 …… 200

12.1 通过打印变量调查故障 …… 200
12.2 通过测试发现故障 …… 203
12.2.1 使用单元测试检查代码片段 …… 204
12.2.2 获得更多断言 …… 207
12.2.3 使用测试集进行回归测试 …… 209
12.2.4 测试整个程序包 …… 210
12.2.5 保证软件可用性 …… 210
12.3 究竟需要多少测试 …… 211
12.4 小结 …… 211



第 1 章 Chapter 1

起航（启动和运行）

欢迎阅读本书。本书从三维图像、游戏编程到控制电子学，再到推文，会让你学到如何解放这个小计算机的全部能量。你将会看到掩藏在表象之下的内部世界，并学会如何创建程序以发挥这台小计算机的全部特性。

1.1 组装好你的树莓派

为跟进本书，你需要一些设备：

- 树莓派
- USB 键盘
- USB 鼠标
- SD 卡
- 显示器
- 电源

还有一些有用的可选设备：

- 有源 USB 集线器（强烈推荐）
- 摄像头模块
- USB 网络摄像头
- USB 无线网络（WiFi）适配器

PDF电子书说明：

本人可以提供各种PDF电子书资料，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我 QQ: 461573687, 或者 QQ: 2404062482。

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ。因PDF电子书都有版权，请不要随意传播，最近pdf也越来越难做了，希望大家尊重下个人劳动，谢谢！

本书中的所有内容都可以在 A 版树莓派上完成。从编程角度而言，B 版树莓派的强大之处在于多了个网络接口。这个接口在你需要安装软件时可以方便地接入互联网。

树莓派兼容任何 USB 键盘、鼠标和大多数的 SD 卡。仅有少数 SD 卡可能会存在问题。如果不确定，可以通过树莓派网上商城购买（http://raspberrypi.org 上可以找到商店链接）。

树莓派拥有一个 HDMI（高清多媒体）视频输出端口，但是大多数显示器只有 VGA 或者 DVI 输入。如果可能，请选用具有 DVI 或者 HDMI 输入的显示器。一个 HDMI 转 DVI 接头只要几十块钱，并且其不会降低图像质量。市场上也有 HDMI 转 VGA 转接器，但是其价格昂贵且性能不稳定。所以只有在没有其他选择时才会使用这种转接头方案。

品牌厂商出品的 USB 电源都可以工作，而一些杂牌廉价 USB 电源可能会出现问题。如果可能，建议不要在电源方面太过节俭。当然，你也可以通过 USB 电缆连接普通计算机给树莓派供电。

有源 USB 集线器可以减少本章后边提到的电源相关的问题。并不是所有的 USB 集线器都是有源的，所以要保证你的 USB 集线器可以通过市电供电。

我们将在第 9 章中谈论如何选择摄像头。这里唯一要指出的是，如果你选择 USB 网络摄像头，请保证它兼容树莓派。部分支持树莓派的网络摄像头可以参考 http://elinux.org/RPi_USB_Webcams。

通过将树莓派用网线连接到路由器或者使用 USB 无线适配器连接到无线网络，你可以将树莓派接入互联网，并安装本书中需要用到的软件。

1.2 可能遇到的问题

树莓派第一个最常见的问题是电源相关的问题。有的 USB 电源无法提供足够的电流，尤其当树莓派连上外围设备或者你超频它时（详见第 5 章），使用这些电源可能会导致更多问题。电源相关的问题通常会导致计算机崩溃。因此，当发现树莓派运行不稳定时，首先要检查电源。为了避免这类问题，可以给树莓派单独使用一个电源，外围设备（如键盘、鼠标等）通过有源 USB 集线器连接到树莓派。

第二个最常见的问题是 SD 卡问题。这类问题可能是电源供电不足或者 SD 卡本身的问题。因此，我们有必要做好备份以保证数据安全。我们可以使用谷歌云硬盘服务来备份数据（在树莓派上运行有点慢），或者简单地在优盘中保存个副本。SD 卡相关问题通常表现在树莓派开机时显示出错信息。多数情况下，重装树莓派可以解决问题。如果重装无法解决，就需要更换新的 SD 卡。

如果遇到了其他问题，我们就需要深入挖掘了，通常，我们需要查看内核缓冲区和系统日志文件。如果遇到的是硬件相关问题，如 USB 设备无法工作，最好去看看内核缓冲区。打开终端模拟器 LXTerminal 并输入：

```
dmesg
```

这条命令将输出 Linux 内核的所有信息。它按照时间先后顺序显示，最后显示的是最近发生的事件。有任何问题都可以显示出来。

下面的命令可以打印出系统日志（通常称为 syslog）：

```
cat /var/log/syslog
```

同样，最新的消息仍然放在最后面。这两个信息可能有时会令人难以捉摸。如果读了之后仍然无法解决问题，最好去树莓派论坛上看看 www.raspberrypi.org/phpBB3/。这个论坛上有很多热心人，他们会帮助我们找到正确的方向。

1.3　树莓派快速指南

本书主要面向编程方向，而不是介绍如何使用树莓派。因此我们不会过多地介绍树莓派。但你会从中了解到树莓派是如何工作的。

有不少操作系统可以运行在树莓派上，本书的命令都是基于 Raspbian 的（一个树莓派默认的操作系统），也是初学者最好的选择[⊖]。如果有 Linux 的使用经验，可以选择 Arch 或者 Fedora。选择了其他系统，就需要将本书的 apt-get 命令转换为相应系统的命令以完成软件包管理。

在树莓派上安装 Raspbian 的最简便方法是使用 NOOBS，可以从 www.raspberrypi.org/dowloads 下载。网站上还有一份快速入门指南，它为树莓派的启动和运行提供了完整的帮助。

Raspbian 提供两种不同的交互方式——终端命令和图形系统（LXDE）。

1.3.1　使用 LXDE（轻量级 X11 桌面环境）

轻量级 X11 桌面环境是 Raspbian 的标准窗口系统。其基本组件和大多数的 Windows 8 之前的 Windows 窗口相似。屏幕左下角有一个按钮，可以打开应用程序菜单。当前运行的程序显示在该按钮右侧的长条上（参见图 1-1）。

⊖ 基于 Debian，由 Mike Thompson 和 Peter Green 针对树莓派硬件对 Debian 进行了专门优化和移植。——译者注

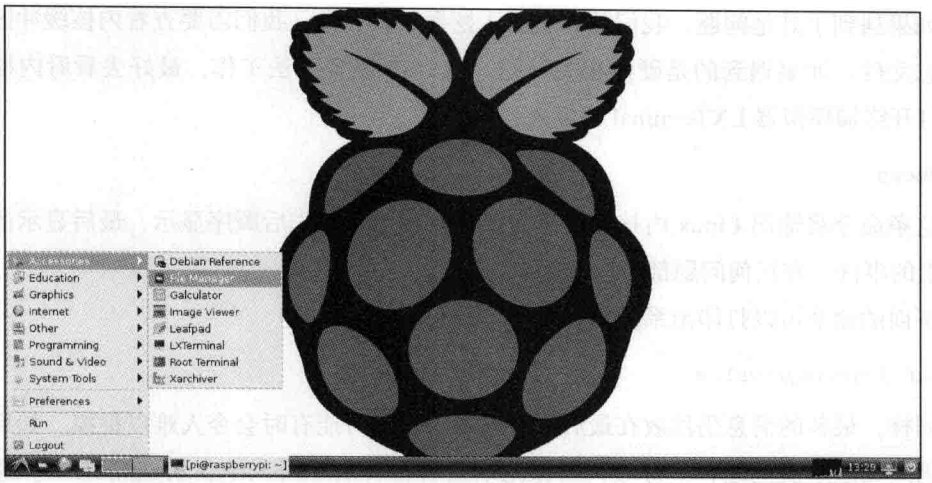


图 1-1 程序菜单打开的 LXDE 桌面

启动树莓派之后，如果你看到的是黑屏白字，询问是否登录，这表示还没有设置自动启动 LXDE。不用担心，只需使用用户名 pi，密码 raspberry 登录后输入如下命令就可以启动 LXDE。

```
startx
```

通过配置 raspi-config 可以设置成启动时自动运行 LXDE（参见下一节）。

1.3.2 使用终端

对于多数应用 LXDE 都很方便，但有时候我们也需要使用命令行。它是一个通过终端操作的非常强大的接口。在 LXDE 环境下，需要打开 LXTerminal 应用来使用它。

打开 LXTerminal，将会看到下面这行字：

```
pi@raspberrypi~$
```

它表示目前正使用用户名 pi 登录到名叫 raspberrypi 的计算机上，当前目录是 ~ 。

Linux 的所有目录都起始于 / 或者 root，它是目录树的基础，每个目录都位于根目录（root）的某个子目录下。cd（更改目录）命令可以在不同的目录间切换。下面的命令展示了如何切换到根目录：

```
cd /
```

执行这条命令后，命令提示符将变为：

```
pi@raspberrypi/$
```

ls 命令可以列出这个目录下的内容。有一个称为 home 的子目录，系统中的每个用户都有自己的 home 目录。执行下面的命令可以让我们进入 home 目录并查看其中的内容：

```
cd home
ls
```

此时，home 目录中只有一个目录：pi。并且命令提示符也提示我们当前正处于 /home 目录。

移动到我们仅有的一个子目录中：

```
cd pi
```

现在，命令提示符重新变回：

```
pi@raspberrypi~$
```

这是因为字符 ~ 是当前用户 home 目录的简写。在终端中输入 ~ 时，系统会将其转换为 /home/pi。

关于命令行还有许多值得学习的地方。要完整地介绍命令行的各个方面，需要再写一本和本书一样厚的书。然而，开始使用命令行时并不需要完全了解它。本书中在使用 LXTerminal 时，都会完整地列出要使用的命令行。



如果你希望学习到更多树莓派相关的知识或 Linux 通用知识，命令行是一个非常棒的开端。不论是在线资料还是印刷图书，都可以找到大量命令行相关的信息。Linux 命令行这本书可以免费在线浏览，这是一本非常棒的启蒙读物。网址 http://linuxcommand.org/tlcl.php。

这里我们有两个建议。首先，不要害怕命令行。开始使用时可能会有点头疼，但学习如何使用它的唯一方法就是多使用它。其次，几乎所有的命令都有内置帮助，通过选项 --help 可以查看这些帮助。例如，如果需要查看 ls 命令的更多用法，可以输入：

```
ls --help
```

该命令会输出：

```
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by
default). Sort entries alphabetically if none of -cftuvSUX nor
--sort is specified.
```

后面还会列出该命令可以使用的各种选项。

1.3.3 通过 Raspi-Config 改变配置

Raspbian 提供了一个帮助设置树莓派硬件的工具，称为 raspi-config。打开 LXTerminal 并输入如下命令就可以使用了：

```
sudo raspi-config
```

在这里，你会发现启动时自动启动 LXDE、超频树莓派等各种选项。超频树莓派会使本书中的一些例子运行得更好，尤其是在安装新软件时。

1.3.4 安装软件

在终端中使用 apt-get 可以给树莓派安装新软件。安装之前，最好把当前的所有软件升级到最新版本。下面是升级命令：

```
sudo apt-get update
sudo apt-get upgrade
```

然后就可以使用 apt-get 安装任何你需要的软件了。例如，如果想使用 iceweasel（火狐浏览器的 Debian 再发布版），可以打开 LXTerminal 并输入：

```
sudo apt-get install iceweasel
```

如果更喜欢使用图形界面来安装程序，可以安装 synaptic[⊖]：

```
sudo apt-get install synaptic
```

只需开启它就可以安装程序：

```
sudo synaptic
```

然后就可以直接搜索需要安装的东西。

安装软件需要在命令行前加上 sudo。这是为了告诉计算机，我们需要做些系统级别的变化，请给予程序足够的权限以完成这些操作。

1.4 Python3

本书中，你将会学到如何使用 Python3 这门编程语言。在 Raspbian 中使用这门语言有

⊖ Debian 及其衍生版本的包管理工具 apt 的图形化前端。——译者注

多种方式。

1.4.1　Python 解释器

有两种方式可以使用 Python，分别是 shell 交互和文本程序。shell 交互可以执行用户输入的每条指令，对于调试和实验非常有利。文本程序就是保存在文本文件中的 Python 代码，它可以一次性全部运行。很容易区分这两种运行方式。如果处于 shell 交互模式，每行都会以三个大于号开始：

```
>>>
```

本书大多数时候都使用程序文件。偶尔使用 shell（尤其是在早期时候）时我们会明确指出。为了明确区分代码运行环境，我们对运行在 shell 下的代码前都加上三个大于号。

1.4.2　运行 Python 程序

写 Python 程序有两种不同的方法。第一种方法是创建一个包含 Python 代码的文本文件，然后运行它。第二种方法是使用集成开发环境（IDE），如 IDLE3。以上方法的运行方式一样，结果也一样。可以根据个人爱好自由选择。

如果希望把程序写入文本文件，就需要一个文本编辑器如 Leafpad。文字处理软件如 LibreOffice 的 Writer 是不能用来写 Python 程序的，因为 Python 无法识别其使用的一些格式。下面给出一个例子，打开 Leafpad 并创建一个新文件，然后输入下面的文字：

```
print("Hello World!")
```

创建和保存文件时，请使用 .py 扩展名，如 testfile.py。要运行它，可以打开 LXTerminal 并移动到文件保存的子目录，执行 python < 文件名 >。使用 cd 命令可以移动到不同的目录。比如文件保存在当前 home 目录下的 programming 文件夹下，在 LXTerminal 中只执行下面命令就可以运行 python 程序了：

```
cd programming
python3 testfile.py
```

如果一切顺利，你将会在屏幕上看到下面一行字：

```
Hello World!
```

第二种方法更简单，IDE 中集成了文字编辑器和 Python 解析器。例如，打开 IDLE3（注意要带数字 3），然后点击 File → New Window。在新窗口中输入以下代码：

```
print("Hello IDLE")
```

继续点击 Run → Run Module。IDLE3 会提示是否保存该模块，选择文件名保存后，它将会返回到 Python 解释器并输出下列内容：

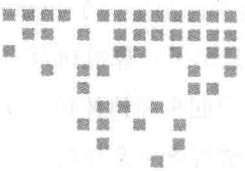
```
Hello IDLE
```

无论选择哪种方式来学习都可以，接下来，不妨选一个自己喜欢的方式开始吧。

1.5 小结

读完本章，你应该理解以下内容：

- 你需要一些额外的硬件以更好地利用树莓派。
- 供电不足是最常见的问题。
- 遇到问题时，最好从 dmesg 和 syslog 开始查找原因。
- Raspbian 使用 LXDE 桌面环境。
- 终端是和底层操作系统最有效的沟通方式。
- raspi-config 可以用来配置树莓派。
- apt-get 可以用来安装新软件。
- Python 可以通过 shell 交互或者文本程序运行。



第 2 章 Chapter 2

Python 简介

本章通过一些代码示例来逐步介绍 Python。你不必了解这些代码的所有细节，本章只是带你领略下编程的感觉。你将会学到如何在屏幕上绘画，甚至如何编写简单的游戏。同时，你将会学到一些编程的基本概念。如果无法完全理解本章中的程序，不用担心，我们会在后面章节中做详细介绍。

2.1 使用 Turtles 绘画

是时候开始编程了！我们强烈建议你将代码一行行输入 IDLE3，这样可以帮助你理解每行都做了什么事情。言归正传，打开 IDLE3，点击 File → New Window，然后输入：

```
import turtle
window = turtle.Screen()
babbage = turtle.Turtle()
babbage.left(90)
babbage.forward(100)
babbage.right(90)
babbage.circle(10)
window.exitonclick()
```

然后点击 Run → Run Module 或者按 F5 键来运行程序。此时会跳出一个对话框让你输入文件名。起什么名字都可以，但起一个容易理解的名字可以帮助我们记住它（这里我们使用 chapter2-example1.py）。

文件中每一行都是一个 Python 指令。Python 一行行检查它们，然后按照先后顺序来执行。图 2-1 显示了计算机执行完所有步骤后的结果：在屏幕上画出一条直线，直线上面连接一个圈。看起来像棒棒糖，实际上，这是一朵花的第一部分。如果你的屏幕没有显示出这个结果，请检查下文件的每一行是否正确，然后再试一次。

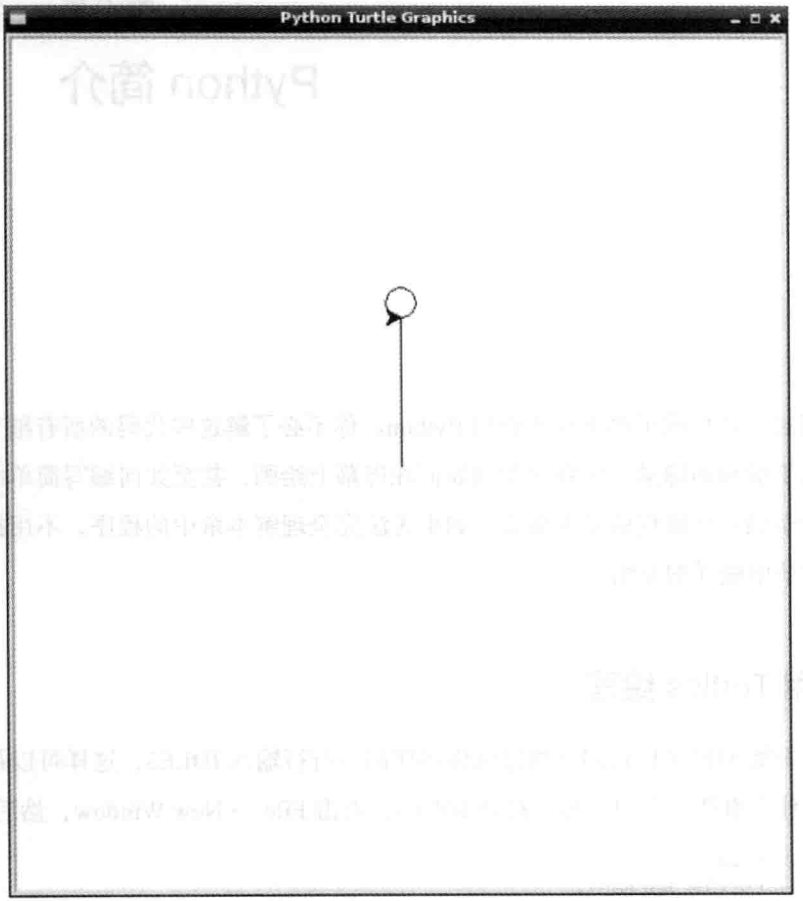


图 2-1　使用 Python turtle 绘出的第一张图

接下来我们仔细看下 Python 是如何根据代码来工作的。

```
import turtle
```

在 Python 程序的开头部分，通常会有一些 import 行。它们为程序导入一些附加特性，就像其他软件中的附件或者插件一样。这些特性被分成不同的模块。后面章节中将介绍更多的 import 命令。本例中我们仅导入 turtle 模块以完成绘图工作。

接下来的代码：

```
window = turtle.Screen()
...
window.exitonclick()
```

创建了一个新窗口用于绘图，并定义了一个动作，单击窗口就可以将其关闭。

下面这行代码使用了我们在第一行中导入的 turtle 模块创建的一个名字叫 babbage 的 turtle 对象（Charles Babbage，计算机先驱，创造了计算机的概念）。

```
babbage = turtle.Turtle()
```

Babbage 有很多个方法可供我们使用。例如下面这行：

```
babbage.left(90)
```

使用了 left() 方法，使 babbage 向左转一定角度。位于 left() 方法后面括号里的参数可以控制该方法运行时的角度。本例中，输入参数为 90，因此 babbage 向左转 90° 。下一行使用了 forward()、right() 和 circle() 方法。

```
babbage.forward(100)
babbage.right(90)
babbage.circle(10)
```

第一个方法将 turtle 向前移动 100 个像素，第二个将其右转 90° ，最后一个画了一个半径为 10 像素的圆。

现在开始添加花瓣。按下面例子编辑代码（变化部分已经用黑体标出）：

```
import turtle
#create window and turtle
window = turtle.Screen()
babbage = turtle.Turtle()
#draw stem and centre
babbage.left(90)
babbage.forward(100)
babbage.right(90)
babbage.circle(10)
#draw first petal
babbage.left(15)
babbage.forward(50)
babbage.left(157)
babbage.forward(50)
#tidy up window
window.exitonclick()
```

运行后会看见这朵花已经有了第一片花瓣。注意我们添加了一些带 # 符号的行，计算机将忽略以 # 开始的行。因此我们可以用这种方法添加注释（或者其他任何阅读代码的

人)。它增加了程序的可读性。当我们几天、几周甚至几年后再来看这段代码，有了注释就可以很容易地理解它做了什么。

2.1.1 使用循环

画花瓣的部分很容易理解（我们使用三角函数来计算出两个左转之间的夹角，但不用担心，我们不会陷入数学计算中）。

现在可以添加一段新代码来画第二片花瓣（总共要画 24 片花瓣）。和画第一片花瓣完全相同，因此只要复制粘贴就可以得到下列代码：

```
#draw second petal
babbage.left(15)
babbage.forward(50)
babbage.left(157)
babbage.forward(50)
```

然后，重复 22 次画出剩余花瓣。好吧，够了，根据编程的一般经验法则，不能重复出现相同的代码。假设你要改变花瓣尺寸，而且需要改 48 个地方（每片花瓣两次），如果忘记任何一个地方，将会得到一个不靠谱的图片。因此，你可以使用循环，用一小段代码告诉计算机反复执行特定部分的代码。

你可以用下面代码来替代从 # 画第一片花瓣（#draw first petal）开始的所有代码：

```
#draw all petals
for i in range(1,24):
    babbage.left(15)
    babbage.forward(50)
    babbage.left(157)
    babbage.forward(50)
window.exitonclick()
```

我们将会在第 3 章中介绍注释后面的第一行代码。现在，只需要理解它表示重复运行后面那段代码 24 次之后，计算机将执行这块代码后的下一个命令。

Python 中循环（或者之后接触的其他类型条件）的代码段通常使用相同的格式。第一行以冒号结尾，之后的每一行都使用相同的缩进。当 tab 或者缩进结束时，Python 就认为该代码段结束了。如果你之前使用过其他编程语言，将会发现 Python 的做法和其他语言有所不同。

运行代码后，你会发现 babbage 转了一圈将所有花瓣都画完了。我们得到了图 2-2 所示的完整花朵。

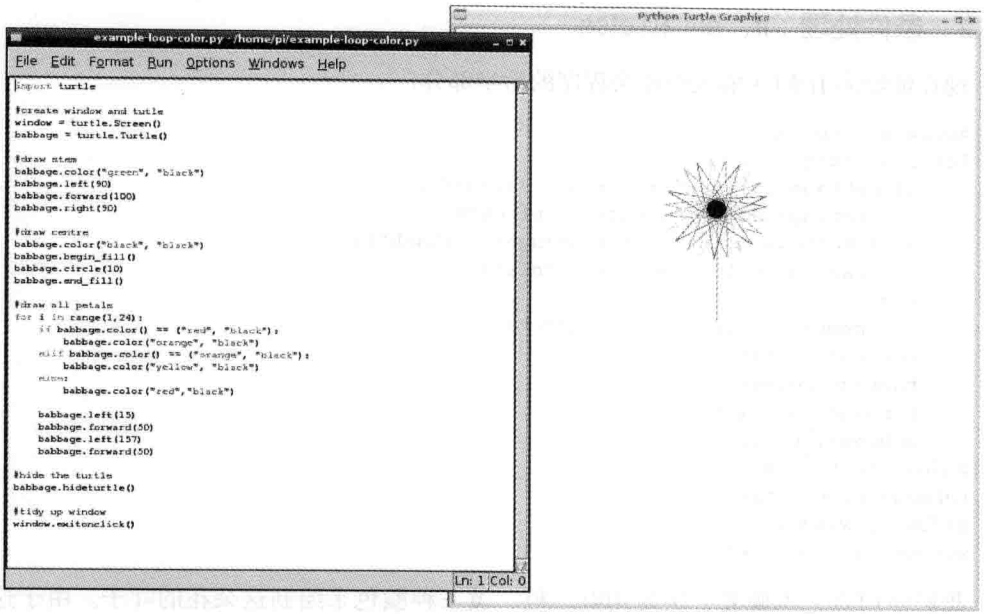


图 2-2　使用循环轻易地画出一朵花

仅用 13 行代码，干得不错。当然，不是所有的花都是黑色的，在图片中添加点颜色会更好看点。turtle 模块提供了一些方法让我们可以选择画线的颜色。下面对前面的代码做了些修改（更改部分以黑体显示）。

```
import turtle
#create window and turtle
window = turtle.Screen()
babbage = turtle.Turtle()
#draw stem
babbage.color("green", "black")
babbage.left(90)
babbage.forward(100)
babbage.right(90)
#draw centre
babbage.color("black", "black")
babbage.begin_fill()
babbage.circle(10)
babbage.end_fill()
```

上述代码中，我们使用了 color(colour1, colour2) 方法（英国人应该看得出来该方法的名字是美式写法）。这里 colour1 是画笔色，colour2 是填充色。画完花朵中心后，我们告诉计算机用 begin_fill() 方法填充中心。然后，我们调用 end_fill() 方法，以防它填充所有的花瓣。

2.1.2 条件处理：if、elif 和 else

现在继续向 IDLE3 输入画花朵程序的后半部分：

```
#draw all petals
for i in range(1,24):
    if babbage.color() == ("red", "black"):
        babbage.color("orange", "black")
    elif babbage.color() == ("orange", "black"):
        babbage.color("yellow", "black")
    else:
        babbage.color("red","black"))
    babbage.left(15)
    babbage.forward(50)
    babbage.left(157)
    babbage.forward(50)
#hide the turtle
babbage.hideturtle()
#tidy up window
window.exitonclick()
```

我们使用点艺术眼光，决定用红、橙、黄三种颜色来绘制这朵花的叶子。由于这是本黑白印刷的书，只有在树莓派上运行程序之后，才能看到彩色图像。当然也可以通过本书网站上的 flower.png 来查看结果。为了转换花瓣色彩，我们使用了 if ... elif ... else 语句。它告诉 Python 根据不同的数据做不同的事情。基本结构如下所示：

```
if <condition> :
    code
```

这里的 <condition> 表示条件，可以为真或假。本例中，我们使用如下条件：

```
babbage.color() == ("red", "black")
```

babbage.color()（注意该方法不带任何参数）告诉应用程序我们当前使用的颜色。不同于我们之前遇到的方法，它反过来给我们输出了一些信息。它的返回值是一对颜色——第一个是画笔色，第二个是填充色（从画完花朵中心之后，我们就没有改变过这两个颜色，因此程序的后继部分仍然使用它们）。双等于号（==）表示“相等”。使用双等于号是因为等号已经被定义为“赋值”，我们在创建 window 和 turtle 对象时已经使用过等号。

如果条件为真（本例中，如果 turtle 的颜色为（“red”，“black”）），Python 将执行 if 之后的代码。如果条件为假，Python 将转而执行 elif（elif 是 else if 的简写）。这里的判断条件和 if 一样。

如果 elif 处的条件为假，Python 将转到 else 处执行。到此为止（也就是说，if 和 elif

处的判断条件都为假），Python 将执行 else 之后的代码。else 处没有判断条件。图 2-3 描述了上述逻辑流程。

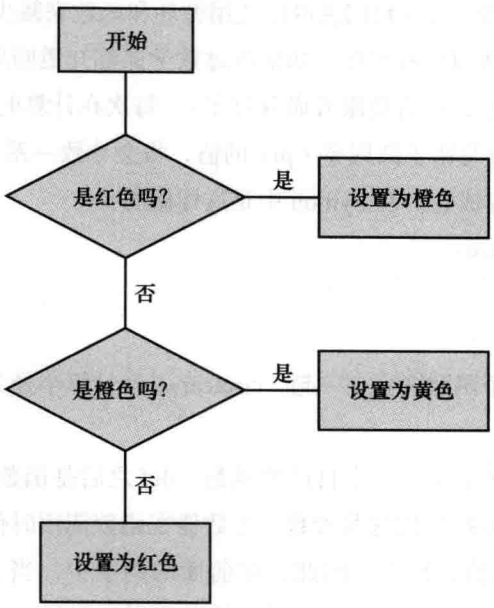


图 2-3　判断花瓣颜色的流程图

这些 if 语句能够在画完一个花瓣后改变画笔颜色。我们在程序后面新加了下面这行：

```
babbage.hideturtle()
```

它会隐藏 turtle（鼠标），以保证鼠标不会遮挡我们的图片。至此，我们的第一个 Python 程序圆满完成。

2.1.3　使用函数和方法组织代码

在开始第二个 Python 程序之前，我们先停下来讨论下“方法”。从前面的代码可以看出来，方法能够有效地控制程序。在前面的例子中，我们使用它们来移动 turtle，改变颜色或者创建窗体。每次都调用这些方法来完成某些事情。例如，通过 babbage.forward(50) 来调用 babbage 的 forward(50) 方法，通过 window.exitonclick() 来调用 window 的 exitonclick() 方法。每次调用这些方法，都会运行保存在 Python 模块中的相应代码。Python 还有个相似的特性——函数。函数和方法的工作方式有点类似，但是函数不需要 import 任何模块。例如，在 Python 解释器中输入：

```
>>> print("Hello World")
```

这行代码将运行 print 函数，将参数输出到屏幕上。是否还记得我们前面提到的不要在程序中出现重复的代码？当时我们说可以使用循环和函数来减少重复。例如，现在需要一个根据给定半径计算圆周长的程序。如果听过数学课都知道圆周长等于 2 × pi × r（如果没有上过数学课也没关系，只需要跟着做就好了）。每次在计算时重写代码可能会使问题变得复杂（如果某个地方写错了圆周率（pi）的值，将会导致一系列很难发现的问题）。因此我们创建一个函数来完成它。在 Python 中是这样做的：

```
def circlearea(radius):
    return radius * 3.14 * 2

print(circlearea(1))
```

这里我们使用了两个函数嵌套在一起。circlearea(1) 计算半径为 1 的圆周长，print() 把计算结果输出到屏幕上。

我们使用关键字 def 定义了一个自己的函数。def 之后是函数名，接下来的小括号里是参数。在程序中可以直接使用这些参数，参数值在函数调用时传递过去。关键字 return 告诉 Python 我们的返回值是什么。因此，在前面的例子中，当 Python 发现 circlearea(1) 时，将会把 1 作为参数 radius 的值传入函数，然后执行 def circlearea(radius)。接着，它返回计算结果 (6.28) 给函数 print。以后你将会看到，方法也可以像函数一样，一个方法通过嵌套将信息传递给另外一个方法。这是一种非常有用的方法，可以在程序的不同部分间传递数据流。

2.2 一个 Python 游戏：猫和老鼠

现在，让我们开始第二个 Python 程序。这次你将写出一个猫和老鼠的游戏。游戏者使用方向键来控制老鼠，使其保持在猫的前方（由计算机控制猫）。保持时间越长，得分越高。



本书中的大段程序大多数都可以从网站 www.wiley.com/go/python-raspberrypi.com 下载。为避免潜在的输入错误，你可以下载这些程序，复制粘贴到 IDE 或者代码编辑器中。下面例子中的程序名字是 Chapter2-catandmouse.py。

打开一个 IDLE3 新窗口并输入下面的代码：

```
import turtle
import time

boxsize = 200
caught = False
score = 0

#functions that are called on keypresses
def up():
   mouse.forward(10)
   checkbound()

def left():
   mouse.left(45)

def right():
   mouse.right(45)

def back():
   mouse.backward(10)
   checkbound()

def quitTurtles():
   window.bye()

#stop the mouse from leaving the square set by box size
def checkbound():
   global boxsize
   if mouse.xcor() > boxsize:
       mouse.goto(boxsize, mouse.ycor())
   if mouse.xcor() < -boxsize:
       mouse.goto(-boxsize, mouse.ycor())
   if mouse.ycor() > boxsize:
       mouse.goto(mouse.xcor(), boxsize)
   if mouse.ycor() < -boxsize:
       mouse.goto(mouse.xcor(), -boxsize)

#set up screen
window = turtle.Screen()
mouse = turtle.Turtle()
cat = turtle.Turtle()
mouse.penup()
mouse.penup()
mouse.goto(100,100)

#add key listeners
window.onkeypress(up, "Up")
window.onkeypress(left, "Left")
window.onkeypress(right, "Right")
window.onkeypress(back, "Down")
window.onkeypress(quitTurtles, "Escape")
```

```
difficulty = window.numinput("Difficulty",
     "Enter a difficulty from easy (1), for hard (5) ",
     minval=1, maxval=5)

window.listen()
#main loop
#note how it changes with difficulty
while not caught:
   cat.setheading(cat.towards(mouse))
   cat.forward(8+difficulty)
   score = score + 1
   if cat.distance(mouse) < 5:
       caught = True
   time.sleep(0.2-(0.01*difficulty))
window.textinput("GAME OVER", "Well done. You scored:
      "+ str(score*difficulty))
window.bye()
```

代码很多，在仔细读之前可以先试玩几次感受一下。这样也可以检验你的输入是否完全正确。如果程序出错，先检查下输入，然后再试一下。图 2-4 给出了程序运行界面。

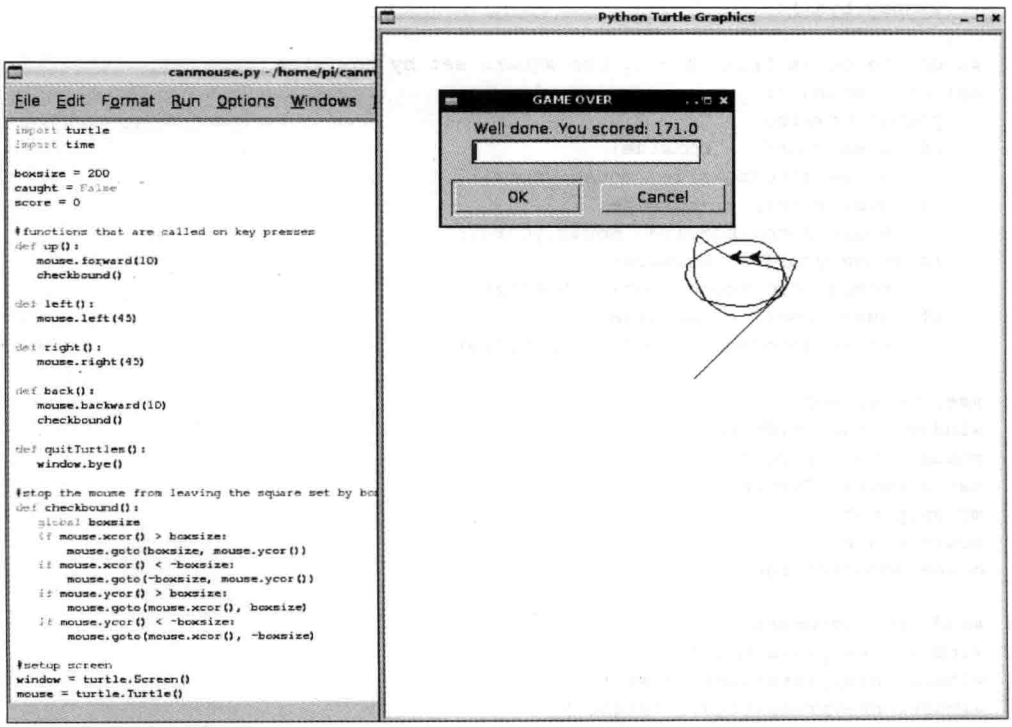


图 2-4 一个简单的猫和老鼠游戏

2.2.1　理解变量

程序前两行导入 turtle 和 time 模块，接下来的三行：

```
boxsize = 200
caught = False
score = 0
```

使用了我们之前接触过但没有深入介绍的概念：变量（variable）。我们可以在变量中存入值以便后面使用。例如，第一行中在变量 boxsize 中存入 200。写完这行，你就可以在程序中使用 boxsize，Python 将使用正确的值来替代它。这个结构之所以称为变量是因为它可以变化。在我们的这个程序中，boxsize 保持不变，但 caught 和 score 的值会发生变化。每次需要一个新值时，可以使用单个等号对其重新赋值。这和我们在第一个程序中对 window 和 babbage 赋值的做法一样，它们分别保存了 screen 和 turtle。在第 3 章中我们将介绍变量，以及可以在变量中存储什么值。

2.2.2　定义函数

程序中接下来的代码定义了一些函数。注意在函数 checkbounds() 中的这一行：

```
global boxsize
```

这行是必需的，因为函数无法存取定义在函数外面的变量。这一行告诉 Python 我们将在本函数中使用变量 boxsize，而该变量是在函数外部定义的。

可能最让人困惑的就是这一节：

```
#add key listeners
window.onkeypress(up, "Up")
window.onkeypress(left, "Left")
window.onkeypress(right, "Right")
window.onkeypress(back, "Down")
window.onkeypress(quitTurtles, "Escape")
```

这段代码告诉窗体当不同的键按下时应该做什么。例如，第一行表示当“向上”键（键盘方向键的向上按键）按下时，运行函数 up（我们已经定义好了）。

2.2.3　在游戏中循环

接下来就到了游戏的主循环中：

```
while not caught:
    cat.setheading(cat.towards(mouse))
    cat.forward(8+difficulty)
    score = score + 1
```

```
    if cat.distance(mouse) < 5:
        caught = True
    time.sleep(0.2-(0.01*difficulty))
```

代码中使用了不同的循环方法。while 循环如下：

```
while condition:
    loop code
```

它表示如果条件（condition）为真，将执行后面的循环代码（loop code）。在程序的变量列表中，可以看到变量 caught 被赋为 False：

```
caught = False
```

因此，这里使用的条件为 not caught（因为 not False 为真，程序开始时该条件为真），程序一直运行到 caught 被赋值为真，因为 not True 表示假。这样描述会有点复杂。但如果按字面意思就很容易理解：前面的 not 表示将真和假反一下。

time.sleep() 告诉 Python 停止在给定的时间，单位为秒。本例中我们随着难度等级（由用户输入一个值存在变量中）增加而减少程序等待时间。你会发现，随着难度增加，猫的运动速度也在增加。



此时也许你很想知道如何记住各个模块中对应的所有方法。比如，怎么才能知道应该使用 forward(10) 而不是 forwards(10) 或者 move_forwards(10)，或者怎么才能知道哪里可以找到向前移动的方法呢？使用 Python 不需要有超人的记忆力，你只需要知道在哪里可以查看帮助就好了。如果使用 IDLE3，按下 F1 Python 文档就会显示在 Web 浏览器里。这里有很多有用信息，非常值得浏览一下。例如需要查看 turtle 模块的信息，只需要在快速查找窗口中输入 turtle，然后选择第一项，你将会看到，我们已经使用过它的方法了。

2.3 小结

我们的 Python 快速导览到此为止。希望这些程序能够帮助你理解它。不用担心无法百分之百地理解全部内容，我们会在第 3 章比较详细地介绍 Python 的其他部分。无论如何，还是希望你能理解下列几个方面：

- Python 程序由一系列命令组成并从上到下执行。
- 可以通过循环和 if 语句控制程序的执行顺序。
- 不必事必躬亲，通过导入模块，使用模块中的方法可以完成许多工作。

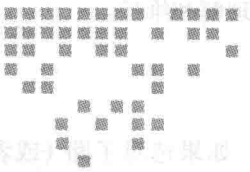
- 函数可以帮助重用代码，也可以使程序变得易于理解和维护。
- 变量可以存储信息以便后面使用。

在 Python 中画一朵花或者写个游戏非常容易。

请记住，编程时，完成一件事情可以有多种方法。如果选对了路（或者至少不走错路），工作会变得更轻松。然而，有时并不能轻易判断出对错。因此，我们以 Python 对此的建议来结束本章。在 Python 解释器中，输入：

```
>>> import this
```

this 是一个特殊的模块。导入之后可以输出很多有用的 Python 建议。现在，你对 Python 已经有一些了解，让我们更进一步去探索 Python 代码吧。



Chapter 3 第 3 章

Python 基础

在前面章节中，我们直接深入 Python 程序中，希望能够给你一个概念——什么是 Python。但你还不清楚它具体是如何工作的。本章中，我们将回答这些问题，并详细介绍如何在 Python 中创建自己的程序。在后继的章节中，我们将分别介绍 Python 的不同特性，帮助你为树莓派编写不同类型的程序。

3.1 变量、值和类型

第 2 章中，我们看到变量可以将数据存储下来供我们在别的地方使用。它们是程序员手里的一个强大工具。现在让我们来看看它们究竟是什么。如果你之前有过其他语言的编程经验，在这里会发现 Python 和其他语言有点不同。

在 Python 解释器中输入语句：

```
>>> score = 0
```

它告诉 Python 你想使用一个名字为 score，值为 0 的变量。在此之后，Python 只要看到 score，就会用值 0 来替换 score。为了验证这一点，继续输入：

```
>>> print(score)
```

请记住，Python 是顺序执行我们的命令的，在使用 score 之前必须先给它赋值。否则，Python 将会报错。

如果想改变 score 的值，只需要给它赋一个新值，如：

```
>>> score = 1
```

现在 Python 再遇到 score 时就会用 1 来替换它（你可以再次执行 print(score) 来验证一下）。你也可以在更新它的值时使用它：

```
>> score = score + 1
```

变量几乎可以使用任何名字但必须以字母或下划线开始，并且不能使用 Python 关键字（如 if、for 等）。Python 的命名习惯是使用小写字母，用下划线将单词分开，如：

```
high_score = 1000
```

在前面的例子中，所有的值都是数字，然而，值不只可以是数字，也可以文字，如：

```
player_name = "Ben"
```

我们甚至可以把同一个变量轮换赋值成数字和文字，例如：

```
>>> our_variable = 1000
>>> print(our_variable)
>>> our_variable = "Some Text"
>>> print(our_variable)
```

然而，变量的当前值只能是一种类型。

3.1.1　值和类型

看到数字 3 时，你只是看到一个 3，而不关心它究竟是一个文字，还是数字。3 就是 3。Python 却不一样。每个数据都有特定的类型，这样 Python 才知道该如何处理它们。通过函数 type() 可以看到 Python 数据的类型。在 Python 解释器中输入：

```
>>> type(3)
<class 'int'>

>>> type("3")
<class 'str'>
```

Python 告诉我们，第一个是 int（整数 integer 的简写），第二个是 str（字符 string 的简写）。这是因为 Python 认为整数 3 和字符 3 是不同的。执行下面这两行代码可以明显看出它们之间的区别：

```
>>> 3+3
6

>>> "3" + "3"
33
```

第一行将两个数字加一起，而第二行却是将两个字符合并在一起。由此可见，区分值的类型非常重要，如果出错，将会得到非常有意思的结果。为了探索更多的类型，可以输入：

```
>>>type(3.0)
<class 'float'>

>>>type(3>2)
<class 'bool'>
```

第一行输出 float(一个浮点数表示一个实数，小数点位置不固定)。第二行输出 bool(布尔类型，只有两个值：True 和 False)。

3.1.2 数字排序

数据的具体类型决定了 Python 可以执行哪些操作。这里我们从数值开始（包括 int 和 float 类型，但不包括 string，虽然它也包含数字)。对于数值，可以有两种操作类型：比较和数值操作。比较，需要两个操作数，返回值为 bool 型。如表 3-1 所示。

表 3-1 数值类型的比较操作

操作符	含义	例子
<	小于	3<2→False
>	大于	3>2→True
==	等于	3=3→True
<=	小于等于	3<=3→True
>=	大于等于	3>=4→False
!=	不等于	3!=4→True

数值操作返回一个数值类型，如表 3-2 所示。

表 3-2 数值操作

操作符	含义	例子
+	加	2 + 2→4
-	减	3-2→1
*	乘	2*3→6
/	除	10/2→5
%	求余	5%2→1
**	乘方	4 ** 2→16
int()	转换为 int 型	int(3.2)→3
float()	转换为 float 型	float(2)→2.0

你可以在 Python 解释器中输入任何一个操作符来验证一下。例如：

```
>>> 3!=3
False
```

在程序中使用数值运算，通常都将其返回值赋值给某个变量。例如：

```
>>> number_1 = 10
>>> number_2 = number_1**2
```

3.1.3　使用 Strings 保存文字

string 类型可以用来保存任何文字。创建字符串只需要将数据用单引号或者双引号括起来就可以了。在 Python 中，不论哪种引号都可以。我们首选双引号，因为它可以处理带’号（单引号或撇号）的字符串。但这绝不是普遍适用的。有些程序员喜欢使用单引号，因为其输入起来更方便。

这个数据类型不同于其他类型，因为许多时候，string 不只是单个数据而是一组字母。它的名字也反映了这个特点——字符“串”，一串字符。

和数值类型一样，Python 也为我们提供了一些操作方法。表 3-3 给出了一些常用的操作。

表 3-3　字符串操作

操作符	含义	例子
string[x]	获取第 x 个字符（从 0 开始数）	"abcde"[1]→"b"
string[x:y]	获取所有从 x 到 y 的字符	"abcde"[1:3]→"bc"
string[:y]string[:y]	获取从字符串开始到第 y 个的字符	"abcde"[:3]→"abc"
string[x:]	获取从第 x 个开始到字符串结束的字符	"abcde"[3:]→"de"
len(string)	返回字符串长度	len("abcde")→5
string+string	合并两个字符串	"abc"+"def"→"abcdef"

3.1.4　布尔值：真或假

最后我们来看 bool 类型。它非常简单，只有两种可能取值：True 和 False。注意在 Python 中，这两个值的首字母要大写，并且不需要任何引号。同时，这个值通常不存在变量中（虽然我们在第 2 章中见到过存在变量中），它通常用于条件语句如 if 的判断条件中，我们将在本章后面部分详细介绍。其主要操作符是与（and）、或（or）和非（not）。

非，就是简单地转换下取值：

```
>>> not True
False
```

```
>>> not False
True
```

与，需要两个操作数，如果两个数都为真，则返回真，否则，返回假：

```
>>> True and False
False

>>> False and False
False

>>> True and True
True
```

或，也需要两个操作数，如果两个数中任何一个为真，则返回真：

```
>>> True or False
True

>>> True or True
True

>>> False or False
False
```

3.1.5 数据类型转换

使用函数 int()、float() 和 str() 可以转换数据类型。它们分别将其他数据类型转换为整数、浮点数和字符串。然而它们却不能随意转换。如果将浮点数转为整数，Python 将舍去所有小数部分。当字符串中只有一个字符时，才能转换成数字。但是，其他类型几乎都可以转换成字符串。下面给出了一些例子。

```
>>> print(int(3.9))
3

>>> print(str(True))
True

>>> print(float("Three point two"))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: 'Three point two'
```

3.1.6 知识测试

下面是一些 Python 语句。看一下你是否能弄懂它们的意思。想出来什么意思之后，将它们输入 Python 解释器检验自己的判断。提示：有一些语句可能会导致错误。

- `int("two")`
- `print(str(3+3) + "3")`
- `type(3=3)`
- `"4" == 4`
- `"Python"[4]`
- `(3 > 2) or (2 > 3)`
- `not"True"`
- `2345[2]`
- `str((not True) and (not False))`
- `10 % 3`

练习1

下面程序中哪些是变量，它们的值是什么，类型是什么？答案在本章结尾。如果不确定，可以在代码中的某些地方加上print语句，并输入Python解释器中看看结果。当程序的运行结果和我们期望的不一样时，这个方法很有用，可以用来确定某些代码究竟做了什么事。

```
prompt_text = "Enter a number: "
user_in = input(prompt_text)
user_num = int(user_in)

for i in range(1,10):
    print(i, " times ", user_num, " is ", i*user_num)

even = (user_num % 2) == 0

if even:
    print(user_num, " is even")
else:
    print(user_num, " is odd")
```

3.2 在结构体中存储值

除了简单数据类型，Python还允许我们将数据用不同方式组合起来创建结构体。最简单的结构体是sequences（线性结构）。它将信息一个接一个地存储起来。它分为有两类：lists（列表）和tuples（元组）。多数情况下，它们是相似的。

来看接下来的例子：

```
>>> list_1 = [1,2,3,4]
>>> tuple_1 = (1,2,3,4)
>>> list_1[1]
2
>>> tuple_1[1]
2
```

我们用方括号将数字括起来构成列表，用圆括号将数字括起来构成元组。到现在为止，它们两个工作起来都是一致的。在结构体名后面跟方括号，方括号中填下标就可以访问单个元素。注意下标从 0 开始，因此 list_1[0] 和 tuple_1[0] 可以访问线性结构中的第一个元素。

当你去更新元素时就会发现列表和元组间的差别：

```
>>> list_1[1]=99

>>> list_1
[1, 99, 3, 4]

>>> tuple_1[1]=99
Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    t1[1]=99
TypeError: 'tuple' object does not support item assignment
```

可见，你可以更新列表中的单个元素，却不能更新元组中的单个元素。然而，你可以一次性覆盖元组中的所有元素。这时，可以告诉 Python 将变量 tuple_1 赋一个新值以取代旧值。

```
>>> tuple_1
(1, 2, 3, 4)

>>> tuple_1=(1,99,2,4)

>>> tuple_1
(1, 99, 2, 4)
```

上一节中我们提到的 strings 就是一串字符，它的操作符可以用于列表和元组。下面继续以 list_1 和 tuple_1 为例：

```
>>> len(list_1)
4

>>> tuple_1[:3]
[1, 99, 2]
```

参考表 3-3，复习下我们可以在这里使用的字符串操作。

列表和元组中的元素可以是任意数据类型，包括列表和元组自身。如果愿意，你可以创建列表的列表的列表的列表。当然，如果真的这样做了，你的代码将会变得异常难懂。

但列表的列表，通常会比较有用。你可以把它当做一个二维表：

```
>>> list_2 =
  [["a","b","c"],["d","e","f"],["g","h","i"],["j","k","l"]]
```

你可以通过主列表和子列表索引来获取元素：

```
>>> list_2[2][0]
'g'

>>> list_2[0]
['a', 'b', 'c']

>>> list_2[0][1:]
['b', 'c']
```

从表 3-4 中可以看出来为什么它可以当做二维表。

表 3-4　二维列表示例

`list_2[0][0]` = "a"	`list_2[1][0]` = "d"	`list_2[2][0]` = "g"	`list_2[3][0]` = "j"
`list_2[0][1]` = "b"	`list_2[1][1]` = "e"	`list_2[2][1]` = "h"	`list_2[3][1]` = "k"
`list_2[0][2]` = "c"	`list_2[1][2]` = "f"	`list_2[2][2]` = "i"	`list_2[3][2]` = "l"

有许多方法可以用来操纵列表。表 3-5 给出了常用的方法。

表 3-5　列表操作方法（以 list_3 = [3, 2, 1] 为例）

操作符	含义	例子
`list.append(item)`	添加元素到列表尾部	`list_3.append(0)` → `[3,2,1,0]`
`list.extend(list_2)`	合并 list_2 到列表尾部	`list_3.extend([0,-1])` → `[3,2,1,0,-1]`
`list.pop(x)`	返回并删除第 x 个元素	See below
`list.insert(x, item)`	插入元素到第 x 个位置	`list_3.insert(99,1)` → `[3, 99, 2, 1]`
`list.sort()`	排序列表	`list_3.sort()` → `[1,2,3]`
`list.index(item)`	返回列表中第一次出现该元素的位置	`List_3.index(2)` → `1`
`list.count(item)`	计算列表中该元素出现的次数	`list_3.count(2)` → `1`
`list.remove(item)`	删除列表中第一次出现的该元素	`list_3.remove(2)` → `[3,1]`

这里的多数例子都和我们之前遇见的不一样，因为它们（除了 index() 和 sort()）改变了 list_3 的值而不是返回一些值。例如，在 Python 解释器中运行第一个例子，当然，你需要额外的一行来显示 list_3 的值。

```
>>> list_3 = [3,2,1]
>>> list_3.append(0)
>>> list_3
[3, 2, 1, 0]
```

然而，index() 和 count() 仅仅返回一个值：

```
>>> list_3.index(2)
1
```

pop(x) 有点特殊，因为它一次做了两件事。首先，它返回列表中第 x 个位置的元素值，同时它还从列表中删除了该元素。试着运行下面的例子，感受下它是如何工作的：

```
>>> list_3 = [1,2,3]
>>> out = list_3.pop(1)
>>> out
2
>>> list_3
[1, 3]
```

上面说过，元组除了不能被修改，它和列表非常类似。所有用于列表操作，只要不改变元素值，都可以用于元组：

```
>>> tuple_2 = (1,2,3)
>>> tuple_2.index(2)
1
>>> tuple_2.sort()
Traceback (most recent call last):
  File "<pyshell#91>", line 1, in <module>
    tuple_2.sort()
AttributeError: 'tuple' object has no attribute 'sort'
```

3.2.1 字典和集合中的非序列元素

你可以认为列表和元组是元素的集合，每个元素都对应了其中的一个下标。例如，在列表 [“a”,“b”,“c”,“d”] 中，a 的下标是 0，b 的下标是 1，以此类推。然而，当你想用非数字的下标时该怎么做呢？例如，你想要创建一个数据结构，把朋友的昵称和真实名字关联起来，如：

```
>>> real_name["Benny"]
'Benjamin Everard'
```

在 Python 中，可以使用通过花括号来定义的字典（dictionary）。你可以使用下列语句创建字典 real_name：

```
>>> real_name = {"Benny" : "Benjamin Everard",
                 "Alex" : "Alexander Bradbury"}
```

字典中的元素称为键值对（key/value pair），其中第一部分（本例中的昵称）是键

(key)，第二部分（全名）是值（value）。只需要给定一个新key及其对应的值就可以在字典中新加元素：

```
>>> real_name["Liz"] = "Elizabeth Upton"
>> real_name
{'Alex': 'Alexander Bradbury', 'Benny': 'Benjamin Everard',
    'Liz': 'Elizabeth Upton'}
```

你可能想知道为什么需要下标或者键。实际上，这些并不是必需的。Python中的集合(set)允许你把一堆数据放在一起而不用指定下标或序号。例如：

```
>>> herbs = {'thyme', 'dill', 'corriander'}
>>> spices = {'cumin', 'chilli', 'corriander'}
>>> "thyme" in herbs
True
```

可以看出，Python使用in操作来测试给定的值是否在集合中。还有些其他操作可以用到集合中。参见表3-6。

表3-6 集合的操作（例子中使用了先前定义的集合）

操作符	含义	例子
set_1 & set_2	返回两个集合共有的元素	herbs & spices→'corriander'
set_1 | set_2	合并两个集合中的元素	herbs | spices→{'dill', 'thyme', 'chilli', 'corriander', 'cumin'}
set_1 - set_2	set_1中存在set_2中不存在的元素	herbs - spices→{'dill', 'thyme'}
set_1 ^ set_2	set_1或set_2中存在的元素，不包括两个集合共有的元素	herbs ^ spices→{'dill', 'cumin', 'thyme', 'chilli'}

3.2.2 知识测试

下面的Python语句是什么意思？试想一下，然后将它们输入Python解释器检验下自己的判断。提示：有一些语句会导致错误。

- ["a", "b", "c"].index("c")
- (3,2,1).pop(2)
- {1,3,5} & {2,4,6}
- {1,2,3} & {1}
- 3 in {1,2,3} ^ {3,4,5}
- "abcde".remove("c")
- 3 not in (1,2,3)

3.3 控制程序流程

while 循环是一种最简单的循环。只要结果是布尔类型的任何语句都可以做它的判断条件，它将会持续循环到条件为假。如果条件始终为真，它将一直循环下去。例如：

```
>>> while True:
    print("Ben is awesome")
```

但愿你还记得第 2 章的循环体。在条件后面要加上冒号，接下来的一行要有缩进，所有缩进部分都属于循环体。要在 Python 解释器中运行这段代码，输入 print 语句之后要按回车键（Enter）。用退格键（backspace）删掉自动产生的 tab（缩进），然后再按回车键。这是告诉 Python 循环体结束了，你需要执行这段代码。

这段代码会陷入死循环，按 Ctrl+C 组合键可以终止它。

无论判断条件选择的多么复杂都可以，只要它的返回值是真或假就可以了。但为了最终能够跳出循环，通常需要一个或多个变量以方便在循环内部改变判断条件。比如说，下面这个猜数字游戏中的主循环。

```
import random
secret = int(random.uniform(0,10))
print("I'm thinking of a number between zero and ten."
      , "Can you guess what it is?")
guess = 11

while guess != secret:
    guess = int(input("Take a guess: "))

print("Well done!")
```

注意，如果输入非数字的字符，这个程序将报错。后面我们会介绍如何简单地修复这个问题，但让我们先来看一下另一种循环。

3.3.1 用循环遍历数据

for 循环可以用来遍历数据，它在每次循环中对一个数据进行处理。通常我们在 rang(x, y) 的帮助下，遍历从 x 到 y 的每一个数据[⊖]。例如，你可以得出一个 12 倍表：

```
>>> for i in range(1,13):
    print(i, " times twelve is ", i*12)
```

rang() 还有一个参数，可以设定两个连续数字之间的间隔。例如我们把 range(1,13) 改成 range(2,14,2)，它将会只计算从 2 到 14 的所有偶数[⊜]。我们还可以使用其他线性数据类

⊖ 包含 x，不包含 y。——译者注

⊜ 不包括 14。——译者注

型（包括字符串）和集合来控制 for 循环。下列例子都是合法的：

```
>>> for i in [1,2,3,4,5,6,7,8,9,10,11,12]:   print(i, " times 12 is
  ", i * 12)
>>> for i in (1,2,3,4,5,6,7,8,9,10,11,12):
    print(i, " times 12 is ", i*12)
>>> for i in {1,2,3,4}:
    print(i, " times 12 is ", i*12)
>>> for i in "123456789":
    print(i, " times 12 is ", int(i)*12)
```

最后一行是一个很好的例子，它展示了使用错误的数据类型会有什么结果。集合和其他类型有点不一样，因为它没有顺序，只是一堆数据的集合。下面两个例子的执行结果是一样的。

```
>>> for i in {1,2,3,4}:
    print(i, " times 12 is ", i*12)

>>> for i in {4,3,2,1}:
    print(i, " times 12 is ", i*12)
```

字典有点特别，因为它保存的不是元素，而是键值对。for 循环中可以使用它，但必须做些特殊处理：

```
>>> real_name = {"Benny": "Benjamin Everard",
                 "Alex": "Alexander Bradbury"}
>>> for key,value in real_name.items():
    print("The real name of " + key + " is " + value)
```

使用集合，你就不能控制循环中访问数据的顺序。实际上，如果你期望用 for 循环顺序访问数据，则需要使用线性数据结构，如列表或元组，而不是使用像集合或字典这样的无序数据结构。

3.3.2 深入理解循环嵌套

编程时经常需要同时遍历多种数据，比如下面这个用来找出 1 到 30 之间所有素数的程序。

```
for i in range(1,30):
    is_prime = True
    for k in range(2,i):
        if (i%k) == 0:
            print(i, " is divisible by ", k)
            is_prime = False
    if is_prime:
        print(i, " is prime ")
```

请注意缩进级别，第一个循环体的缩进为一，第二个循环体的缩进为二。这一点非常重要，只有这样，Python 才能理解哪个代码属于哪个循环体，以及每个循环体在何处结束。

使用嵌套时要注意，它可能会使程序变慢。上面的例子运行得很快，但如果你试图计算 3000 以内的素数（只需要在第一行的 30 后面加两个零），程序运行就会花非常长的时间。不只是因为外层循环要多循环上千次，每次走进内层循环也需要多执行很多次。你会发现整个程序运行起来很慢（如果你在做这个实验，记得按 Ctrl+C 组合键可以停止它）。幸运的是，我们还可以改进它。试一下这个程序：

```
for i in range(1,3000,2):
    is_prime = True
    for k in range(2,i):
        if (i%k) == 0:
            print(i, " is divisible by ", k)
            is_prime = False
            break
    if is_prime:
        print(i, " is prime ")
```

首先，使用 range(1, 3000, 2) 跳过所有的偶数。我们直接省去一半时间。其次，你可能注意到了，在 if 里面我们多加了一个 break。这会让我们节约更多时间。我们是在计算素数，因此不用关心每个数的所有约数。一旦发现这个数字是非素数[⊖]，就可以直接使用 break 跳出循环，继续执行下面一行（if is_prime: ）。这两点优化可以使程序运行得更快。

3.3.3 使用 if 语句控制程序分支

不仅可以使用循环来不断执行某段代码，还可以使用分支来控制 Python 程序流，使其根据不同条件，执行不同的代码。分支由 if 语句实现。我们已经见过很多次 if 语句，现在来复习一下。if 语句，像 while 循环一样，只需要一个布尔类型的条件。它后面还可以有附加语句如 elif（else-if）和 else 语句。例如：

```
num = int(input("enter a number: "))
if num%2 == 0:
    print("Your number is divisible by 2")
elif num%3 == 0:
    print("your number is divisible by 3")
elif num%5 == 0:
    print("your number is divisible by 5")
else:
    print("your number isn't divisible by 2,3 or 5")
```

⊖ 除了 1 和它本身还有其他约数。——译者注

一个 if 语句最多只执行一段代码，只要 Python 发现条件为真，就执行该段代码并结束整个 if 语句。因此如果在上面的程序中输入 10，它将只返回该数字可以被 2 整除，而不会试下该数字能否被 5 整除（如果你期望它试一下，就需要写一个新的 if 语句而不是用 elif）。如果没有一个条件为真，则执行 else 后面的代码段。在其他例子中可以看到，if 语句可以不带 elif 或 else。如果没有 else 语句，同时判断条件也不成立，Python 就会跳过 if 语句，不执行其中的任何代码。

3.3.4 捕获异常

如果你喜欢尝试，在前面的例子中试图输入过非数字的字符，就会发现程序会报错。这是因为 Python 不能把任意字符转换成数字，它不知道该怎么做了。此时，Python 就会报出错误。目前为止，我们只是放任它这么做，然后，就会导致程序崩溃。

然而，如果你知道某段代码可能会报出异常，可以告诉 Python，这样程序就知道遇到问题时该如何处理。可以参考下面代码：

```
try:
    code where there might be a problem
except type_of_error:
    code to run if there's an error
```

你可以告诉 Python 要处理的错误类型，因为 Python 在异常时会输出错误类型。如：

```
>>> num = int(input("Enter a number: "))
Enter a number: dasd
Traceback (most recent call last):
  File "<pyshell#176>", line 1, in <module>
    num = int(input("Enter a number: "))
ValueError: invalid literal for int() with base 10: 'dasd'
```

可以看到，这是个 ValueError（值错误）。因此前一个例子可以改成：

```
is_number = False
num = 0

while not is_number:
    is_number = True
    try:
        num = int(input("enter a number: "))
    except ValueError:
        print("I said a number!")
        is_number = False

if num%2 == 0:
    print("Your number is divisible by 2")
elif num%3 == 0:
    print("your number is divisible by 3")
```

```
elif num%5 == 0:
    print("your number is divisible by 5")
else:
    print("your number isn't divisible by 2,3 or 5")
```

练习 2

试着使用 try 语句来捕获由于用户输入错误导致的异常，修复本章开始的猜数字游戏。

3.4 使用函数复用代码

我们已经使用过一些函数，如 print() 和 input()。这些都是 Python 内置函数。你还可以自己定义函数。第 2 章中你已经定义过自己的函数，这里我们来复习一下。看下这个：

```
>>> def square(num):
    return num**2
>>> square(4)
16
```

这段代码定义了一个名叫 square（平方）的函数，然后使用这个函数计算了 4 的平方。使用它的时候需要在名字后面跟一个包含在小括号里的数字——我们称为参数。函数运行时，参数名（本例中是 num）将会被赋值为你给定的数字。return 语句可以出现在程序中的某个地方，用来给主程序返回数据。如果有多个 return 语句，Python 将在第一次遇到 return 时返回。

你也可以创建包含多个参数的函数。例如，下面的程序就有一个函数需要两个参数并返回其中较大的一个。

```
def biggest(a,b):
    if a>b:
        return a
    else:
        return b

print("The biggest of 2 and 3 is ", biggest(2,3))
print("The biggest of 10 and 5 is ", biggest(10,5))
```

至此为止，它们都工作地很好。然而如果需要在函数中改变了变量的值呢？看看下面的程序：

```
def add_one(num):
    num = num + 1
    return num

number_1 = 1
number_2 = add_one(number_1)
```

```
print("number_1: ", number_1)
print("number_2: ", number_2)
```

运行这段程序之前，先想一下运行结果。可以肯定的是，number_2将会是2，但number_1呢？它应该是1么？因为我们在主程序中给它赋值为1。还是2呢？因为我们把它传递给函数add_one()并在里面改变了它的值。

运行后，你会发现number_1是1。这是因为这一行：

```
num = num + 1
```

你告诉Python不想让num继续保持原来的值（和number_1一样），而是给它一个加1后的新值。

然而，可变数据类型如列表、集合和对象就不一样了。如果它们中的一种数据类型传入函数并改变它，那么作为参数传入的原始值也会跟着变化。比如：

```
def add_item(list_1):
    list_1.append(1)
    return list_1

list_2 = [2,3,4]
list_3 = add_item(list_2)
print("list_2: ", list_2)
print("list_3: ", list_3)
```

运行之后将得到：

```
list_2:  [2, 3, 4, 1]
list_3:  [2, 3, 4, 1]
```

通常情况下，这样做没有什么问题，但有时你并不想让作为参数传入的原始值发生变化。这种情况下，你需要使用copy.deepcopy()，不过首先你得导入copy模块。如果你把上个例子改为：

```
import copy
def add_item(list_1):
    list_1.append(1)
    return list_1

list_2 = [2,3,4]
list_3 = add_item(copy.deepcopy(list_2))
print("list_2: ", list_2)
print("list_3: ", list_3)
```

你将得到：

```
list_2:  [2, 3, 4]
list_3:  [2, 3, 4, 1]
```

可选参数

有时你需要创建这样一个函数，它有时候需要参数，有时候不需要。比如，你可能需要创建函数 increment()，该函数接受两个参数并将它们相加。当只给一个参数时，该函数就把这个参数加上 1。这时候你就可以使用默认值（本例中为 1）作为参数[⊖]：

```
def increment(num1=1, num2):
  return num2 + num1
```

这里唯一需要注意的就是可以省略的参数必须放在其他参数之后。

3.5 组合装配

本章中我们已经介绍过很多东西，但还没有介绍如何把这些东西组装起来，创建一个你想要的程序。本节中，我们将介绍一个简单的学生成绩数据库程序，其中使用的大多数知识都是在前面讲过的。

该程序包含一些默认数据并允许你修改它。我们将使用合适的数据类型和数据结构来存储不同的信息，使用不同的函数来操纵这些数据。我们甚至还加了个简单的菜单以方便用户管理这些数据。



注意 本书中大多数较长代码的例子都可以从网站 www.wiley.com/go/python-raspberrypi 下载。为避免输入错误，你可以下载，复制并粘贴到 IDE 或代码编辑器中。

代码如下（你也可以从本书网站上找到它，名字是 chapter3-student-1.py）

```
students = [["Ben", {"Maths": 67, "English": 78, "Science": 72}],
            ["Mark", {"Maths": 56, "Art": 64, "History": 39,
             "Geography": 55}],
            ["Paul", {"English": 66, "History": 88}]]

grades = ((0, "FAIL"),(50, "D"),(60,"C"),(70, "B"), (80,"A"),
          (101, "CHEAT!"))

def print_report_card(report_student=None):
    for student in students:
        if (student[0] == report_student) or
           (report_student == None):
            print("Report card for student ", student[0])
            for subject, mark in student[1].items():
```

⊖ 原文如此，实际上 num1 = 1 应该放在后面，因为默认参数需要放在参数表后面。——译者注

```
                for grade in grades:
                    if mark < grade[0]:
                        print(subject, " : ", prev_grade)
                        break
                    prev_grade = grade[1]

def add_student(student_name):
    global students
    for student in students:
        if student[0] == student_name:
            return "Student already in database"
    students.append([student_name, {}])
    return "Student added successfully"

def add_mark(student_name, subject, mark):
    global students
    for student in students:
        if student[0] == student_name:
            if subject in student[1].keys():
                print(student_name, " already has a mark for ",
                      subject)
                user_input = input("Overwrite Y/N? ")
                if user_input == "Y" or "y":
                    student[1][subject] = mark
                    return "Student's mark updated"
                else:
                    return "Student's mark not updated"
            else:
                student[1][subject] = mark
                return "Student's mark added"
    return "Student not found"

while True:
    print("Welcome to the Raspberry Pi student database")
    print("What can I help you with?")
    print("Enter 1 to view all report cards")
    print("Enter 2 to view the report card for a student")
    print("Enter 3 to add a student")
    print("Enter 4 to add a mark to a student")
    print("Enter 5 to exit")

    try:
        user_choice = int(input("Choice: "))
    except ValueError:
        print("That's not a number I recognise")
        user_choice = 0
    if user_choice == 1:
        print_report_card()
    elif user_choice == 2:
        enter_student = input("Which student? ")
```

```
            print_report_card(enter_student)
        elif user_choice == 3:
            enter_student = input("Student name? ")
            print(add_student(enter_student))
        elif user_choice == 4:
            enter_student = input("Student name? ")
            enter_subject = input("Subject? ")
            num_error = True
            while num_error:
                num_error = False
                try:
                    enter_mark = int(input("Mark? "))
                except ValueError:
                    print("I don't recognise that as a number")
                    num_error = True
            print(add_mark(enter_student, enter_subject, enter_mark))
        elif user_choice == 5:
            break
        else:
            print("Unknown choice")
        input("Press enter to continue")
    print("Goodbye and thank you for using the Raspberry Pi"
          , "Student database")
```

目前，该程序的学生数据结构是字典的列表的列表[⊖]。使用字典的字典或者列表的列表的列表都可以完成相同的效果。试着分别使用这两种数据结构修改程序，感觉下哪种方式更自然一点。

练习 3

该程序中的数据可以用元组来存取么？为什么呢？答案详见本章末尾。

3.6 使用类来构建对象

类允许你将数据和函数组合起来构成一个对象。实际上，我们在前面章节中已经使用过它。回忆下这两行：

```
window = turtle.Screen()
babbage = turtle.Turtle()
```

当时我们只是快速跳过去，现在让我们来看看它是如何工作的。turtle.Turtle() 返回一个由 turtle 模块中 Turtle 类创建的对象。同样的，tuttle.Screen() 返回一个由 turtle 模块中

⊖ 各科成绩用字典保存，每个学生和自己的成绩组成一个列表，最后所有学生共同构成最外面的列表。——译者注

Screen类创建的对象。简而言之，类是用来构建对象的蓝图。对象可以存储数据，并且提供可以让你操作数据的方法。而方法其实就是类中的函数。

你已经见识过对象是多么有用了。在第2章的例子中，你不用关心turtle的数据是如何存放的，因为它们已经包含在对象中了。你只需要将turtle对象保存在一个叫babbage的对象里，当调用某个方法时，该方法就知道如何存取它需要的各种东西。这样可以使程序整洁易用。例如下面的例子：

```
babbage.forward(100)
```

这行代码将turtle向前移动并将结果画在屏幕上。在屏幕上画这条线，它知道用什么颜色的画笔，turtle的起始位置在哪里以及其他各种它所需要的信息，因为它们都已经存储在对象中了。

让我们用一个简单的例子看看对象里都有什么东西：

```
class Person():
    def __init__(self, age, name):
        self.age = age
        self.name = name

    def birthday(self):
        self.age = self.age + 1

ben = Person(31, "Ben")
paul = Person(42, "Paul")
print(ben.name, ben.age)
print(paul.name, paul.age)
```

这里有几点需要注意，Python中，变量、函数和方法的名字通常用小写字母，类是个例外。因此Person类由大写字母P开头。如果不这样做Python也不会报错，但遵守惯例可以方便人们阅读对方的代码。可以看到，方法的定义方式和函数一样，区别只是参数总是以self开始，这表示本地变量。在这个例子中，本地变量包括self.age和self.name。它们会在类的每一个实例中都创建一份。本例中，我们用People类创建了两个对象（即类的实例），每个对象都有各自的一份self.age和self.person拷贝。我们可以在对象外面读写它们（就像我们在print方法中使用的那样）。这被称为Person类的属性。

这里还有两个方法。__init__是每个类都有的特殊方法。该方法在创建或“初始化”类的实例时会被调用。因此，ben = Person(31,“Ben”)会创建Person类的一个对象，并使用参数(31,“Ben”)调用__int__方法。通常可以用来设置属性。第二个方法birthday()展示了如何使用类方法而不用在类的外面关心数据的保存问题。给Person对象一个birthday()方法，拿来用就可以了，例如：

```
ben.birthday()
```

它将把 age 加一。

有时，我们不希望从头开始创建类，而是根据已经存在的类来建一个新的类。例如，如果你想创建一个类来保存 parents（父母）的相关信息，它们也存在年龄（age），名字（name），和生日（birthday），如果再为 Parent 写一遍这些代码就会显得浪费。Python 允许我们从其他类中继承。下面给出一个例子：

```
class Person():
    def __init__(self, age, name):
        self.age = age
        self.name = name

    def birthday(self):
        self.age = self.age + 1

class Parent(Person):
    def __init__(self, age, name):
        Person.__init__(self,age,name)
        self.children = []

    def add_child(self, child):
        self.children.append(child)

    def print_children(self):
        print("The children of ", self.name, " are:")
        for child in self.children:
            print(child.name)

john = Parent(60, "John")
ben = Person(32, "Ben")
print(john.name, john.age)
john.add_child(ben)
john.print_children()
```

Person 是 Parent 的超类，Parent 是 Person 的子类。把类名放入要定义的类名后面的括号里，它就变成这个要定义的类的超类。你可以调用超类的 __init__ 方法，会自动获得超类的属性和方法的访问权限而不用重写代码。

类的最大优势就是它可以方便重用代码。像我们在前几章中看到的那样，它可以方便地操纵 turtle 而不用关心它做了些什么，是怎么做的。因为 turtle 类封装了这些信息，你只要知道方法名字，就可以毫无障碍地使用它们了。贯穿本书，你将看到如何使用类提供的方法方便地构建复杂游戏，而不用担心这些方法的技术细节是如何实现的。

下面的代码使用类重写了学生数据库程序，感受下类是如何工作的（文件名 chapter3-

student-2.py)

```
student_data= [["Ben", {"Maths": 67, "English": 78,
                        "Science": 72}],
              ["Mark", {"Maths": 56, "Art": 64, "History": 39,
                        "Geography": 55}],
              ["Paul", {"English": 66, "History": 88}]]

grades = ((0, "FAIL"),(50, "D"),(60,"C"),(70, "B"), (80,"A"),
          (101, "CHEAT!"))

class Student():
    def __init__(self, name, marks):
        self.name = name
        self.marks = marks

    def print_report_card(self):
        print("Report card for student ", self.name)
        for subject, mark in self.marks.items():
            for i in grades:
                if mark < i[0]:
                    print(subject, " : ", prev_grade)
                    break
                prev_grade = i[1]

    def add_mark(self, subject, mark):
        if subject in self.marks.keys():
            print(student_name, " already has a mark for ",
                  subject)
            user_input = input("Overwrite Y/N? ")
            if user_input == "Y" or "y":
                self.marks[subject] = mark
                return "Student's mark updated"
            else:
                return "Student's mark not updated"
        else:
            self.marks[subject] = mark
            return "Student's mark added"

class Students():
    def __init__(self, all_students):
        self.students = []
        for student, mark in all_students:
            self.add_student(student, mark)

    def add_student(self,student_name, marks = {}):
        if self.exists(student_name):
            return "Student already in database"
        else:
            self.students.append(Student(student_name, marks))
            return "Student added"
```

```
    def print_report_cards(self, student_name = None):
        for student in self.students:
            if student_name == None or student.name:
                student.print_report_card()

    def exists(self, student_name):
        for student in self.students:
            if student_name == student.name:
                return True
        return False

    def add_mark(self, student_name, subject, mark):
        for student in self.students:
            if student_name == student.name:
                return student.add_mark(subject, mark)
        return "Student not found"

students = Students(student_data)
print(students.students)
while True:
    print("Welcome to the Raspberry Pi student database")
    print("What can I help you with?")
    print("Enter 1 to view all report cards")
    print("Enter 2 to view the report card for a student")
    print("Enter 3 to add a student")
    print("Enter 4 to add a mark to a student")
    print("Enter 5 to exit")

    try:
        user_choice = int(input("Choice: "))
    except ValueError:
        print("That's not a number I recognise")
        user_choice = 0

    if user_choice == 1:
        students.print_report_cards()
    elif user_choice == 2:
        enter_student = input("Which student? ")
        students.print_report_cards(enter_student)
    elif user_choice == 3:
        enter_student = input("Student name? ")
        print(students.add_student(enter_student))
    elif user_choice ==4:
        enter_student = input("Student name? ")
        enter_subject = input("Subject? ")
        num_error = True
        while num_error:
            num_error = False
            try:
                enter_mark = int(input("Mark? "))
            except ValueError:
```

```
                print("I don't recognise that as a number")
                num_error = True
            print(students.add_mark(enter_student, enter_subject,
                                    enter_mark))
        elif user_choice == 5:
            break
        else:
            print("Unknown choice")
        input("Press enter to continue")

print("Goodbye and thank you for using the Raspberry"
      , "Pi Student database")
```

3.7 使用模块获得附加特性

到目前为止，你已经见过 import 很多次了，但我们并没有解释它们究竟做些了什么。事实上，非常简单，import 只是将 Python 代码从另外一个文件中转移到当前程序中。如果你创建一个名为 module_example.py 的文件，并写入下面这行：

```
print("Hello World")
```

将其保存在你的 home 目录（对于默认用户就是 /home/pi）。现在，在 IDLE3 的 Python 解释器中输入：

```
>>> import module_example
Hello World
```

当然，这是个毫无意义的模块。通常模块中包括可以使用的函数或对象。把 module_example 改成：

```
def hello():
   print ("Hello World")
```

你必须重启 IDLE 才能重新使用新代码。重启之后运行：

```
>>> import module_example
>>> module_example.hello()
```

第一行将 module_example 的所有函数和类导入你的工程中，在函数或类名前加上模块名作为前缀就可以使用它们了。有时，你只需要模块中的一部分，那么可以只导入其中的一部分：

```
>>> from module_example import hello
```

现在，只要输入下面这行就可以运行该函数：

```
>>> hello()
```

注意，这里不需要加模块名前缀。因为我们已经将其导入当前名字空间。这样做的时候需要确保它和自己定义的其他函数或类没有冲突。为了方便使用，你也可以将某个模块的所有东西都导入当前名字空间：

```
>>> from module_example import *
```

这样做需要非常小心，以避免名字空间冲突。

使用模块而不是将所有东西放入一个文件有很多优点。它使得代码可以在不同工程间复用（关于代码复用，还记得我们是怎么说的么？）。它还可以避免将大工程保存在单个文件中带来的不便。你可以将不同的模块分给不同的组，方便团队一起工作。

后继章节中，我们将围绕几个特别的模块，为你的树莓派工程增加许多足够酷的特性。

3.8 小结

我们并没有完整覆盖 Python 的所有特性。这样做需要写一本更厚的书，同样，在后继章节中，我们也没有足够的篇幅去介绍每个方面。然而，我们介绍的这些已经足够让你开始起步，同时也希望这些东西足够让你理解大多数 Python 程序。如果你卡在某个地方了，请翻到本章，或者查阅 Python 文档（它才是真的包含了所有特性，就是不太容易理解）。

学完本章，你应该了解下列内容：

- 变量可以保存数据。
- 数据具有类型，如 int、float 或 bool。
- 列表、元组、字典或集合可以将数据聚集起来以方便存取。
- while 循环会重复运行直到条件为假（False）。
- for 循环可以操作一组数据中的每个元素。
- 函数帮助你重用代码，可以避免重写它们。
- 类可以封装数据和方法使之更容易使用。
- Python 有上百个模块，在程序中导入它们可以增加附加特性。
- 你可以创建自己的模块。这样可以帮助把程序分割成多个方便管理的文件。

3.9　习题答案

练习 1

变量包括：

- `prompt_text` holds `"Enter a number: "` which is a string.
- `user_in` holds whatever the user types and is a string.
- `user_num` holds the number version of whatever the user types, converted into an `int`.
- `i` holds the numbers 1 to 9 as `int`.
- `even` is a `bool` that holds `True` or `False` (depending on whether the user's number is even).

练习 2

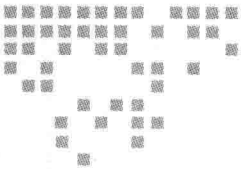
```
import random
secret = int(random.uniform(0,10))
print("I'm thinking of a number between zero and ten."
      , "Can you guess what it is?")
guess = 11

while guess != secret:
    try:
        guess = int(input("Take a guess: "))
    except ValueError:
        print("A number! Guess a number!")

print("Well done!")
```

练习 3

从技术上来说是可行的。但只是无谓地增加了代码复杂性。因为你无法改变它的值，必须每次创建新的元组而不能够简单地增加或修改当前值。



Chapter 4 第 4 章

图形编程

第 3 章中，我们介绍了很多关于如何加工、处理数据的方法，毕竟操纵数据是计算机的基本工作。你已经见过如何构建一个简单的字符驱动菜单来帮助控制程序。然而，这种风格在 20 世纪 80 年代已经不流行了。现在，虽然它在某些应用中还有使用，但是大多数人都喜欢用鼠标（或者触摸屏）。

你可能见过这三种不同类型的图形工具箱 Tk、GTK 和 Qt。TK 是古典样式的库，现在还有应用，但缺乏现代特性。GTK 是一个很流行的工具箱，其中 LXDE（Raspbian 默认桌面[⊖]）就使用了 GTK。Qt（有时读作 cute）由诺基亚原创，当时主要是为了它们那个时运不佳的智能手机。后来被诺基亚卖给 Digia 公司，由其继续开发。GTK 和 Qt 都可以免费使用，说实话，在这两个中间没什么好选择的。本章选择使用 Qt，是因为它有较好的可移植性和较好的技术支持。

开始使用 pyside 模块之前需要先安装它。在 LXTerminal 中，输入：

```
sudo apt-get install python3-pyside
```

整个过程可能需要一点时间，可以先去喝杯茶。

4.1 图形用户界面（GUI）编程

贯穿本书，你将发现如果使用正确的模块来完成那些繁重工作，在 Python 里可以非

⊖ 在本书翻译时，有消息称由于后向兼容性，Raspbian 将放弃 GTK 转投 Qt。——译者注

常轻易完成很多事情。图形用户界面（GUI）并不难。这里有各种各样有用的小工具，你需要做的就是选择几个需要的，并将其添加到自己的程序中。

你也可以使用继承。第3章中，我们介绍了类，你可以创建一个继承了超类所有特性的新类。这里，你将看到如何快速使用该方法基于原先的类创建一个新类。

让我们直接看下例子。第2章中，你见过turtle模块，也知道如何设置它使其等待按键按下。这比简单的文字输入要好，但也好不了太多。因此这里的第一个例子中，你将看到如何创建一个简单的GUI来控制turtle。让我们从下面的代码开始（同样的，可以手工输入或从网站上找到文件chapter4-turtle-start.py）：

```
import turtle
import sys
from PySide.QtCore import *
from PySide.QtGui import *

class TurtleControl(QWidget):
    def __init__(self, turtle):
        super(TurtleControl, self).__init__()
        self.turtle = turtle

        self.left_btn = QPushButton("Left", self)
        self.right_btn = QPushButton("Right", self)
        self.move_btn = QPushButton("Move", self)
        self.distance_spin = QSpinBox()

        self.controlsLayout = QGridLayout()
        self.controlsLayout.addWidget(self.left_btn, 0, 0)
        self.controlsLayout.addWidget(self.right_btn, 0, 1)
        self.controlsLayout.addWidget(self.distance_spin,1 , 0)
        self.controlsLayout.addWidget(self.move_btn,1 , 1)
        self.setLayout(self.controlsLayout)

        self.distance_spin.setRange(0, 100)
        self.distance_spin.setSingleStep(5)
        self.distance_spin.setValue(20)
#set up turtle
window = turtle.Screen()
babbage = turtle.Turtle()

# Create a Qt application
app = QApplication(sys.argv)
control_window = TurtleControl(babbage)
control_window.show()

# Enter Qt application main loop
app.exec_()
sys.exit()
```

现在就可以运行它，但上面的按钮还没有动作（等会儿再来添加）。首先让我们来看看这里都做了些什么。主要代码集中在 TurtleControl 类中，它继承自 Qwidget（大多数 Qt 类以字母 Q 开头）。从 Qwidget 继承后，你得到了所需的全部基本功能。现在需要做的就是将其从一个通用组件变得满足适合本程序需要。只需要告诉它，你需要哪些组件就可以了。

这里有三个按钮和一个整数旋转框（spinbox，允许输入数字，并可以增大、减小该数字——看看运行后的程序就知道 spinbox 如何工作了）。

除了用户看到的这些组件，你还要添加一个布局（layout）。Qt 有几种不同的布局类型（后面将会遇到另外一种），本程序使用 QGridLayout。像本例这样的简单控制面板程序非常适合使用网格布局（grid layout）。它的基本原理是将窗口分割成网格，由你来告诉 Qt 希望将组件放入哪个网格中。如果用户改变了窗口大小，Qt 将动态调整网格充满整个空间，并保持原有组件在网格中的位置不变。

想要在窗口中显示任何部件，都需要将它们加入布局（layout）。例如：

```
self.controlsLayout.addWidget(self.right_btn, 0, 1)
```

其中 0 和 1 分别表示从窗口左上角开始的水平和垂直坐标（即将平面直角坐标系上下颠倒一下）。这里，按钮位于窗口最上方一行，从左边数第二列。

当所有条目都添加到布局之后，你要用下面这行代码告诉窗口可以使用该布局。

```
self.setLayout(self.controlsLayout)
```

你还可以通过修改窗口组件的一些设置而改变它们的行为。本例中，可以使用以下方法调整整数旋转框（spinbox）：

```
self.distance_spin.setRange(0, 100)
self.distance_spin.setSingleStep(5)
self.distance_spin.setValue(20)
```

分别设置了最大最小值，以及每次单击改变的步长和初始值。

希望你能认出了刚开始的 turtle 代码。最后五行用来创建控制窗口并运行它。

4.2 添加控制

这些代码只是创建了一个漂亮的控制窗口，实际上并没有做任何事情。下一步要告诉

Qt 你要如何控制它。通过为事件链接一个动作就可以做到这一点。这里的事件指的是点击按键，动作指的是当事件发生时你希望运行的方法。

为了做到这点，请把下面代码添加到 TurtleControl 类的 __init__ 方法末尾：

```
        self.move_btn.clicked.connect(self.move_turtle)
        self.right_btn.clicked.connect(self.turn_turtle_right)
        self.left_btn.clicked.connect(self.turn_turtle_left)

    def turn_turtle_left(self):
        self.turtle.left(45)

    def turn_turtle_right(self):
        self.turtle.right(45)

    def move_turtle(self):
        self.turtle.forward(self.distance_spin.value())
```

你可能会注意到在每次链接方法时，不像平时调用方法一样，作为参数的方法后面没有跟小括号。就像这样：

```
self.move_btn.clicked.connect(self.move_turtle)
```

而不是这样：

```
self.move_btn.clicked.connect(self.move_turtle())
```

这是因为在方法后面加上括号，就是告诉 Python 运行该方法并运行结果作为参数。就像：

```
def move_turtle(self):
    self.turtle.forward(self.distance_spin.value())
```

然而，如果你不在方法后面加括号，就是告诉 Python 将该方法本身作为参数。这正是你在这里要做的事情，如此，Qt 才知道在事件发生时去运行什么。

在本书网站上可以找到完整版的代码，文件名是 chapter4-turtle.py。图 4-1 给出了程序运行时的截图。

基本上 Pyside 和 Qt 就这么多内容，不会再出现更复杂的问题。但其含有大量的组件，我们不可能将它们全部演示一遍。在下一个例子中，为使你熟悉该工具箱，我们会将其尽可能多地展示出来，这样以后需要时就可以使用其他组件了。

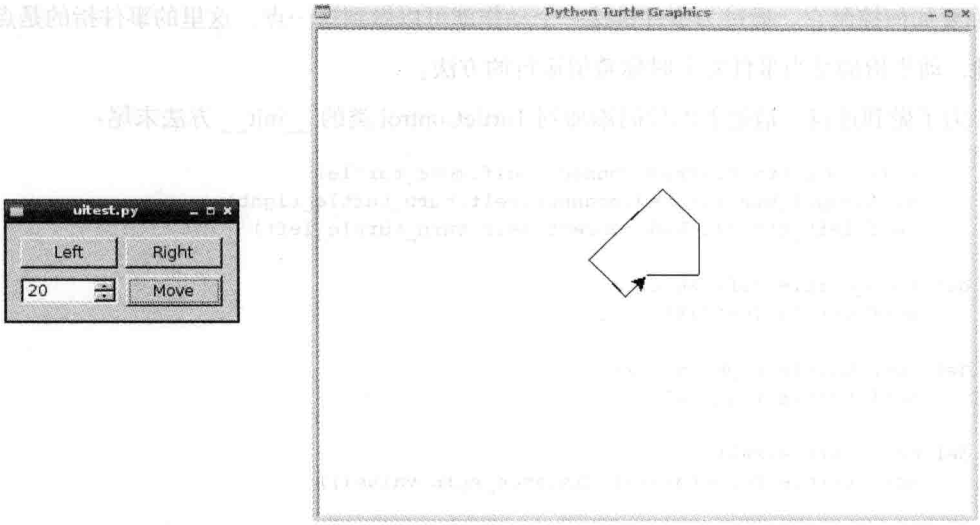


图 4-1 Turtle 的鼠标加强版接口

知识测试

练习 1

扩展 turtle 控制程序，当移动 turtle 时可以改变它的颜色。这里给出了一些提示，如果将设置 turtle 的代码改为：

```
#set up turtle
window = turtle.Screen()
babbage = turtle.Turtle()
window.colormode(255)
```

你就可以把 turtle 的颜色设为从 1 至 255 的红色、绿色、蓝色。如：

```
turtle.color(100,0,0)
```

另一个有用的东西是 QLables。它可以给窗体增加小段文字描述。可以这样做：

```
self.red_label = QLabel("Red", self)
```

可以用它给整数旋转框加标签（或者其他任何标签）。

4.3 创建 Web 浏览器

前面的例子中可以看出我们可以简单地将组件组合起来创建一个界面。在下面这个例

子中，我们将使用 Qt 组件创造一个自己的浏览器（其实也是由一系列组件组合出来的）。你将会发现，其实不用真正编程实现，只需要将不同的 Qt 部分组合起来就可以了。

首先你需要为浏览器创建一个窗体。上个例子中，你自己创建一个组件，Qt 为它生成了一个窗体。对于简单的工具这样子就足够了。但这次要建立一个更强大的应用，就需要单独创建一个窗体并在它上面添加所有需要的东西。使用 QMainWindow，可以添加很多东西，比如说菜单。当然，在添加菜单前需要先添加些其他东西。

对于任何浏览器来说，最重要的就是显示网页。关于网页中包含了什么东西，第 7 章将会详细介绍。对于本章来说，你只需要知道，QWebView 可以包办这一切就好了。

上个例子中，你使用了网格布局。对于大多数控制面板，它都可以很好地工作。在这个应用中，你将会使用到盒子布局。区别于网格，它有两种不同的布局，QVBoxLayout（垂直布局）和 QHBoxLayout（水平布局），分别是水平盒子和垂直盒子。在盒子布局中添加条目时，它将会水平或垂直地挨着排列在一起。将这些布局以某种合适的方式相互嵌套在一起可以创建出复杂的布局。当然，这需要较长时间练习才能做到。一旦你熟悉之后，就会发现它相当强大。

我们的 Web 浏览器将具有典型的 Web 浏览器布局。即顶部有个控制条，用户浏览的网页将占据剩下的大部分窗体。为创建这种界面，你需要两种布局，一个控制条用的 QHBoxLayout，和一个用来装前面的一个控制条和 QWebView 组件的 QVBoxLayout。当改变窗体尺寸时，Qt 将自动调整布局，各个组件将能很好地利用空间。

希望随着创造浏览器的过程，你会更清楚这些东西，就让我们开始吧！下面的代码创建了窗体，并添加上了适当的组件（参见本书网站中文件 chapter4-Web-browser-begin.py）。

```
import sys
from PySide.QtCore import *
from PySide.QtGui import *
from PySide.QtWebKit import *

class Browser(QWidget):

    def __init__(self):
        super(Browser, self).__init__()

        self.webview = QWebView(self)
        self.webview.load("http://www.google.com")
        self.setGeometry(0, 0, 800, 600)

        self.menu_bar = QHBoxLayout()
        self.main_layout = QVBoxLayout()
        self.main_layout.addLayout(self.menu_bar)
```

```
        self.main_layout.addWidget(self.webview)
        self.setLayout(self.main_layout)

class BrowserWindow(QMainWindow):
    def __init__(self):
        super(BrowserWindow, self).__init__()
        self.widget = Browser()
        self.setCentralWidget(self.widget)

# Create a Qt application
app = QApplication(sys.argv)
window = BrowserWindow()
window.show()

# Enter Qt application main loop
app.exec_()
sys.exit()
```

这就是一个简单的 Web 浏览器所需要的全部代码。你可以运行一下，它将会打开 Google 的主页，你可以从这里开始浏览（见图 4-2）。你应该已经很熟悉这些代码。新的内容包括 QMainWindow（后面将会介绍如何控制它），QWebView（如你所见，这是个可以轻松添加网络浏览器的方法）和盒子布局。

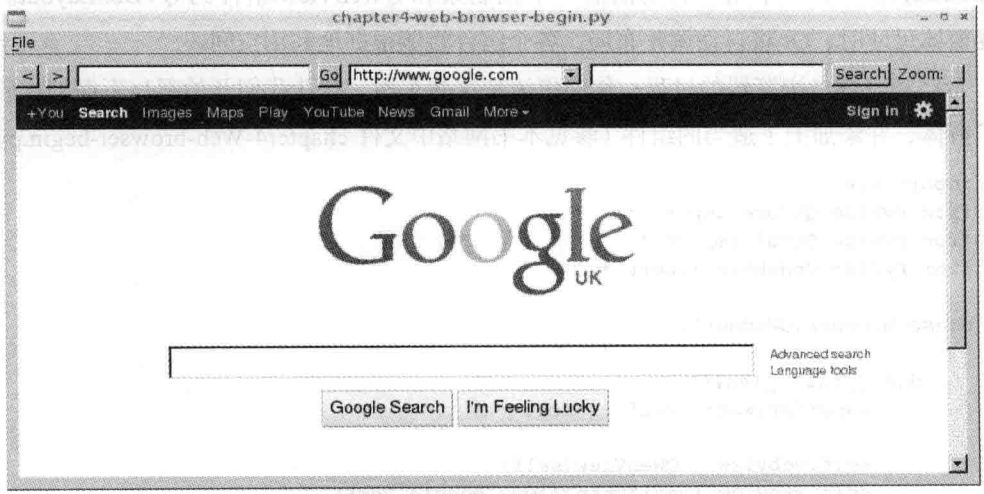


图 4-2　网络浏览器基础

盒子布局已经全部建立起来，你需要做的就是给浏览器的 self.menu_bar 添加条目，它们将出现在屏幕顶端。

Web浏览器的最基本控制部分就是后退和前进按钮。在这个任务中，我们可以像在上个例子中一样使用QPushButtons。将下面的黑体部分加入你的代码中的Browser类中：

```
class Browser(QWidget):

    def __init__(self):
        super(Browser, self).__init__()

        self.webview = QWebView(self)
        self.webview.load("http://www.google.com")
        self.setGeometry(0, 0, 800, 600)

        self.back_btn = QPushButton("<", self)
        self.back_btn.clicked.connect(self.webview.back)
        self.back_btn.setMaximumSize(20,20)

        self.forward_btn = QPushButton(">", self)
        self.forward_btn.clicked.connect(self.webview.forward)
        self.forward_btn.setMaximumSize(20,20)

        self.menu_bar = QHBoxLayout()
        self.menu_bar.addWidget(self.back_btn)
        self.menu_bar.addWidget(self.forward_btn)
        self.main_layout = QVBoxLayout()
        self.main_layout.addLayout(self.menu_bar)
        self.main_layout.addWidget(self.webview)

        self.setLayout(self.main_layout)
```

现在运行一下，你的浏览器将拥有历史功能，可以前进和后退。当然，QWebView完成了所有繁重的工作，这里只需要将点击按钮和QWebView的前进和后退方法连接起来。



注意 前面提到过，但这里仍值得再次强调——编程时，可以从已有模块中获得的功能，不需要重新再实现一遍。花上一点时间学习一个模块可以在后面节省你很多时间。

和前面使用的网格分布不同，盒子分布中，Qt为计算绘制组件的尺寸提供了更多自由。有时这很不错，但其他时候，你需要给它更多指导。在Web浏览器中，你希望按钮占用尽可能少的空间，把空余屏幕都让给网页。为了做到这一点，我们在添加的组件上调用setMaximumSize()方法。比如按钮，用来保证它们不会大于20×20。

拥有一个文本输入框是Web浏览器的另一个特性，用户可以输入想要访问的站点地址。几种不同的Qt组件都可以用作文本输入，最常见的就是QTextEdit。它可以用来显示或编辑文本。实际上，除了处理文字，它还可以处理图像、表格、标题等其他类似的东西。

QPlainTextEdit 是另一个类似于 QTextEdit 的常见组件，只不过它只能用作处理简单文本，不能处理复杂文本。它们都有十分强大的选项，在你的编程生涯中有很大机会能用的到。由于它们被设计来处理多行文本，用作地址栏有点浪费。对于单行的简单文本（如 URL 栏），QLineEdit 是最好的选择。

你还需要一个“转到”按钮，用来告诉浏览器去加载页面。按下列代码更新 Browser 类（更新部分以黑体显示）就可以做到这些。

```
class Browser(QWidget):

    def __init__(self):
        super(Browser, self).__init__()

        self.webview = QWebView(self)
        self.webview.load("http://www.google.com")
        self.setGeometry(0, 0, 800, 600)

        self.back_btn = QPushButton("<", self)
        self.back_btn.clicked.connect(self.webview.back)
        self.back_btn.setMaximumSize(20,20)

        self.forward_btn = QPushButton(">", self)
        self.forward_btn.clicked.connect(self.webview.forward)
        self.forward_btn.setMaximumSize(20,20)
        self.url_entry = QLineEdit(self)
        self.url_entry.setMinimumSize(200,20)
        self.url_entry.setMaximumSize(300,20)

        self.go_btn = QPushButton("Go", self)
        self.go_btn.clicked.connect(self.go_btn_clicked)
        self.go_btn.setMaximumSize(20,20)

        self.menu_bar = QHBoxLayout()
        self.menu_bar.addWidget(self.back_btn)
        self.menu_bar.addWidget(self.forward_btn)
        self.menu_bar.addWidget(self.url_entry)
        self.menu_bar.addWidget(self.go_btn)
        self.menu_bar.addStretch()
        self.main_layout = QVBoxLayout()
        self.main_layout.addLayout(self.menu_bar)
        self.main_layout.addWidget(self.webview)

        self.setLayout(self.main_layout)

    def go_btn_clicked(self):
        self.webview.load(self.url_entry.text())
```

这里有些新东西。一个是在 URL 栏中调用了 setMinimumSize() 方法。像 setMaximumSize

一样，它向 Qt 传递一些你希望怎么显示窗体的额外信息。另一个帮助 Qt 恰当地排列窗体的新方法是调用 addStretch()。它添加了一个虚假组件，仅仅用来填充剩余空间。本例中，它占据了菜单栏右边的所有空间，因此 Qt 把所有的控制组件放到左边。

现在可以运行下试试看。唯一需要注意的是你需要在 Web 地址前输入 http://。（从技术角度看是因为 URL 或全球资源定位器需要指定网络协议。例如，http://yoursite.com/document 和 ftp://yoursite.com/document 就指向不同地方。大多数现代浏览器都允许你省略网络协议并假定你使用的是 http。然而，当一个组件请求 URL 地址时，它仍然需要协议前缀。）

本例中，我们为浏览器添加了一个叫 go_btn_clicked() 的新方法。与将方法和事件链接起来相比，这样做会更强大。self.url_entry.text() 返回用户输入的文本，它允许你使用这个文本作为参数调用 Webview 的 load 方法。

此时，你已经拥有一个真正意义上的浏览器。虽然没有主流浏览器，如火狐、谷歌浏览器或 Midori 功能强大，但它并不缺少任何基本要素。我们下一个要添加的功能是书签选择器。选择它一半是因为这是个有用的特性，一半是因为它让我们有理由来炫耀另一个有用的 Qt 组件——组合框（QComboBox）。

组合框这个名字比起其他你已经十分熟悉的组件来显得有点古怪。这是一个带有下拉箭头的框，用户可以在下拉列表中选择。如果还不清楚，很快你就能看到了。

本书稍后会介绍几种在对象之间保存信息的方法。但简单起见，我们不让用户改变或添加书签。毕竟，本章只是针对用户界面，我们将紧紧围绕这个主题。

在 Browser 类中添加下面例子中的黑体部分（非黑体部分是为了便于你知道添加到哪里）：

```
self.go_btn = QPushButton("Go", self)
self.go_btn.clicked.connect(self.go_btn_clicked)
self.go_btn.setMaximumSize(20,20)

self.favourites = QComboBox(self)
self.favourites.addItems(["http://www.google.com",
                          "http://www.raspberrypi.org",
                          "http://docs.python.org/3/"])
self.favourites.activated.connect(self.favourite_selected)
self.favourites.setMinimumSize(200,20)
self.favourites.setMaximumSize(300,20)

self.menu_bar = QHBoxLayout()
self.menu_bar.addWidget(self.back_btn)
self.menu_bar.addWidget(self.forward_btn)
self.menu_bar.addWidget(self.url_entry)
```

```
self.menu_bar.addWidget(self.go_btn)
self.menu_bar.addStretch()
self.menu_bar.addWidget(self.favourites)
self.main_layout = QVBoxLayout()
self.main_layout.addLayout(self.menu_bar)
self.main_layout.addWidget(self.webview)

self.setLayout(self.main_layout)

def go_btn_clicked(self):
    self.webview.load(self.url_entry.text())

def favourite_selected(self):
    self.webview.load(self.favourites.currentText())
```

十分简单，运行下代码，你就可以看到 QcomboBox 执行结果。

像 URL 一样，我们也调用了 self.Webview.load。但这次使用的参数是当前从组合框中选择的文本。

还有两个控制项需要添加到菜单栏中，让我们一次把它们都做出来。第一个是搜索条，输入搜索内容，然后按下按钮就可以运行谷歌搜索。第二个是缩放条，允许用户放大、缩小页面。

使用下面黑体部分更新 Browser 类：

```
self.favourites = QComboBox(self)
self.favourites.addItems(["http://www.google.com",
                          "http://www.raspberrypi.org",
                          "http://docs.python.org/3/"])
self.favourites.activated.connect(self.favourite_selected)
self.favourites.setMinimumSize(200,20)
self.favourites.setMaximumSize(300,20)

self.search_box = QLineEdit(self)
self.search_box.setMinimumSize(200,20)
self.search_box.setMaximumSize(300,20)

self.search_btn = QPushButton("Search", self)
self.search_btn.clicked.connect(self.search_btn_clicked)
self.search_btn.setMaximumSize(50,20)

self.zoom_slider = QSlider(Qt.Orientation(1),self)
self.zoom_slider.setRange(2, 50)
self.zoom_slider.setValue(10)
self.zoom_slider.valueChanged.connect(self.zoom_changed)

self.zoom_label = QLabel("Zoom:")

self.webview.loadStarted.connect(self.page_loading)
```

```
self.menu_bar = QHBoxLayout()
self.menu_bar.addWidget(self.back_btn)
self.menu_bar.addWidget(self.forward_btn)
self.menu_bar.addWidget(self.url_entry)
self.menu_bar.addWidget(self.go_btn)
self.menu_bar.addStretch()
self.menu_bar.addWidget(self.favourites)
self.menu_bar.addStretch()
self.menu_bar.addWidget(self.search_box)
self.menu_bar.addWidget(self.search_btn)
self.menu_bar.addWidget(self.zoom_label)
self.menu_bar.addWidget(self.zoom_slider)
self.main_layout = QVBoxLayout()
self.main_layout.addLayout(self.menu_bar)
self.main_layout.addWidget(self.webview)

self.setLayout(self.main_layout)

def go_btn_clicked(self):
self.webview.load(self.url_entry.text())

def favourite_selected(self):
self.webview.load(self.favourites.currentText())

def zoom_changed(self):
self.webview.setZoomFactor(self.zoom_slider.value()/10)

def search_btn_clicked(self):
self.webview.load("https://www.google.com/search?q="
                  + self.search_box.text())

def page_loading(self):
self.url_entry.setText(self.webview.url().toString())
```

虽然有两个控制项，却要用四个组件来完成。搜索框有一个 QLineEntry 和一个 QPushButton。类似于之前添加的 URL 控制项，它们协同工作。只不过它会在你输入的字符前面加上 https://www.google.com/search?q= 。例如，如果搜索树莓派，它将打开 URL http://www.google.com/search?q=Raspberries，也就是让谷歌去搜索树莓派。这里用了 https:// 而不是 http://。字母 s 代表安全（secure），如果使用 https://，浏览器到网站的所有数据都会被加密。然而，并不是所有网站都支持 https。只要服务器支持，QWebView 可以使用两者中的任何一个。

缩放按钮是 QSlider，这是另外一个你可能熟悉的控制形式。它需要设置的地方会多一点，下面就是如何设置的例子：

```
self.zoom_slider.setRange(2, 50)
self.zoom_slider.setValue(10)
```

第一行设置了滑块的最大和最小值，第二行设置了初始值。

把 valueChanged 动作连接到 zoom_changed() 方法。这里又是连接到一个 QWebView 的方法，把复杂的工作交给它。zoome_changed() 唯一做的就是将滑块的值除以 10 以保证缩放更容易控制。

如果仔细观察，你会发现我们不止添加了两个额外的控制部分。还有这些行：

```
  self.webview.loadStarted.connect(self.page_loading)
. . .
def page_loading(self):
  self.url_entry.setText(self.webview.url().toString())
```

它们保证 URL 框能够实时显示当前网页的地址。

4.4 添加窗口菜单

主浏览器布局已经完成，但在应用程序完工前还有很多需要添加。还记得开始时我们说过通过扩展 QMainWindow 可以添加菜单吗？现在我们就开始做。

窗体已经有一个菜单，你需要做的就是添加些东西进去。菜单项和组件类似，只不过我们通过 QActions 创建。

把 BrowserWindow 类改成下面的样子可以给文件菜单添加一项来关闭窗体：

```
class BrowserWindow(QMainWindow):
  def __init__(self):
    super(BrowserWindow, self).__init__()
    self.widget = Browser()
    self.setCentralWidget(self.widget)

    self.exitAction = QAction(QIcon('exit.png'), '&Exit', self)
    self.exitAction.setShortcut('Ctrl+Q')
    self.exitAction.setStatusTip('Exit application')
    self.exitAction.triggered.connect(self.close)

    self.menu = self.menuBar()
    self.fileMenu = self.menu.addMenu('&File')
    self.fileMenu.addAction(self.exitAction)
```

还有一个用来打开本地文件的菜单项要添加。不同于 Web 浏览器中的所有事务，它要打开一个新窗口。在新窗口中，用户可以浏览他们的文件，并选择一个想要打开的。此时，你可能觉得需要很多工作来创建一个新窗口，添加整个布局，并把所有需要的组件联合起来。然而，这里又可以让 Qt 来为我们完成繁重的工作。Qt 有一系列称为对话框的组件。它们就是用来完成通用功能的一些简单窗口，这可以大大减轻我们的工作。按下列方

式更新 BrowserWindow 类，就可以给 Web 浏览器加入打开文件对话框。

```
        self.exitAction = QAction(QIcon('exit.png'), '&Exit', self)
        self.exitAction.setShortcut('Ctrl+Q')
        self.exitAction.setStatusTip('Exit application')
        self.exitAction.triggered.connect(self.close)

        self.openFile = QAction(QIcon('open.png'), 'Open', self)
        self.openFile.setShortcut('Ctrl+O')
        self.openFile.setStatusTip('Open new File')
        self.openFile.triggered.connect(self.showDialog)

        self.menu = self.menuBar()
        self.fileMenu = self.menu.addMenu('&File')
        self.fileMenu.addAction(self.openFile)
        self.fileMenu.addAction(self.exitAction)

    def showDialog(self):
        fname, _ = QFileDialog.getOpenFileName(self, 'Open file',
                '/home')
        self.widget.webview.load("file:///" + fname)
```

本例中，我们不需要创建新的对象，只要调用 QFileDialog 的 getOpenFileName() 方法。这将会打开一个标题为“打开 /home 目录下的文件”（Open File in the directory /home）的新窗口（参见图 4-3）。一旦用户选择了想要的文件，它将会返回两个东西：文件名和过滤器。因为 Web 浏览器不需要知道过滤器，把它分配给 _ 并丢弃掉。

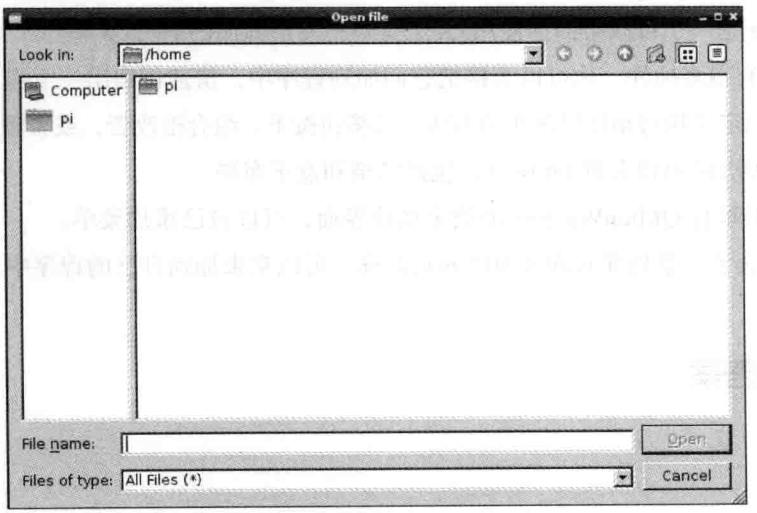


图 4-3 使用 QfileDialog 打开本地文件

QWebView 使用 file:///（注意这里是三个斜线）打开本地文件，因此使用时需要在文

件名前加上这个前缀。

至此，Web 浏览器就结束了。如果你没有跟上去，可以从网站上找到完整代码，文件名是 chapter-Web-browser-complete.py。和所有应用程序一样，我们还可以给它加上很多东西。但我们所做的足以用来介绍 pyside 模块和 Qt 工具箱。这是个庞大的工具箱，因此出于需要，我们只能介绍些基础。一切都记录在 http://srinkikom.github.io/pyside-docs/。

知识测试

练习 2

你已经看到对话框可以在程序中快速添加功能。在这个练习中，要使用之前的 turtle 程序，在其中添加一个打开 QColorDialog 的按钮，用来设置 turtle 颜色。

这里给出一些提示。QColorDialog.getColor() 会返回类型为 QColor 的值。使用 variable_name.getRgb()[:3] 可以从 QColor 的变量中获得 RBG 的值。在末尾需要使用 [:3]，因为它的返回值是一个包含 4 个值的元组（最后一个是透明度，这里用不到）。

4.5 小结

学完本章，应该了解以下内容：

- Pyside 是一个可以帮助你使用 Qt 工具包写图形界面的 Python 库。
- Qt 含有很多插件，你可以直接把它们加到程序中，快速创建出一个强大的界面。
- 你可以定义执行动作时调用的方法，如按钮按下，组合框改变，或者滚动条移动等。
- 有多种方法可以布置 Qt 窗口，包括网格和盒子布局。
- 使用继承自 QMainWindow 的类来构建界面，可以自己添加菜单。
- Qt 包含了一系列拥有现成窗口的对话框，可以拿来加到自己的程序中。

4.6 习题答案

练习 1

```
import turtle
import sys
from PySide.QtCore import *
from PySide.QtGui import *
```

```
class TurtleControl(QWidget):
    def __init__(self, turtle):
        super(TurtleControl, self).__init__()
        self.turtle = turtle

        self.left_btn = QPushButton("Left", self)
        self.right_btn = QPushButton("Right", self)
        self.move_btn = QPushButton("Move", self)
        self.distance_spin = QSpinBox()
        self.red_spin = QSpinBox()
        self.green_spin = QSpinBox()
        self.blue_spin = QSpinBox()
        self.red_label = QLabel("Red", self)
        self.green_label = QLabel("Green", self)
        self.blue_label = QLabel("Blue", self)
        self.colour_btn = QPushButton("Colour", self)
        self.controlsLayout = QGridLayout()
        self.controlsLayout.addWidget(self.left_btn, 0, 0)
        self.controlsLayout.addWidget(self.right_btn, 0, 1)
        self.controlsLayout.addWidget(self.distance_spin, 1 , 0)
        self.controlsLayout.addWidget(self.move_btn, 1 , 1)
        self.controlsLayout.addWidget(self.red_spin, 2,1)
        self.controlsLayout.addWidget(self.green_spin, 3,1)
        self.controlsLayout.addWidget(self.blue_spin, 4,1)
        self.controlsLayout.addWidget(self.red_label, 2,0)
        self.controlsLayout.addWidget(self.green_label, 3,0)
        self.controlsLayout.addWidget(self.blue_label, 4,0)
        self.controlsLayout.addWidget(self.colour_btn, 5,0)
        self.setLayout(self.controlsLayout)

        self.distance_spin.setRange(0, 100)
        self.distance_spin.setSingleStep(5)
        self.distance_spin.setValue(20)

        for spinner in [self.red_spin, self.green_spin,
                        self.blue_spin]:
            spinner.setRange(0, 255)
            spinner.setSingleStep(5)
            spinner.setValue(150)

        self.move_btn.clicked.connect(self.move_turtle)
        self.right_btn.clicked.connect(self.turn_turtle_right)
        self.left_btn.clicked.connect(self.turn_turtle_left)
        self.colour_btn.clicked.connect(self.colour_turtle)

    def turn_turtle_left(self):
        self.turtle.left(45)

    def turn_turtle_right(self):
        self.turtle.right(45)
```

```
def move_turtle(self):
    self.turtle.forward(self.distance_spin.value())

def colour_turtle(self):
    self.turtle.color(self.red_spin.value(),
                      self.green_spin.value(),
                      self.blue_spin.value())
#set up turtle
window = turtle.Screen()
babbage = turtle.Turtle()
window.colormode(255)

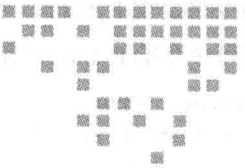
# Create a Qt application
app = QApplication(sys.argv)
control_window = TurtleControl(babbage)
control_window.show()

# Enter Qt application main loop
app.exec_()
sys.exit()
```

练习 2

需要将下列函数连接到点击按钮动作上。

```
def colour_turtle(self):
   self.colour = QColorDialog.getColor()
   self.turtle.color(self.colour.getRgb()[:3])
```



第 5 章 Chapter 5

搭建游戏

在第 4 章中，你使用 GUI 工具箱构建了图形化软件。它使得在程序中添加按钮、文本框和各种小工具变得十分容易。但是，有另外一类图形化程序并不使用这些东西：游戏。我们仍然需要在屏幕上画一些东西，但不是复选框和菜单，我们需要火球、英雄、恶魔坑和各种梦幻图形。显然，PySide 不能胜任这个任务，但另一个模块 PyGame 可以帮我们完成这些工作。

Raspbian 内置了 PyGame，但只能用于 Python2。由于我们使用 Python3，我们需要安装一下。本例中，你需要从头编译这个模块，这是个学习如何编译的好机会。首先，你需要安装 PyGame 需要的所有包，所以，打开 LXTerminal，输入 apt-get（包管理器）来安装附属程序：

```
sudo apt-get update
sudo apt-get install libsdl-dev libsdl-image1.2-dev \
  libsdl-mixer1.2-dev libsdl-ttf2.0-dev libsmpeg-dev \
  libportmidi-dev libavformat-dev libswscale-dev \
  mercurial python3-dev
```

你可能会注意到，这些代码大部分都以 -dev 结尾。这些是开发文件。编译使用这些库的软件时要用到它们。

下一步就是获取 PyGame 的最新版本。由于我们使用 Python3，我们需要获取做新版本，因此我们直接从开发平台中下载：

```
hg clone https://bitbucket.org/pygame/pygame
cd pygame
```

这两行命令将会把 PyGame 的当前版本下载到一个名叫 pygame 的目录中。完成之后，你就可以用下列命令给 Python3 编译和安装该模块：

```
python3 setup.py build
sudo python3 setup.py install
```

如果一切顺利，就可以在 Python3 中使用 PyGame 了。在 IDLE 中打开 Python Shell，输入下面命令可以验证它是否工作：

```
>>> import pygame
```

这里如果出现任何错误，就意味着哪里出错了，在继续阅读本章之前，你需要回过头来重复前面步骤。如果一切正常，可以尝试下列命令观察 PyGame 是如何工作的：

```
>>> pygame.init()
>>> window = pygame.display.set_mode((500, 500))
>>> screen = pygame.display.get_surface()
>>> rect = pygame.Rect(100, 99, 98, 97)
>>> pygame.draw.rect(screen, (100, 100, 100), rect, 0)
>>> pygame.display.update()
```

你会看到这些命令打开了一个新窗口并绘制了一个灰色矩形。第一行创建并运行 GyGame。第二行打开一个新窗口。参数 -(500，500)- 是一个包含新窗口宽度和高度的元组。注意这里为什么要有两层圆括号？外层括号用来传递参数，里层括号表示元组。第三行获得一个可以在上面绘画的平面，并将其存储在变量 screen 中。

你将会用到两个主要的 PyGame 类：Rect 和 Sprite。第二个我们稍后再看。第一个，Rect，对于 PyGame 的工作方式来说非常重要。第四行就创建了这样一个矩形，它的左上角坐标位于 100,99，宽 98 像素，高 97 像素。这里你需要知道，PyGame 的坐标是从屏幕的左上角开始，相对于正常图形坐标，它是上下颠倒的。

矩形并不总是会显示到屏幕上（后面你将会看到它还有其他用途），在下一行代码中，我们将该矩形绘制在屏幕上。参数 (100, 100, 100) 表示红色、绿色和蓝色值（每一个取值范围都在 0 到 255 之间），0 表示线宽（0 表示填充，1 或者大于 1 表示线的具体宽度）。当输入这些代码时，如果你注意观察，会发现矩形并没有出现在屏幕上。这是因为要使该变化生效，你需要刷新屏幕，正如我们在最后一行所做的那样。

你可以尝试在屏幕上画出更多矩形，体会一下不同的参数是如何影响形状的。

5.1 构建游戏

这些是 PyGame 最基础的部分。现在我们要构建一个简单的平台游戏。在这个系统，

你要控制一个角色跑动并跳跃障碍直到完成目标。为了增加难度，我们从屏幕上方掉落下火球，游戏角色需要躲开它们。角色如果从平台上掉下来，就会被末日之火终结。

开始编程时，我们通常不是首先开始写代码。而是根据自己对程序如何运行的理解制订一个计划。这有点像程序的骨架，我们稍后会使之更充实和具体。我们首先会设计出所有的类和方法，但不去具体实现它们。接下来我们开始编程，不断丰富这些设计，直到编程结束。对于这个游戏，我们给出下面的设计（你可以把它打出来，也可以简单地从网站上下载文件 chapter5-pygame-skell.py，然后再阅读本章过程中给它添加内容）：

```
import pygame
import sys
from pygame.locals import *
from random import randint

class Player(pygame.sprite.Sprite):
    '''The class that holds the main player, and controls
    how they jump. nb. The player doesn't move left or right,
    the world moves around them'''

    def __init__(self, start_x, start_y, width, height):
        pass

    def move_y(self):
        '''this calculates the y-axis movement for the player
        in the current speed'''
        pass

    def jump(self, speed):
        '''This sets the player to jump, but it only can if
        its feet are on the floor'''
        pass

class World():
    '''This will hold the platforms and the goal.
    nb. In this game, the world moves left and right rather
        than the player'''

    def __init__(self, level, block_size, colour_platform,
                 colour_goals):
        pass

    def move(self, dist):
        '''move the world dist pixels right (a negative dist
        means left)'''
        pass

    def collided_get_y(self, player_rect):
        '''get the y value of the platform the player is
```

```
            currently on'''
            pass

        def at_goal(self, player_rect):
            '''return True if the player is currently in contact
            with the goal. False otherwise'''
            pass

        def update(self, screen):
            '''draw all the rectangles onto the screen'''
            pass

    class Doom():
        '''this class holds all the things that can kill the player'''

        def __init__(self, fireball_num, pit_depth, colour):
            pass

        def move(self, dist):
            '''move everything right dist pixels (negative dist
            means left)'''
            pass
        def update(self, screen):
            '''move fireballs down, and draw everything on
            the screen'''
            pass

        def collided(self, player_rect):
            '''check if the player is currently in contact with
            any of the doom.
            nb. shrink the rectangle for the fireballs to
            make it fairer'''
            pass

    class Fireball(pygame.sprite.Sprite):
        '''this class holds the fireballs that fall from the sky'''

        def __init__(self):
            pass

        def reset(self):
            '''re-generate the fireball a random distance along
            the screen and give them a random speed'''
            pass

        def move_x(self, dist):
            '''move the fireballs dist pixels to the right
            (negative dist means left)'''
            pass

        def move_y(self):
```

```
        '''move the fireball the appropriate distance down
        the screen
        nb. fireballs don't accellerate with gravity, but
        have a random speed. if the fireball has reached the
        bottom of the screen, regenerate it'''
        pass

    def update(self, screen, colour):
        '''draw the fireball onto the screen'''
        pass

#options
#initialise pygame.mixer
#initialise pygame
#load level
#initialise variables
finished = False
#setup the background
while not finished:
    pass
    #blank screen
    #check events
    #check which keys are held
    #move the player with gravity
    #render the frame
    #update the display
    #check if the player is dead
    #check if the player has completed the level
    #set the speed
```

这里真正的代码不多，但已经足以我们了解这是怎么回事。这是一个真正 Python 程序，你可以运行它。它除了不停地循环运行不会做任何事情，按 Ctrl+C 组合键可以停止它。但是这给了你一个开工的基础。随着我们添加代码，我们可以确保它一直可以运行，这样我们就可以不断查看它是如何工作的，以确保它能正确地运行。注意每个方法里的 pass 语句，该语句不做任何事情，但如果一个方法里没有任何代码，Python 将会报错，因此我们需要这一行以确保代码能够运行。

5.2 初始化 PyGame

现在我们添加一些东西来启动它。请在文件中对应的注释后面插入下列代码。

```
#options
screen_x = 600
screen_y = 400
game_name = Awesome Raspberry Pi Platformer"
```

```
#initialise pygame
pygame.init()
window = pygame.display.set_mode((screen_x, screen_y))
pygame.display.set_caption(game_name)
screen = pygame.display.get_surface()
#initialise variables
clock = pygame.time.Clock()

    #check events
    for event in pygame.event.get():
        if event.type == QUIT:
            finished = True

    #set the speed
    clock.tick(20)
```

你应该能认出的 initialise pygame 后面的代码段是我们之前见到的。唯一的区别就是我们把屏幕尺寸换成两个变量。这样方便以后更改它们而不用记住哪些代码具体做了什么。所有关键选项都集中存储在全局变量中以允许你控制程序运行方式。

#check events 后面的部分用来等待用户按下关闭按钮关闭窗口后，退出循环。clock 的两行使用了 PyGame 的 timer 来设置合适的循环速度。由于每轮循环都对应游戏的一帧，我们需要确保它不会运行的太快（否则，在用户有机会操作之前所有的动作都结束了）。你只需要像在第 2 章的 turtle 游戏中那样，告诉 Python 休眠一个合适的时间即可，但有个小问题是你不知道剩下的循环体需要多长运行时间。如果你在一个较慢的计算机上运行，游戏的运行速度会和在较快的机器上不一样。clock.tick(fps) 可以解决这个问题。它会计算循环体运行所需时间，通过暂停一个合适的时间来确保每秒钟固定运行 fps 次循环。在上面代码中，最后一行会计算需要等待多久以确保每秒钟固定执行 20 次循环。

现在让我们从 Player 类开始构建每个类。请将下列代码添加到 Player 类中：

```
def __init__(self, start_x, start_y, width, height):
    pygame.sprite.Sprite.__init__(self)
    self.image = pygame.transform.scale(
        pygame.image.load(player_image), (width, height))
    self.rect = self.image.get_rect()
    self.rect.x = start_x
    self.rect.y = start_y
    self.speed_y = 0
    self.base = pygame.Rect(start_x, start_y + height, width,2)
```

Player 的关键点就是它继承自 pygame.sprite.Sprite。你可以使用它来绘制图像，但首先你需要设置两个关键变量：self.image 和 self.rect。一旦设置好它们，继承自 pygame.sprite.Sprite 的那部分会把图像画出来。显然，self.image 是你希望传递给 sprite 的图像，self.rect

是 PyGame 要绘制图像的矩形区域。设置完成后，你可以像操作其他矩形区域一样移动，操纵它，PyGame 会相应地移动图像。后面你将看到，改变矩形区域的 x、y 属性可以移动它。

倒数第二个局部变量（speed_y）用来追踪用户控制角色上下跳跃的速度。最后一个局部变量（base）是一个表示角色双脚的非常短小的矩形。你将会使用它来检测角色是否站在平台上。

现在角色已经就绪，可以显示到屏幕上，但是首先你还需要一些代码。将下列代码添加到合适的区域：

```
#options
player_spawn_x = 50
player_spawn_y = 200
player_image = "lidia.png"

#initialise variables
player = Player(player_spawn_x, player_spawn_y, 20, 30)
player_plain = pygame.sprite.RenderPlain(player)

#render the frame
player_plain.draw(screen)

#update the display
pygame.display.update()
```

为了绘制精灵，你需要先给它提供一幅图像。如果你有艺术细胞，可以自己创作一个图像。这个网站提供了大量可以用于制作游戏的图像 http://opengameart.org。它们都是可以免费下载并在游戏中使用的。但其中有些是需要许可的，即如果你在游戏中使用了它们并将游戏分发给其他人，你需要向他们提供 Python 代码。如果愿意，他们可以修改代码，或者使用你的代码构建其他游戏。这个就是大家所说的开源（参见注释）。http://opengameart.org 网站上的每个文件都有链接说明它们所遵守的许可。值得注意的是除非你要将游戏分发给其他人，否则你不用担心这些东西。并不是所有的文件都可以用于 PyGame。我们推荐使用后缀为 .png 的图像。经过搜索，我们使用 http://opengameart.org/sites/default/files/styles/medium/public/lidia.png 作为我们游戏的主角，你也可以自己选择其他图像（当然你需要相应得更新 player_image 变量）。目前的代码只会从它运行的目录中查找 player_image 图像文件。

开　源

开源是指人们相互共享他们编写的程序。不仅限于可执行程序，还包括程序源代码。其他人可以以自认为合适的方法随意修改代码。例如，树莓派上的 Linux 操作系统

就是开源的。因此所有的工具如 Midori 网络浏览器，甚至 Python 自己也遵循同样的标准。大体上说，有两类不同的开源形式——一个是自己做的任何修改都需要共享，另一个却不需要。知识共享（creative commons）和开源类似，只是换成了艺术品，如图像、声音和著作。在你的代码里引入开源软件片段或者知识共享艺术品时，了解你需要承担的责任是很重要的。通常这些都会有链接详细介绍它们遵守的许可。对于知识共享，很容易阅读和理解。如果你需要遵照许可共享你的代码（或者你希望开源自己的工作），最简单的方法就是将其放在开源网站上，如 github.com。网站将免费为你共享它。

为了绘出精灵，你还需要将其加入 RenderPlain。它将会创建一个可以绘制到屏幕上的对象。这里，角色对象有它自己的 RenderPlain 对象，叫做 player_plain。

现在你可以运行代码。它将把精灵显示到坐标 player_spawn_x,player_spanw_y。从技术上说，它将会每秒重绘 20 次，但由于它一直在同一个地方，你无法感觉到。如图 5-1 所示。

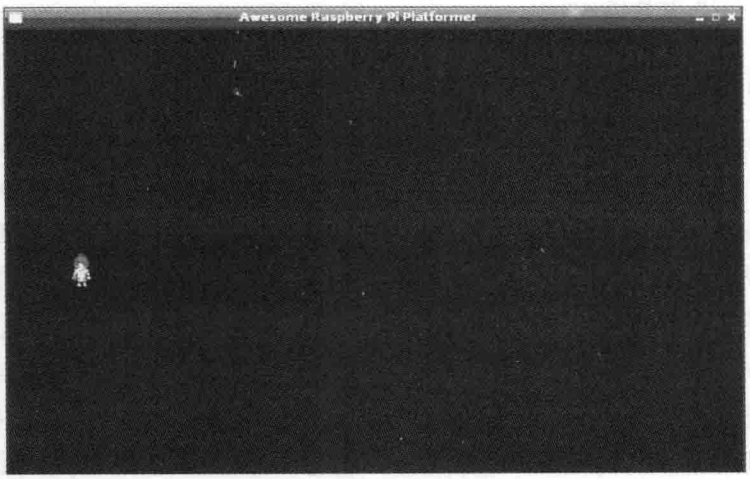


图 5-1 平台游戏简单开头

绘制出角色后，下一步就是让她动起来。这里只是一个简单的测试代码以保证动画可以正常工作。稍后你将会更新代码来使你的角色跳跃起来。将下面方法添加到 Player 类中：

```
def move_y(self):
    self.rect.y = self.rect.y + 1
```

它将把角色缓慢下移。为了完成其他效果，你需要在游戏循环中的合适位置添加下列代码：

```
#blank screen
screen.fill((0, 0, 0))

#move the player with gravity
player.move_y()
```

现在运行代码，你将会看到角色向窗口下方移动，直到消失到底端。在构建一个供角色遨游的世界之前，我们只能做这些事情了。

5.3 为角色创建世界

我们希望可以用尽可能简单的方法来扩展这个游戏，让它看起来更酷。因此我们希望能简单地设计出新的关卡。为了达到要求，我们将每个关卡定义成一个字符串列表。每个字符串对应于屏幕上的一行，字符串中的每个字符代表该行上的一个块。短线 - 表示这里有个平台，字母 G 表示这里是目标（即角色为了完成关卡需要抵达的目标），其他表示空白。将下列代码加到 #options 之后来创建一个基本关卡：

```
level=[
 "                                ",
 "                                ",
 "                                ",
 "                                ",
 "                                ",
 "                                ",
 "                                ",
 "           ---                 G",
 "      -- --     ---        ------",
 " -- -              -------       "]]
platform_colour = (100, 100, 100)
goal_colour = (0, 0, 255)
```

如果你的输入正确，每行的宽度都应该一样。这是个用来测试的非常简单的关卡。最后两行分别设置了平台和目标的颜色。正如我们在本章开头看到的矩形区域一样，这里使用 RGB 值表示颜色。

现在添加下列代码到 World 类的 __init__ 方法中，用来加载关卡：

```
def __init__(self, level, block_size,
        colour_platform,
        colour_goals):
self.platforms = []
self.goals = []
self.posn_y = 0
self.colour = colour_platform
self.colour_goals = colour_goals
```

```
        self.block_size = block_size

        for line in level:
            self.posn_x = 0
            for block in line:
                if block == "-":
                    self.platforms.append(pygame.Rect(
                        self.posn_x, self.posn_y,
                        block_size, block_size))
                if block == "G":
                    self.goals.append(pygame.Rect(
                        self.posn_x, self.posn_y,
                        block_size, block_size))
                self.posn_x = self.posn_x + block_size
            self.posn_y = self.posn_y + block_size
```

代码非常简单，先循环每一行，再循环行内的每一个字符，根据发现的字符构建相应的矩形。在使用这些方块之前，你需要先为 World 类添加 update 方法，该方法可以将方块绘制到屏幕上。

```
    def update(self, screen):
            '''draw all the rectangles onto the screen'''

            for block in self.platforms:
                pygame.draw.rect(screen, self.colour, block, 0)
            for block in self.goals:
                pygame.draw.rect(screen, self.colour_goals, block, 0)
```

现在你需要添加下列代码来创建对象并刷新它。和以前一样，只要找到注释，把它们放到正确的地方就可以了。

```
#initialise variables
world = World(level, 30, platform_colour, goal_colour)

#render the frame
world.update(screen)
```

运行程序，你会发现没有什么可以玩。游戏的世界已经绘出，但角色还是越过平台，不断向下缓慢移动，直到在屏幕上消失。

5.3.1 检测冲突

幸运的是，PyGame 的 Rect 的方法 colliderect() 能够非常简单地让两个游戏元素相互影响。它简单地不可思议，使用格式如下：

```
rect1.colliderect(rect2)
```

这里 rect1 和 rect2 是两个矩形。如果这两个矩形发生重叠，该语句将返回 True，否则

返回 False。你可以用它来检测游戏角色是否和游戏世界重叠，这样角色就不会穿过平台继续下落。从 World 类开始，添加如下代码：

```
def collided_get_y(self, player_rect):
        '''get the y value of the platform the player is
        currently on'''
        return_y = -1
        for block in self.platforms:
            if block.colliderect(player_rect):
                return_y = block.y - block.height + 1
        return return_y
```

这段代码不仅检查游戏角色是否接触到了游戏世界，还可以返回游戏角色碰到游戏世界中的矩形的上边的纵坐标，返回值 -1 表示没有两者没有接触。下一步我们来更新 Player 类，使其能在合适的时候运动或者静止。

```
def move_y(self):
    '''this calculates the y-axis movement for the player
    in the current speed'''
    collided_y = world.collided_get_y(self.base)
    if self.speed_y <= 0 or collided_y < 0:
        self.rect.y = self.rect.y + self.speed_y
        self.speed_y = self.speed_y + gravity
    if collided_y > 0 and self.speed_y > 0:
        self.rect.y = collided_y
    self.base.y = self.rect.y+self.rect.height
```

为了让游戏角色下落的自然点，你还需要添加一个东西：

```
#options
gravity = 1
```

现在运行程序，如图 5-2 所示，游戏角色会向下降落到平台上。

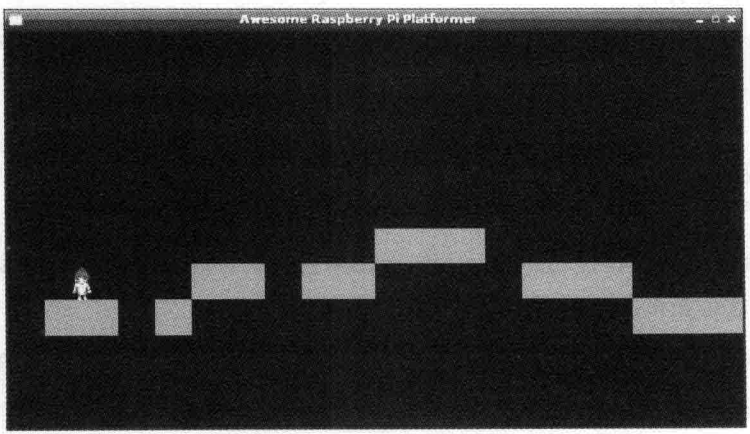


图 5-2 英雄现在可以站在我们为之创造的世界上

让我们来看下 move_y() 是如何使这一切变成现实的。代码处理了两种情况，每种情况都有自己的 if 块。第一种情况是游戏角色可以根据其当前速度和受到的重力自由地上下移动，即它没有碰到游戏世界的任何一个部分（也就是 collided_get_y 返回 -1）。我们也希望游戏角色可以从一个平台跳到更高的平台，因此如果游戏角色当前是向上移动的（也就是 if self.speed_y <= 0），我们像角色没有碰到游戏世界的情况一样处理它。如果发生这两种情况中的任意一种，我们就运行下列代码：

```
self.rect.y = self.rect.y + self.speed_y
self.speed_y = self.speed_y + gravity
```

这段代码根据当前速度将对象移动一个距离，然后根据重力更新当前速度（重力是一个加速度，它模仿了真实世界的物理现象）。

第二种情况（由第二个 if 语句处理）指的是角色开始接触到平台并且不再移动。在这种情况下，程序会调整角色坐标以使之刚好接触到平台。调整坐标是因为如果角色每帧移动多余一个像素，就有可能在接触平台后移动了若干像素才被检测到。随后你会发现当角色从高处下落时，有时候会出现角色向下越过平台一点然后又移动到平台边缘的情况，这也是模仿了人类从高处跳下时恢复站立的过程。

5.3.2 左右移动

现在游戏看上去像点样子了。有角色，有世界，但是角色还不能畅游世界。下一步要给它添加动作。有两种类型的动作。首先我们要让角色在接触到平台后能够跳跃，其次角色还要能够左右移动。

先为 Player 类添加 jump() 方法。因为你已经能够让角色落下，跳跃也一样简单。你要做的就是确保她在地面上，然后设置它向上移动并让其自行落下。

```
def jump(self, speed):
    if world.collided_get_y(self.base)>0:
        self.speed_y = speed
```

现在开始让角色左右移动。实际上如果让角色保持不动，只是左右移动世界会很简单。这样可以确保角色不会因为移动而从屏幕上消失。

现在你要做的是遍历世界中的所有矩形（包括平台和目标），并按照给定的偏移量移动它们。我们可以通过更新矩形的 x 和 y 坐标来完成这点。Rect 类中有两种方法可供使用：move(x-distance, y-distance) 和 move_ip(x-distance, y-distance)。move() 返回一个移动了给

定距离的相同的新矩形，move_ip() 只是移动当前矩形的坐标（ip 表示就地（in place））。因为不需要每次移动时创建新矩形，我们选择 move_ip()。代码如下：

```
def move(self, dist):
    for block in self.platforms + self.goals:
        block.move_ip(dist, 0)
```

现在剩下的事情就是在主循环中添加代码，当按键按下时能够运行合适的方法。PyGame 为此提供了两种方法。第一种方法是监听按键事件，然后检测哪个键按下并根据检测结果执行相应动作。当你只需要关心按键动作时可以选择这种方法。第二种方法是使用 pygame.key.get_pressed() 返回一个按键事件队列。[⊖]如果键按下队列中相应键的值为真，否则为假。如果你希望游戏玩家可以按住向下键不动使角色持续移动，显然是第二种方法更好。我们希望用户能够持续按下方向键，因此请在游戏循环中的适当位置加入下列代码：

```
#check which keys are held
    key_state = pygame.key.get_pressed()
    if key_state[K_LEFT]:
        world.move(2)
    elif key_state[K_RIGHT]:
        world.move(-2)
    if key_state[K_SPACE]:
        player.jump(jump_speed)
```

注意，K_LEFT、K_RIGHT 和 K_SPACE 都是我们从 pygame.locals 中导入的常量，此外还有从 K_a 到 K_z 的字母键。

下面添加一个选项表示跳跃速度（负数是因为 PyGame 的坐标系统是从左上角开始的）：

```
#options
jump_speed = -10
```

现在运行代码，你会将会看到一个所谓的平台游戏基础（见图 5-3）。你可以移动角色，跳跃缺口。然而这里还缺少两个重要的部分。首先，现在没有办法完成关卡，其次，没有任何东西可以让你停下来。

⊖ 这个队列是一个事件的列表——当事件发生时就会被加到队列中，如果被读过了，它们就会被删除掉。——译者注

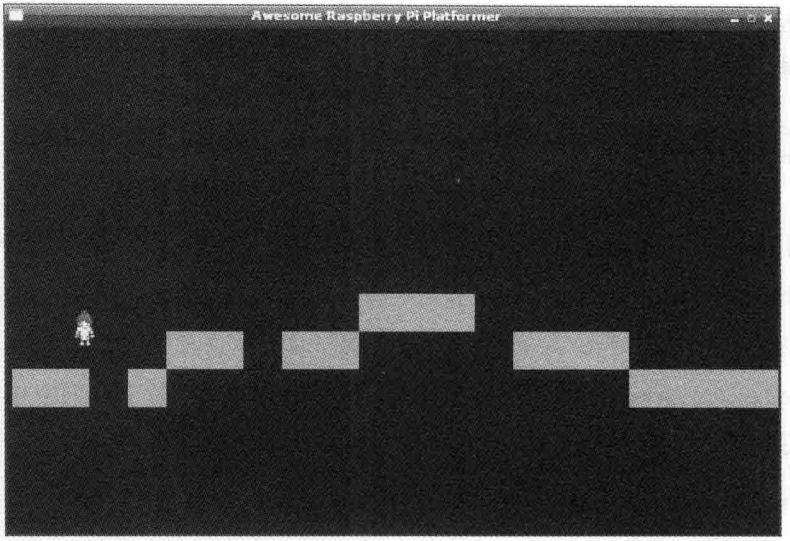


图 5-3 在树莓派上运行的一个不超过两百行的由 Python 写的平台游戏基础

5.3.3 达到目标

让我们先来完善第一个缺陷。一半是因为它比较简单，一半是因为完成之后你就有一个可供测试的游戏了。由于代码已经可以按照我们的意愿创建、显示并移动目标，我们需要做的就是检测角色是否到达目标。我们分两步来做。首先在 World 类中添加下列方法：

```
def at_goal(self, player_rect):
    for block in self.goals:
        if block.colliderect(player_rect):
            return True
    return False
```

该方法和我们之前创建的 collide_get_y() 方法一样，只是它的返回值是 True 和 False。之后，你需要在游戏循环中运行该方法，因此，加入下列代码：

```
#check if the player has completed the level
  if world.at_goal(player.rect):
    print("Winner!")
    finished = True
```

现在运行程序，你将可以跑动、跳跃，直到抵达目标并完成关卡。你可也能从平台上掉下来，消失在深渊里。但这还不够有挑战性。

5.3.4 制造挑战

为了给游戏增加剧情，我们加入一个 Doom 类，其中保存了可以杀死角色的道具。在本游戏中，玩家需要面临两个挑战。首先，在屏幕下方会有毁灭之火，如果角色落到上面就会被消灭。其次，从游戏玩法角度看也是更重要的，会有火球从天空落下。角色必须避开它们，因为这些火球挡在通往目标的路上。

首先，燃起毁灭之火。我们把它们绘制在屏幕底端的矩形区域。在 Doom 类中添加下列代码：

```
def __init__(self, fireball_num, pit_depth, colour):
    self.base = pygame.Rect(0, screen_y-pit_depth,
                screen_x, pit_depth)
    self.colour = colour

def collided(self, player_rect):
    return self.base.colliderect(player_rect)

def update(self, screen):
    '''move fireballs down, and draw everything on the screen'''
    pygame.draw.rect(screen, self.colour, self.base, 0)
```

在选项、变量和游戏循环的适当位置添加下列代码：

```
#options
doom_colour = (255, 0, 0)
#initialise variables
doom = Doom(0, 10, doom_colour)
    #render the frame
    doom.update(screen)
    #check if the player is dead
    if doom.collided(player.rect):
        print("You Lose!")
        finished = True
```

这是一段很好的自注释代码。注意，你不需要移动毁灭之火的矩形区域，它将一直停留在屏幕下方。调用 update() 方法是为了移动火球。现在让我们看一下结果。

这里我们添加完整的 Fireball 类：

```
class Fireball(pygame.sprite.Sprite):
   '''this class holds the fireballs that fall from the sky'''
   def __init__(self):
      pygame.sprite.Sprite.__init__(self)
      self.image = pygame.transform.scale(
                   pygame.image.load(fireball_image),
                   (fireball_size, fireball_size))
      self.rect = self.image.get_rect()
      self.reset()
```

```
    def reset(self):
        self.y = 0
        self.speed_y = randint(fireball_low_speed,
                               fireball_high_speed)
        self.x = randint(0,screen_x)
        self.rect.topleft = self.x, self.y

    def move_x(self, dist):
        self.rect.move_ip(dist, 0)
        if self.rect.x < -50 or self.rect.x > screen_x:
            self.reset()

    def move_y(self):
        self.rect.move_ip(0, self.speed_y)
        if self.rect.y > screen_y:
            self.reset()
```

正如你所看到的，该类和 Player 类一样继承了 Sprite 类。move_x() 方法的工作方式和 World 类中相应的函数也一样。所不同的是这里只移动一个火球，因为每个火球都是一个独立对象。

为了保持挑战性，火球需要不断地从空中落下。有很多方法可以做到这一点，这里我们选择创建固定数量的火球，当火球在屏幕上消失后就立即复位它。reset() 方法将火球放到屏幕上方的随机位置并赋予火球一个随机的速度。

randint(a, b) 返回一个在 a 到 b 之间（包含 a 和 b）的随机整数。变量 screen_x 用来确保火球在屏幕上。两个全局变量（fireball_low_speed 和 fireball_high_speed）用于指定火球的最小速度与最大速度。它们的值表示每帧移动的像素数。这里你不需要冲突检测，因为这些都放在 Doom 类中做了。

现在来更新 Doom 类：

```
class Doom():
    '''this class holds all the things that can kill the player'''
    def __init__(self, fireball_num, pit_depth, colour):
        self.base = pygame.Rect(0, screen_y-pit_depth,
                                screen_x, pit_depth)
        self.colour = colour
        self.fireballs = []
        for i in range(0,fireball_num):
            self.fireballs.append(Fireball())
        self.fireball_plain = pygame.sprite.RenderPlain(
                                self.fireballs)

    def move(self, dist):
        for fireball in self.fireballs:
            fireball.move_x(dist)
```

```
    def update(self, screen):
        for fireball in self.fireballs:
            fireball.move_y()
        self.fireball_plain.draw(screen)
        pygame.draw.rect(screen, self.colour, self.base, 0)

    def collided(self, player_rect):
        for fireball in self.fireballs:
            if fireball.rect.colliderect(player_rect):
                hit_box = fireball.rect.inflate(
                          -int(fireball_size/2),
                          -int(fireball_size/2))
                if hit_box.colliderect(player_rect):
                    return True
        return self.base.colliderect(player_rect)
```

如你所见，这里创建了一个火球列表，并将它们添加到 fireball_plain 中。和 player_plain 一样，这也是一个 RenderPlain，允许你把火球绘制到屏幕上。注意这里使用全局变量表示火球的大小和数目。改变它们可以急速地影响游戏性，并且在许多方面它都是改变难度的关键变量。

collided() 方法和我们之前见到的有点不同。它比较角色和火球体积一半的矩形是否有接触。这是因为无论角色还是火球都不是一个规则矩形，在两个对象还有距离的情况下这两个矩形可能会接触到。这会极大地挫伤游戏玩家的积极性。我们使用的并不是最好的方法，但却是朝着降低角色阵亡的方向努力。换句话说，角色有可能擦到火球边缘而侥幸逃脱，但如果碰撞检测返回 True，就意味着真的发生了碰撞。



注意 实际上使用 PyGame 提供的 pygame.sprite.collide_mask(sprite1, sprite2) 可以精确地检测到精灵碰撞。但这需要消耗更多的计算资源，并且对于我们这个任务会显得过度杀伤精灵。

添加这两个类之后，游戏只需要添加下列代码就可以工作了：

```
#options
fireball_size = 30
fireball_number = 10
fireball_low_speed = 3
fireball_high_speed = 7
fireball_image = "flame.png"
```

在 Doom 的初始化中加入火球：

```
doom = Doom(fireball_number, 10, doom_colour)
```

你还需要在 keypress 小节加入几行代码使得火球会随着背景移动（只需要添加黑体部分）：

```
#check which keys are held
  key_state = pygame.key.get_pressed()
  if key_state[K_LEFT]:
      world.move(2)
      doom.move(2)
  elif key_state[K_RIGHT]:
      world.move(-2)
      doom.move(-2)
```

我们仍然使用从 http://opengameart.org 中下载的精灵图像。这里我们使用了：http://opengameart.org/sites/default/files/flame.png。为了使其能够工作，文件需要下载并保存到你运行程序的那个目录下。或者，你也可以使用精灵的绝对路径。例如，如果你把所有东西都存在 /home/pi/my_game/ 目录下，你可以将下面这行

```
fireball_image = "flame.png"
```

改为

```
fireball_image = "/home/pi/my_game/flame.png"
```

这样无论你在哪个目录下运行游戏，它都可以工作。现在保存程序并运行，游戏看起来应该和图 5-4 一样。

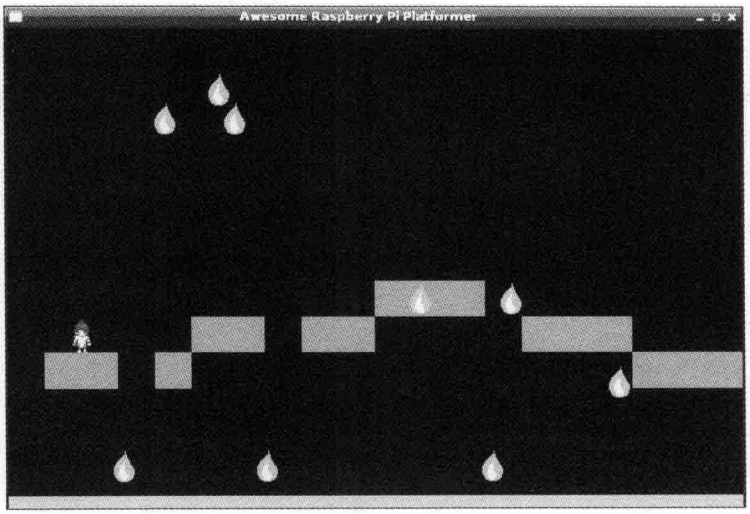


图 5-4　一个边缘略显粗糙但可以工作的平台游戏

5.4 在游戏中加入自己的风格

我们已经揭开游戏的运行机制。玩家需要穿越世界、躲避火球，并最终到达目标。虽然还有需要完善的地方，但基础已经做好。现在是时候添加些自己的风格了。毕竟，本章不是为了教你如何复制代码来创建游戏，而是为了让你能够构建自己的游戏。现在你对于如何定制游戏了解的已经够多了。从选项（option）部分改起是个不错的主意。

根据你的显示器，你可能想改变窗口大小。如果你觉得这个太简单，可以添加更多的火球，或者让火球变大。也许，你希望能跳得更高，跑得更快。所有的这些都很简单。实际上，从前面的章节中你应该学到如何在游戏开始时添加一个游戏菜单。在 #initialise pygame 之前，你可以做一个简单的基于文本的菜单，用来选择关卡难度。更难一点的，你可以……事实上，你只需要自己来做。如果你更有雄心，可以做一个图形化菜单替代文本菜单。

5.5 添加音乐

但愿你已经做完了自己的定制版游戏，如果没有完成也不用担心，等本章剩余部分结束后你可以继续进行。

现在需要添加点耀眼的东西。这部分不会影响游戏的运行机制，却可以增加娱乐性。首先添加声音效果，其次是添加背景。

开始添加音乐前，我们需要初始化合成器。它会将声音设备安装好以准备播放。这些都由下列代码完成：

```
#initialise pygame.mixer
pygame.mixer.pre_init(44100, -16, 8, 2048)
pygame.mixer.init()
```

虽然开始的时候我们只播放跳跃音效，但是这段允许我们同时播放 8 种声音。同样，我们访问 http://opengameart.org。这次文件来自 http://opengameart.org/sites/default/files/qubodup-cfork-ccby3-jump.ogg。你需要下载它或者其他音乐文件。我们将该文件添加到选项中：

```
#options
jump_sound = "qubodup-cfork-ccby3-jump.ogg"
```



MP3 文件有时候也可以工作，但比较难对付。有可能导致 PyGame 游戏崩溃。因此可能还是尽量选择 OGG 文件。

下面我们需要修改 Player 类使其能够在适当时候播放音乐。在 __init__() 方法的末尾加入下列代码：

```
self.sound = pygame.mixer.Sound(jump_sound)
```

你还需要修改 jump() 方法：

```
def jump(self, speed):
    if world.collided_get_y(self.base)>0:
        self.speed_y = speed
        self.sound.play()
```

这就是你要给游戏添加的音效。你会发现可以很容易地给游戏添加更多音效。例如当抵达目标时播放一段音乐，或者角色挂掉时候播放一段音乐。你也可以播放背景音乐。然而，要注意你买的这些音乐都是有版权的。在自己的游戏中使用它们没有问题。但是如果要发布自己的游戏，就可能会带来麻烦。可以看看这个网站 http://freemusicarchive.org。和 http://opengameart.org 一样，该网站上有大量文件可供下载，你也可以将它们放到你的游戏中。其中许多文件都是得到许可的，如果你要发布自己的游戏，需要同时发布自己的源代码。

5.6 添加布景

我们为游戏添加的第二个细节是背景。当然，你需要一个背景图片。我们已经从网站 http://opengameart.org/sites/default/files/background.zip 中选取了文件 background.png。这个是很好的乡村景色，你也可以换成一些灰暗的图片来改变游戏氛围。为了加载文件你需要在选项中添加下列代码：

```
#options
background_image = "background.png"
```

使用图片前，你需要一个继承自 Sprite 类的新类来显示它（像 Player 和 Fireball 一样）。由于不需要操作矩形区域或者检测碰撞，你可以简单地为其加载一个图像。详见下面代码：

```
#set up the background
background = pygame.transform.scale(pygame.image.load(
             background_image), (screen_x, screen_y)).convert()
bg_1_x = -100
bg_2_x = screen_x - 100
```

第一行加载图片并将其拉伸到整个屏幕，然后运行 counert() 方法。该方法将图片从 PNG 转换成 PyGame 的外观。这会让屏幕显示更快，尤其是对于这个尺寸的图像很重要。

这里你也可能使用了其他图像，但你会失去图像的透明部分，使其变成完全的矩形。第二行和第三行创建两个变量来存储图像的 x 坐标。这里之所以有两个值是因为我们要绘制这个图像两次，用来创建一个连续循环的背景，这样玩家就不会移动到图像末尾了。

为了移动背景，需要在游戏循环中的适当位置添加下列代码：

```
#check which keys are held
  key_state = pygame.key.get_pressed()
  if key_state[K_LEFT]:
     world.move(2)
     doom.move(2)
bg_1_x = bg_1_x + 1
     bg_2_x = bg_2_x + 1
     if bg_1_x < screen_x:
         bg_1_x = -screen_x
     if bg_2_x < screen_x:
         bg_2_x = -screen_x
  elif key_state[K_RIGHT]:
     world.move(-2)
     doom.move(-2)
     bg_1_x = bg_1_x - 1
     bg_2_x = bg_2_x - 1
     if bg_1_x > -screen_x:
         bg_1_x = screen_x
     if bg_2_x > -screen_x:
         bg_2_x = screen_x
```

这里还有一些故事。首先，你是否注意到背景比 world 或者 doom 移动地慢？这个称为视差滚动（parallax scrolling）。它创建了多层背景使其以不同的速度移动，形成立体的运动效果。就像你坐在运动的车上看窗外，离你近的物体比离你远的物体看起来移动的速度快。这不是 3D 图像，而是帮助创造一种层次错落的错觉。如果你想做得更好一点，可以添加更多的背景层次。比如，可以画一些树，让它们比平台移动地慢一点，然后添加一些山，使其比树移动地慢点，最后添加个太阳，让它移动地最慢。和声音一样，你可以随便添加自己喜欢的东西进去。

第二个值得注意的事情是代码中用于移动背景的 if 语句。当图像的一端移动到屏幕边缘时，程序将其另一边重新移到屏幕另一侧。这样就创造了在两个背景图像中不断循环的无限滚动背景。剩下的事情就是将图像绘制到屏幕上：

```
#render the frame
screen.blit(background,(bg_1_x, 0))
screen.blit(background,(bg_2_x, 0))
```

这两行必须紧跟着 #render the frame，否则它们会遮住其他要显示的部分。图 5-5 给出了游戏的最终结果。

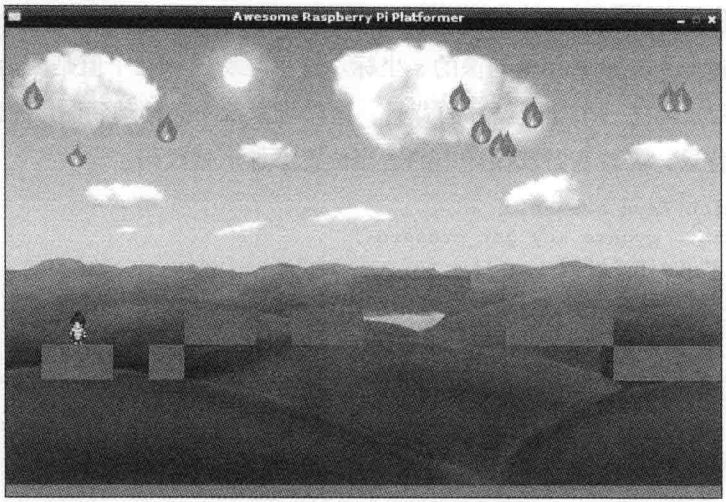


图 5-5 添加了所有元素的游戏

添加点睛之笔

游戏的实质部分已经全部完成。然而，我们还可以做一些事情让人们能够更方便地使用他。之前，我们用到了关卡（level），它被硬编码到程序中。如果我们可以将其放在文本文件中，允许用户选择加载哪个文件来选择关卡，然后程序加载相应文件，这样会比以前更好一点。由于关卡是文本形式，我们的工作就更简单了。

运行 python3 chapter5-platformer.py（或者你自己命名的文件）将运行程序内置的默认关卡。而运行 python3 chapter5-platformer.py mylevel 将运行加载了指定文件 mylevel 的游戏。为了实现这一点，我们需要使用 sys.argv。它属于 sys 模块，它是一个包含了所有参数的列表，并能将其传递给 Python。sys.argv[0] 是我们运行的脚本名字，因此 sys.argv[1] 就是后面跟的文件名（如果存在）。我们需要做的就是把下面代码加到程序中：

```
#load level
if len(sys.argv) > 1:
    with open(sys.argv[1]) as f:
        level = f.readlines()
```

读取文件和我们之前读取数组一样。即 - 表示平台，G 表示目标，多行一起构成不同的游戏关卡。



注意 如果你没有跟着一步步做下去，可以从网站上下载完整游戏：chapter5-platformer.py。我们强烈建议你随着本章一起完成该游戏，它可以帮助你更好地理解这些内容。

5.7 让游戏更上一层楼

我们可以继续为游戏添加更多内容。然而，我们的指导将到此为止。不是因为游戏完成了，而是由于你现在应该学习到足够的知识，可以独立完成它了。我们不会命令你要把游戏做成什么样子——它是你的游戏，想加什么都可以。这里我们给出一些改进的想法：

- 如果你还没有尝试过修改过游戏选项，试着修改一下。
- 为游戏创建新关卡会让你感觉到这是你自己的游戏。
- 游戏中使用的图片仅作为推荐，自己上网找一下，尝试替换一下它们。
- 为当前的矩形对象添加精灵，如平台和末日之火。
- 将游戏关卡融入游戏世界。每个世界都有一个不同的主题，为每个主题选择一个不同的图片、音乐。
- 为角色添加道具。可以是统计成绩的金币，或者增强道具使得角色可以跑得更快或者跳得更高。
- 添加更多可以消灭玩家的怪物。如某些持续向右移动的东西，使得玩家必须保持运动以避开它们，或者某些冲向玩家的怪物，玩家必须踩上面才能消灭它们，或者从下面射上去的东西。
- 每一关卡结束时统计成绩。成绩可以是完成时间，玩家收集到的目标（金币），或者其他一些东西。
- 让精灵动起来。http://opengameart.org 上有许多动作的连续图像，通过不断地在一组图像中切换，可以使精灵动起来。
- 如果你一直按下方向键，精灵会加速运动而不是保持恒定的速度，也可以增加一个运行键，运行玩家加速游戏。

这仅仅是一些参考想法。我们并不想也无法列出你可以在游戏中添加的全部内容。因此开动脑筋想想看你还可以在游戏中实现什么内容。如果看起来不错，你可以上传到树莓派商店让其他人试一下。最后请不要忘了你在游戏中使用的任何图片和声音的许可。

5.8 逼真的游戏物理

用来创建简单游戏 PyGame 是一个很好的模块。正如你所见到的，使用它可以很简单地在屏幕上绘制对象并移动它们。但有时你需要更多能量。在前面的例子中，重力影响了角色的下落，但其他的物理现象并没有考虑进去。如果需要对象之间以更真实的方式交

互，如相互弹跳，你需要使用物理库。

PyMunk 就是这样一种库，它可以帮助你创建更真实的游戏。使用它，你可以创建一个空间并加入对象。PyMunk 将会解决它们之间的交互。

可以从 http://code.google.com/p/pymunk/downloads/list 上下载 PyMunk（你需要源程序）。如下所示，下载完成后，解压缩后进入新目录（使用 LXTerminal 而不是 Python 运行下列命令）：

```
unzip pymunk-4.0.0.zip
cd pymunk-4.0.0
```

不幸的是，build 文件中有一些小错误使得其不能在树莓派上正确构建。为了安装它，你需要使用文本编辑器（如 LeafPad）打开 steup.py，找到下面这行：

```
elif arch == 32 and platform.system() == 'Linux':
  compiler_preargs += ['-m32', '-O3']]
```

请确保这行包含的是 arch==32 而不是 64。在第二行中删除 '-m32' 使其变成这样：

```
compiler_preargs += ['-O3']]
```

保存文件，现在你可以按照下列方法安装 PyMunk：

```
python3 setup.py build_chipmunk
python3 setup.py build
python3 setup.py install
```

同样，这些命令需要在 LXTerminal 中的 PyMunk 目录下执行。该过程可能需要一段时间，完成之后，你可以打开 Python 并输入下列命令来检测它是否工作：

```
>>> import pymunk
```

希望你没有遇到任何错误

下面的例子可以在网站上的 chapter5-pymunk.py 中找到。

```
import pygame, pymunk
from pygame.locals import *
from pygame.color import *
from pymunk import Vec2d
import math, sys, random

def to_pygame(position):
    return int(position.x), int(-position.y+screen_y)

def line_to_pygame(line):
    body = line.body
    point_1 = body.position + line.a.rotated(body.angle)
    point_2 = body.position + line.b.rotated(body.angle)
    return to_pygame(point_1), to_pygame(point_2)
```

```
###options####
screen_x = 600
screen_y = 400
num_balls = 10
pygame.init()
screen = pygame.display.set_mode((screen_x, screen_y))
clock = pygame.time.Clock()
running = True

space = pymunk.Space()
space.gravity = (0.0, -200.0)

#create the base segment
base = pymunk.Segment(pymunk.Body(),(0, 50), (screen_x, 0), 0)
base.elasticity = 0.90
space.add(base)

#create the spinner
spinner_points = [(0, 0), (100, -50), (-100, -50)]
spinner_body = pymunk.Body(100000, 100000)
spinner_body.position = 300, 200
spinner_shape = pymunk.Poly(spinner_body, spinner_points)
spinner_shape.elasticity = 0.5
spinner_joint_body = pymunk.Body()
spinner_joint_body.position = spinner_body.position
joint = pymunk.PinJoint(spinner_body, spinner_joint_body, (0, 0),
                        (0, 0))
space.add(joint, spinner_body, spinner_shape)

#create the balls
balls = []
for i in range(1, num_balls):
    ball_x = int(screen_x/2)
    radius = random.randint(7, 20)
    inertia = pymunk.moment_for_circle(radius, 0, radius, (0, 0))
    body = pymunk.Body(radius, inertia)
    body.position = ball_x, screen_y
    shape = pymunk.Circle(body, radius, (0, 0))
    shape.elasticity = 0.99
    space.add(body, shape)
    balls.append(shape)

while running:
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False
    screen.fill((0, 0, 0))
#draw the ball
    for ball in balls:
        pygame.draw.circle(screen,(100, 100, 100), to_pygame(ball.
  body.position), int(ball.radius), 0)
```

```
#draw the spinner
    points = spinner_shape.get_vertices()
    points.append(points[0])
    pygame_points = []
    for point in points:
        x,y = to_pygame(point)
        pygame_points.append((x, y))
    color = THECOLORS["red"]
    pygame.draw.lines(screen, color, False, pygame_points)

#draw the line
    pygame.draw.lines(screen, THECOLORS["lightgray"], False,
                      line_to_pygame(base))

    space.step(1.0/50.0)
    pygame.display.flip()
    clock.tick(50)
```

如你所见，这段代码使用 PyGame 绘制图形，使用 PyMunk 解决它们如何运动。

PyMunk 运行需要一个 space，下列两行代码演示了如何建立 space：

```
space = pymunk.Space()
space.gravity = (0.0, -200.0)
```

它们从 Space 类创建了一个新对象，并设置了它的重力。我们可以继续创建对象，并把它们加入 space。本例使用了圆环、线段和一组点构成的多边形。（注意，在树莓派上，PyMunk 有一个 bug，你只能在 space 中添加一个单节段。）关节接头可以将多个形状连接成一个整体，并将其一点固定在某个位置。由于重力原因，它可以沿中心点旋转却不会落下。

在主循环中，我们调用了 space.step(1.0/50.0)，告诉 PyMunk 以五十分之一秒的步进移动 space 中的所有对象。

这里有一个小困扰，PyGame 和 PyMunk 使用了不同的坐标系统。之前我们说过，PyGame 的 (0, 0) 点在左上角，而 PyMunk 的却在左下角。这意味着为了在正确的位置绘制对象，你需要重新计算坐标。这就是 _pygame(position) 要做的事情。

和定义不同对象的位置一样，在 PyMunk 中，你可以给它们定义不同的物理属性，如弹性和惯性。通过调整这些属性，你可以控制对象在你的世界中如何交互。

运行代码，你将看到 PyMunk 解决了如何移动的大部分难题，如处理碰撞和计算球如何从地板上弹开等其他问题。结果如图 5-6 所示。然而这里有个代价问题——它需要的计算量远远超过我们之前的游戏。有时树莓派可以处理简单的物理模拟，你需要注意节约使用，否则它们会运行地很慢。使用 raspi-config 工具，可以超频树莓派，可以帮助模拟程序更快运行。

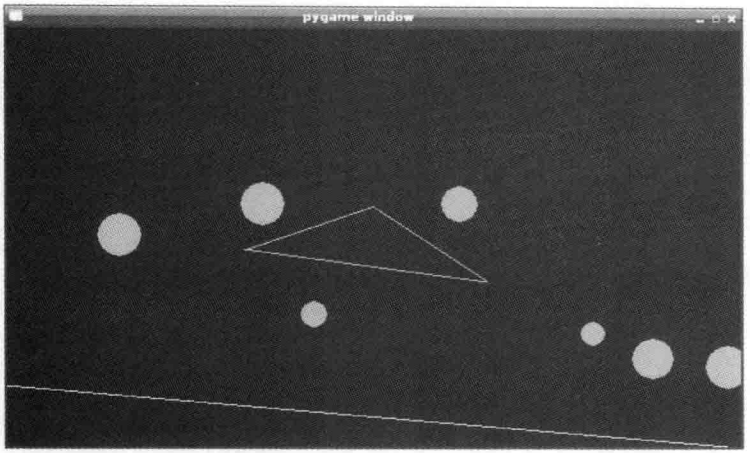


图 5-6 PyMunk 物理模拟引擎完成了繁重的真实世界交互模拟



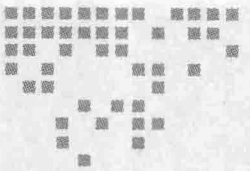
树莓派默认运行频率是 700MHz。也就意味着它可以每秒钟处理 7000,000,000 条指令。对于需要大量计算的任务，你可以使用 raspi-config 工具超频 CPU。它可以达到 1000MHz（或者 1GHz）。然而，并不是所有的树莓派超频后都可以正常工作，有些可能会不稳定。如果你的树莓派启动时候停住了，试着减少超频。

我们仅仅接触了些基础，希望它们足够为你启蒙。在你之前下载的 PyMunk ZIP 文件中还有一些例子。并不是所有例子都可以在 Python3 下面运行，但它们可以给你一个尝试的机会。这里还有些过时的但却很不错的文档 http://pymunk.googlecode.com/svn/tags/pymunk-2.0.0/docs/api/index.html。

5.9 小结

读完本章，你需要了解以下内容：

- PyGame 是一个帮助你在 PyGhon 中创建游戏的模块。
- 继承自 Pygame.sprite.Sprite 的类可以在屏幕上绘制图像。
- 精灵被绘制在类的 self.rect 的矩形内。
- 你可以使用这些矩形来检测对象间的冲突。
- 视差滚动可用于使 2D 图形形成立体运动效果。
- PyGame 也可以处理音频。
- 为了更真实地模拟物理世界，可以使用物理引擎如 PyMunk，但它会使运行速度变慢。



Chapter 6

第 6 章

使用 OpenGL 创建图形

没有人否认 3D 图形看起来很酷。它们能创造出比 2D 图形更多的深度感，可以让程序员创建出更丰富的世界。然而，随之而来的是费用。首先，相比 2D 图形，它需要更多的计算资源，其次，它的编程显然更复杂。

普通 PC 上你通常会发现显卡（有时也会称为图形处理单元，或 GPU）。它提供了额外的计算资源使得计算机可以显示复杂 3D 场景。基本上，它就是添加了许多可以快速处理浮点运算的处理器。快速扫一眼树莓派你就会发现由于它和 PC 布局不同，没有空间可以添加 GPU。它的主板包括了处理器、内存、扩展槽，一切东西都集成到一个片上系统（SoC）中。这就是树莓派中间的最大的那块芯片。如果你从侧面仔细看，会发现它由两层组成，上面一层是 RAM，下面一层用于处理。

底层并不只是 CPU（中央处理器单元）。事实上，CPU 只是它的一小部分。它还包含一个远比 CPU 强大的 GPU。当你运行普通程序时，GPU 处于空闲状态，CPU 处理所有工作。但处理 3D 图形时，CPU 本身不能够处理它，因此你需要将工作分一部分给 GPU。在本章中，我们将看到如何利用 GPU 来使用 OpenGL（GL 代表图形库，Graphics Library）创建 3D 场景。

在开始前我们需要诚实地告诉你：这是本书中最复杂的一章。这是不争的事实。使用 OpenGL 需要一些数学知识、一个新的编程语言和一些新概念。我们将放慢脚步，解释清楚我们遇见的所有事情。然而，如果你只是为了学习在 3D 世界中简单地画个魔方，可以直接跳到第 8 章。

6.1　获取模块

你需要两个模块：PyGame 和 RPiGL。如果你还没有安装 PyGame，请参阅第 4 章中的相应命令。RPiGL 可以从下面地址获取 https://github.com/stephanh42/rpigl。使用下载 Zip 按钮下载压缩包，然后按照下列命令（在 LXTerminal 中）解压缩，并安装它。

```
unzip rpigl-master.zip
cd rpigl-master
python3 setup.py build
sudo python3 setup.py install
```

如果一切顺利，你就可以运行 demo 程序了，尝试下列命令：

```
cd demos
python3 bumpedspere.py
```

你将会看到一个凹凸不平的球体在旋转（是的，这个文件名上有个拼写错误）。如果看到出错信息，在继续本章内容之前，你需要修正它们。

6.2　创建旋转立方体

旋转立方体是一个基本图形，所有的新 3D 程序员都需要尝试下它。它简单易懂，也涵盖了所有的基础，并且也不需要大量的数据来建立 3D 模型。

再次提醒本书的网站是 www.wiley.com/go/python-raspberrypi. 为了避免潜在的拼写错误，你可以下载、复制和粘贴代码到你的 IDE 或者代码编辑器中。

完整程序位于网站上的 chapter6-spinning-cube.py 中。由于篇幅太长，我们来分段阅读它。

首先你需要建立数据：

```
vertices = [(0.0,0.0,0.0), (0.5,0.0,0.0), (0.5,0.5,0.0),
            (0.0, 0.5,0.0), 0.0,0.0,-0.5), (0.5,0.0,-0.5),
            (0.5,0.5,-0.5), (0.0, 0.5,-0.5)]


indices_face_1 = (0, 1, 2, 0, 3)
```

我们要绘制的立方体的所有顶点保存在 vertices 列表中。实际上你绘制的不止一个旋转立方体。还包括旋转立方体的六个表面中的四个、静态立方体的边和一些点。做完这些，你将学到在屏幕上绘画的其他方法。

元组 indices_face_1 中保存了一组点，你将使用它们在屏幕上绘制特定的形状（其中的数值是列表 vertices 中的下标）。这些下标表示了立方体的一个表面，在实际代码中还有更多表面。我们随后会使用这些表面从同一个顶点池中绘制多个形状。

下一步是建立 OpenGL 环境（注意，如果你在跟着下载的文件往下看，这里我们不再按顺序进行）。

```
self.vertex_shader = glesutils.VertexShader(vertex_glsl)
self.fragment_shader = glesutils.FragmentShader(
                       fragment_glsl)

self.program1 = glesutils.Program(self.vertex_shader,
                                  self.fragment_shader)
self.program1.use()

glClearDepthf(1.0)
glDepthFunc(GL_LESS)
glEnable(GL_DEPTH_TEST)

glClearColor(0.5, 0.5, 0.5, 1)
```

之前，我们说过 GPU 是一个额外的处理器，需要使用 3D 图形时你可以用它来做数学计算。要使用它，我们需要对其编程，编程就需要源代码。这些程序由顶点着色和片段着色两部分构成。代码保存在变量 vertex_glsl 和 fragment_glsl 中，这叫做程序中内嵌程序（稍微我们会详细介绍）。为了使用它们，你需要把它们转化为着色器对象，然后将着色器对象和一个可运行程序绑定在一起。在你的 Python 程序中可以有超过一个的 OpenGL 程序，你可以使用 use() 方法来切换它们。这里只有一个程序，因此在开始的地方只调用了一次 use()。

最后的四行用来设置 OpenGL。首先，前三行指定清除缓冲区的值为 1。最后一行清除颜色缓冲区，将屏幕设置成中等灰度。OpenGL 颜色的四个值分别是红、蓝、绿和 Alpha（透明度）。每个值的范围都是 0 到 1。

下一个任务是将数据加载到 GPU 中。

```
self.verteces_buffer = array_spec.create_buffer(
                       vertex_attrib=vertices)

self.elements_face_1 = glesutils.ElementBuffer(
                       indices_face_1)
```

主程序运行在 CPU 上，3D 建模由 GPU 完成。在树莓派中这两者距离很近，但是数据在两个单元之间传输也需要一些时间。由于这一点，在开始运行之前最好将尽可能多的信息加载到 GPU 中。这就是缓冲区的作用。这里我们使用两类缓冲区来存储顶点和

下标信息。程序开始运行之后，只需传送你要使用的缓冲区的标记。在这个特殊程序中，每个 buffer 中只存了少量信息，使用和不适用缓冲区的差别并不明显。然而如果你需要加载复杂的 3D 模型，每个缓冲区都需要存储大量信息，这样就可以明显看出传输时间的区别。

6.2.1　向量和矩阵

在我们创建的 3D 世界中，所有的点都被定义成坐标 (x, y, z)。x 表示水平位置，y 表示垂直位置，z 表示深度。用数学名词描述这组数字，就是向量。物体中的每一个点称为顶点。[⊖]每一个顶点都有一个向量来描述它的位置。你需要操作这些点才能在 3D 世界中移动物体。例如，你想放大物体，需要给每一个顶点一个比例因子。或者你想旋转物体，需要相应地移动每个顶点坐标。

这些都可以由向量矩阵代数完成。简单地说，就是对于你想做的每一个变换，都可以创建一个矩阵（纵横排列的二维数据），然后将向量乘以该矩阵得到一个新的向量。

看到乘这个词不用疑惑，这不是普通乘法。其数学过程有点复杂，由于 OpenGL 已经把这些都做好了，你不用再为此烦恼。在这个阶段你需要知道的是，要移动一个物体，需要创建一个矩阵，并将顶点坐标乘以合适的矩阵。如果你想超越本章范围，有很多在线或者印刷资料可供参考。这对于学习更多相关知识非常有用。

下面这段代码创建了两个矩阵，给我们演示了这一过程。transforms.compose() 用来将多个矩阵合并成一个。

```
self.outer_matrix = transforms.compose(
        transforms.rotation_degrees(20, "z"),
        transforms.rotation_degrees(20, "y"),
        transforms.rotation_degrees(20, "x"),
        transforms.scaling(1.2))

self.points_matrix = transforms.compose(
        transforms.stretching(0.1, 1, 1.5),
        transforms.translation(-0.5, -0.5, -0.5))
```

transforms.rotation_degrees()、transforms.scaling() 和 transforms.stretching() 都会返回一个矩阵，分别用于旋转、缩放和拉伸操作时拿来与向量相乘。

在继续绘制屏幕之前，我们回过头来看一下我们加载到 GPU 的顶点和片段着色器的代码：

⊖ 原文如此，可能是作者笔误，这里应该是物体中两条边的交叉点。——译者注

```
array_spec = glesutils.ArraySpec("vertex_attrib:3f")

vertex_glsl = array_spec.glsl() + """
uniform mat4 transform_matrix;
void main(void) {
  gl_Position = transform_matrix * vec4(vertex_attrib, 1.0);
  gl_PointSize = 2.0;
}
"""

fragment_glsl = """
uniform vec4 color;
void main(void) {
  gl_FragColor = color;
}
"""
```

它创建了两个包含代码的字符串：vertex_glsl 和 fragment_glsl。你可能注意到，这并不是 Python 代码。对 GPU 编程需要使用一种称为 GLSL（Graphics Library Shader Language，图形库着色语言）的特殊语言。有点类似于 C 语言（一种可以为主 CPU 编写程序的语言）。GLSL 和 Python 的主要区别在于：

- 每个语句都以分号结尾。
- 缩进级别无关紧要，代码段包围在花括号中。
- 变量有自己的类型，且只能存储这种类型的数据。

关键字和函数也不一样。由于这些都是全新的概念，我们一行一行来解释。

```
array_spec = glesutils.ArraySpec("vertex_attrib:3f")
```

创建了一个新的 ArraySpec 对象（在别处需要用到）。参数表示你将给它传递一个 vertex_attrib 的属性，该属性是一个三维浮点向量（3f）。

```
vertex_glsl = array_spec.glsl() + """
```

创建了一个变量并赋一个字符串给它。array_spec.glsl() 返回一段代码，用于创建 array_spec(vertex_attrib) 中的属性。三个引号告诉 Python，这是一个多行的字符串。

```
uniform mat4 transform_matrix;
```

创建了一个名为 transform_matrix 的 uniform 变量。该变量可以存储 4×4 矩阵。uniform 关键字表示可以从 Python 代码中给它赋值。

```
void main(void) {
```

创建 main 函数，每次程序运行时都会运行该函数。第一个 void 表示它不返回任何值，第二个表示它不需要任何输入参数。注意花括号，它表示代码段的开始。

```
gl_Position = transform_matrix * vec4(vertex_attrib, 1.0);
```

这里的缩进并不是必须的（Python 中却是必须的），这里加入缩进只是为了方便阅读程序。它将表示顶点位置的向量和表示转换的矩阵相乘。注意它们都是 4 维矩阵。现在先不用关心最后一个数值，先把它设为 1.0。

将顶点绘制在屏幕上时，每个顶点 shader 都需要设置 gl_Position 作为自己的变量。设置好之后，OpenGL 会处理好剩下的工作。

```
gl_PointSize = 2.0;
}
"""
```

gl_PointSize 只是简单地设置你绘制在屏幕上的点的大小（我们将涉及点、线和一点三角形）。花括号用来结束 main 函数。三个双引号表示字符串结束。

顶点着色器在场景里的每个顶点将调用一次，片段着色器在场景里的每个像素（顺序）将调用一次。因此片段着色器的运行次数远比顶点着色器多，片段着色器都很简单。本例中，它只有 4 行：

```
fragment_glsl = """
uniform vec4 color;
void main(void) {
  gl_FragColor = color;
}
"""
```

像顶点着色器通常设置 gl_Position 一样，片段着色器通常设置 gl_FragColor。它是一个四维向量表示该位置的颜色。

这些都就绪之后，剩下的事情就是把这些物体放入 3D 世界：

```
#Draw outer lines
self.program1.uniform.transform_matrix.value =
                                self.outer_matrix
self.program1.uniform.color.value = (1, 1, 1, 1)
self.verteces_buffer.draw(elements=self.elements_outer,
                          mode=GL_LINE_STRIP)
#Draw points
self.program1.uniform.transform_matrix.value =
                                self.points_matrix
self.program1.uniform.color.value = (0, 0, 0, 1)
self.verteces_buffer.draw(elements=self.elements_points,
                          mode=GL_POINTS)

#Draw spinning cube
rotation_matrix =
      transforms.compose(
      transforms.rotation_degrees(self.angle, "z"),
```

```
            transforms.rotation_degrees(self.angle, "y"),
            transforms.rotation_degrees(self.angle, "x")))
    self.program1.uniform.transform_matrix.value =
                                    rotation_matrix
    self.program1.uniform.color.value = (1, 0, 0, 1)
    self.verteces_buffer.draw(elements=self.elements_face_1,
                              mode=GL_TRIANGLE_STRIP)
```

这里有三个不同的 draw 函数集，但它们的格式都一样。首先它们使用属性 self.program1.uniform.transform_matix.value 为顶点着色器中的 transform_matirx 赋值。然后使用相似的一行代码为片段着色器中的 color 赋值。最后调用 self.verteces_buffer.draw() 为将像素绘制到 3D 世界中。该函数需要两个参数。第一个参数是 elements，即包含了顶角下标的元素缓存。第二个参数是 mode，用来告诉 OpenGL 这些顶点的含义。这里的三个分别是 GL_POINTS、GL_LINE_STRIP 和 GL_TRIANGLE_STRIP。

GL_POINTS 的含义很明显，它将每个顶点绘制成一个点。GL_LINE_STRIP 表示画一条连续的线，每个顶点都是线上的一个点。还有一种 GL_LINES 模式，表示在每一对顶点间画一条独立的线。

GL_TRIANGLE_STRIP 用来绘制一个连续的三角形。本例中，我们使用 5 个顶点来定义是个正方形。实际上，我们使用了 4 个顶点，只是其中一个使用了两次。使用三角形带，前三个点可以组成一个三角形，然后是第三、四、五个点，然后是第五、六、七个点，然后是第七、八、九个点等[⊖]。前一个三角形的后两个点构成了当前三角形的前两个点。使用这种方法，你可以拼出任意表面。

还有其他两种方法，GL_TRIANGLES 是每三个顶点绘制一个三角形，GL_TRIANGLE_FAN 是每个顶点都共享一个单独的顶点，各个三角形形成一个扇形序列，看起来和花瓣一样。

6.2.2 组合包装

完整代码如下所示（提示，可以从网站上下载 chapter6-spinning-cube.py）：

```
import pygame
from rpigl import glesutils, transforms
from rpigl.gles2 import *

vertices = [(0.0,0.0,0.0), (0.5,0.0,0.0), (0.5,0.5,0.0),
```

⊖ 即新的三角形可重用最后两个索引，这里可能是作者笔误，应该是第二、三、四，三、四、五，四、五、六个点。——译者注

```
            (0.0, 0.5,0.0),
            (0.0,0.0,-0.5), (0.5,0.0,-0.5), (0.5,0.5,-0.5),
            (0.0, 0.5,-0.5)]

indices_face_1 = (0, 1, 2, 0, 3)
indices_face_2 = (4, 5, 6, 4, 7)
indices_face_3 = (1, 5, 6, 1, 2)
indices_face_4 = (0, 4, 7, 0 ,3)
indices_outer = (0, 1, 2, 3, 0, 4, 5, 1, 5, 6, 2, 6, 7, 3, 7, 4)
indices_points = (0, 1, 2, 3)

array_spec = glesutils.ArraySpec("vertex_attrib:3f")

vertex_glsl = array_spec.glsl() + """
uniform mat4 transform_matrix;
void main(void) {
  gl_Position = transform_matrix * vec4(vertex_attrib, 1.0);
  gl_PointSize = 2.0;
}
"""

fragment_glsl = """
uniform vec4 color;
void main(void) {
  gl_FragColor = color;
}
"""

class MyWindow(glesutils.GameWindow):

    def init(self):

        self.angle = 10

        self.vertex_shader = glesutils.VertexShader(vertex_glsl)
        self.fragment_shader =
                    glesutils.FragmentShader(fragment_glsl)

        self.program1 = glesutils.Program(self.vertex_shader,
                                          self.fragment_shader)
        self.program1.use()

        glClearDepthf(1.0)
        glDepthFunc(GL_LESS)
        glEnable(GL_DEPTH_TEST)


        glClearColor(0.5, 0.5, 0.5, 1)

        self.program1.uniform.light_dir.value = ((0, 1, -1))
```

```
        self.verteces_buffer =
                  array_spec.create_buffer(vertex_attrib=vertices)
        self.elements_face_1 =
                  glesutils.ElementBuffer(indices_face_1)
        self.elements_face_2 =
                  glesutils.ElementBuffer(indices_face_2)
        self.elements_face_3 =
                  glesutils.ElementBuffer(indices_face_3)
        self.elements_face_4 =
                  glesutils.ElementBuffer(indices_face_4)

        self.elements_outer =
                  glesutils.ElementBuffer(indices_outer)
        self.elements_points =
                  glesutils.ElementBuffer(indices_points)

        self.outer_matrix =
                  transforms.compose(
                  transforms.rotation_degrees(20, "z"),
                  transforms.rotation_degrees(20, "y"),
                  transforms.rotation_degrees(20, "x"),
                  transforms.scaling(1.2))

        self.points_matrix =
                  transforms.compose(
                  transforms.stretching(0.1, 1, 1.5),
                  transforms.translation(-0.5, -0.5, -0.5))

    def on_frame(self, time):
        self.angle = self.angle + time*0.02
        self.redraw()

    def draw(self):
        #Draw outer lines
        self.program1.uniform.transform_matrix.value =
                                  self.outer_matrix
        self.program1.uniform.color.value = (1, 1, 1, 1)
        self.verteces_buffer.draw(elements=self.elements_outer,
                                  mode=GL_LINE_STRIP)

        #Draw points
        self.program1.uniform.transform_matrix.value =
                                  self.points_matrix
        self.program1.uniform.color.value = (0, 0, 0, 1)
        self.verteces_buffer.draw(elements=self.elements_points,
                                  mode=GL_POINTS)

    #Draw spinning cube
            rotation_matrix = transforms.compose
                   (transforms.rotation_degrees(self.angle, "z"),
                    transforms.rotation_degrees(self.angle, "y"),
                    transforms.rotation_degrees(self.angle, "x"))
```

```
            self.program1.uniform.transform_matrix.value =
                                    rotation_matrix
            self.program1.uniform.color.value = (1, 0, 0, 1)
            self.verteces_buffer.draw(elements=self.elements_face_1,
                                    mode=GL_TRIANGLE_STRIP)
            self.program1.uniform.color.value = (0, 1, 0, 1)
            self.verteces_buffer.draw(elements=self.elements_face_2,
                                    mode=GL_TRIANGLE_STRIP)
            self.program1.uniform.color.value = (0, 0, 1, 1)
            self.verteces_buffer.draw(elements=self.elements_face_3,
                                    mode=GL_TRIANGLE_STRIP)
            self.program1.uniform.color.value = (0, 1, 1, 1)
            self.verteces_buffer.draw(elements=self.elements_face_4,
                                    mode=GL_TRIANGLE_STRIP)

MyWindow(200, 200, pygame.RESIZABLE).run()
```

运行结果应该是一个如图 6-1 所示的 3D 渲染图：

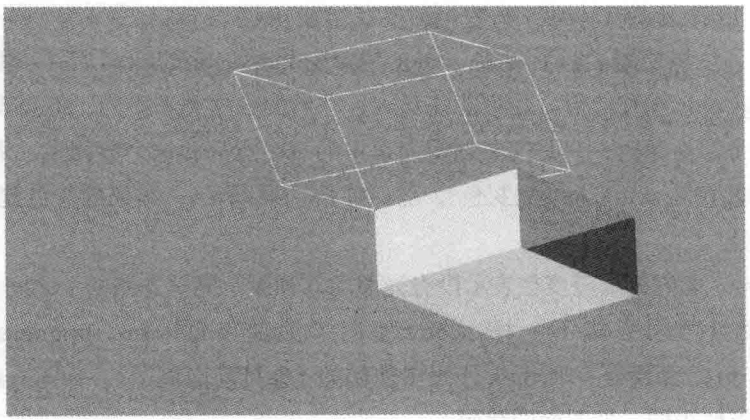


图 6-1　大多数 3D 图形使用的基本技术演示

6.2.3　增加光照

运行前面的例子，你会看到一个立方体的四个面围绕一些线和点在旋转。你可能注意到这里缺少些什么：光。不论这些表面朝向哪里，它们全部都是一个亮度，这几乎不可能出现在现实生活中。相反，根据朝向，总是有一个或更多的光源从不同角度照向物体。

有些版本的 OpenGL 可以自动处理这部分。然而树莓派（和大多数移动设备）上的 OpenGL ES 版本做不到，因此你必须自己来计算光线。

下面的例子将创建一个点光源照射下的旋转立方体。随着立方体的移动，你将会看到

它的不同部分在变亮或者变暗。在计算某个点的亮度时需要使用两个数据——立方体上的点到光源的距离和光线的照射角度。

计算点到光源的距离

我们先看第一点。立方体的每个面都是由屏幕上的上百个像素显示出来。片段着色器计算出这些像素的值。为了计算像素的颜色，片段着色器需要知道每个像素的亮度。这些都由下面的代码来完成：

```
fragment_glsl = """
uniform vec4 color;

varying float brightness;

void main(void) {
  gl_FragColor = brightness*color;
}
"""
```

可以看到，这里使用了两个变量：color 和 brightness。brightness 是浮点型，可以用来替换给 gl_FragColor 设置的值。你可能注意到，它们的数据类型不同，但是没有问题。当给向量乘以一个浮点数时，只是简单地将向量中的每个点都和这个数相乘一下。比如，你有一个颜色向量 (0.8, 0.8, 0.8)（亮灰色），然后你把它乘以 0.5，结果的颜色就是（0.4, 0.4, 0.4）(中等灰色)。

brighteness，取值范围应该在表示没有光照的 0 和表示完全光照的 1 之间。

这两个变量声明方式不一样。color 使用了关键字 uniform，brightness 被设置成 varying。uniform 的变量是在 Python 代码中使用的（参见前面例子）。而 verying 变量是用来给顶点着色器使用的，见下面的代码：

```
vertex_glsl = array_spec.glsl() + """

...

varying float brightness;

void main(void) {
  ...6;
  float distance = length(vec4(light_position, 1.0) - gl_Position);
  brightness = 1.0/(distance * distance);
}
"""
```

这里没有计算角度和位置的代码（后面我们再介绍），但这段代码给出了如何根据距离来计算亮度明暗。length() 函数返回一个向量的长度（根据勾股定理计算出来）。本例中，

它用来计算顶点和光源的距离。

当物体远离光源时，它就会变暗。然而，这种变化并不是线性的。如果你把物体移动到原来距离的两倍远，它并不会变得只有以前的一半亮，而是会只剩下四分之一的亮度。物体的亮度和它到光源的距离的平方成反比，这体现在最后一行代码中。

这段代码只计算了顶点的亮度。为了让程序能更好地运行，它需要知道表面上每个点的亮度。幸运的是，OpenGL 为我们做好了这部分。当你使用 varying 关键字来创建变量时，它会插补一些值传入片段着色器。也就意味着它会根据每个点到三个顶点的距离改变该点的亮度，以渲染出一个平滑协调的亮度。

计算反射角

亮度的另一个因素是表面和光源的角度。为了计算这个角度。你需要一个法线。所谓法线就是垂直于平面的向量。

向量可以用于描述位置，也可以用来描述直线。例如，向量 (1, 1, 1) 可以表示位于这个坐标的点，也可以表示 3D 世界中的一条从点 (0, 0, 0) 到 (1, 1, 1) 的直线。顶点的向量表示顶点的位置，法线的向量表示直线。

法线是一个表示物体朝向的向量。以立方体的第一面为例，顶点为 (0.0, 0.0, 0.0), (0.5, 0., 0.0)，(0.5, 0.5, 0.0)，(0.0, 0.5, 0.0)。所有顶点都在 z 轴的一个平面上，因此它的法线是 (0, 0, 1.0)。如果你从点 (0, 0, 0) 到 (0, 0, 1.0) 画一条线，它将和该平面成 90 度角。法线的长度通常都是 1。

为了计算亮度，你需要计算法线和光源的角度。如果它们在同一个直线上，物体表面应该是全亮。如果成 90 度或者更大角度，该平面不会被照亮。然而，和距离类似，这两者之间也不是线性关系，而是成正弦关系。正弦函数正好和我们的情况不符，因此你需要计算余弦。

为此，你需要使用三角函数工具箱。这里有个函数叫 dot product（点积），需要两个向量，返回值如下所示：

```
dot(A,B) = length(A) * length(B) * cosine(A,B)
```

计算点积远比计算余弦快，因此我们可以利用这个数学捷径。如果每个变量的长度都为 1，事情就会变得很简单，这时点积就等于余弦。

法线的长度永远为 1，因此你不用担心它。GLSL 中有个 normalize() 函数，可以得到任意向量的归一化向量——和该向量方向相同，长度为 1。因此你可以计算这两个向量的余弦。

```
cosine = dot(surface_normal, normalize(gl_Position -
                                       light_position)
```

然而，大多数时候并不止有一个光源。还有很多从墙壁和其他物体的反光。将所有这些都计算进来显然会很复杂。简单的做法就是增加一个环境光量。也就是简单地将从各个角度照向物体的光加在一起。可以这样添加环境光：

```
brightness = max(cosine, ambient_light)
```

下面代码将所有这些合并在一起构成顶点着色器：

```
vertex_glsl = array_spec.glsl() + """
uniform mat4 transform_matrix;
uniform vec3 light_position;
uniform float ambient_light;
uniform vec3 face_normal;

varying float brightness;
void main(void) {
  gl_Position = transform_matrix * vec4(vertex_attrib, 1.0);
  vec4 spun_face_normal = normalize(transform_matrix *
                                    vec4(face_normal, 1.0));
  float distance = length(vec4(light_position, 1.0) - gl_Position);
  vec4 light_direction = normalize(vec4(light_position, 1.0) -
                                   gl_Position);
  float light_amount_angle = max(dot(spun_face_normal,
                                 light_direction), ambient_light);

  float light_distance_drop = 1.0/(distance * distance);
  brightness = light_amount_angle * light_distance_drop;
}
"""
```

这里点积计算并不是直接使用原始表面法线，而是使用经过变换的立方体的表面法线。本例中，从技术上说 transform_matrix 并不缩放或者拉伸法线，因此不需要规格化。然而为了使代码能够用在其他程序中使用，我们加上了规格化。

剩下的代码和上一个例子基本相同。这里没有点或线，只有立方体的 6 个面。这段代码在网站上的 chapter6-lighting.py。

```
import pygame
from rpigl import glesutils, transforms
from rpigl.gles2 import *

vertices = [(0.0,0.0,0.0), (0.5,0.0,0.0), x/4984720?c=pledges
            (0.5,0.5,0.0), (0.0, 0.5,0.0),
            (0.0,0.0,-0.5), (0.5,0.0,-0.5),
            (0.5,0.5,-0.5), (0.0, 0.5,-0.5)]
```

```
faces = [{"vertex_index":(0, 1, 2, 0, 3), "normal":(0,0,1),
          "colour":(1, 0, 0, 1)},
         {"vertex_index":(4, 5, 6, 4, 7), "normal":(0,0,-1),
          "colour":(0, 1, 0, 1)},
         {"vertex_index":(1, 5, 6, 1, 2), "normal":(1,0,0),
          "colour":(0, 0, 1, 1)},
         {"vertex_index":(0, 4, 7, 0 ,3), "normal":(-1,0,0),
          "colour":(1, 0, 1, 1)},
         {"vertex_index":(3, 2, 6, 3, 7), "normal":(0,1,0),
          "colour":(1, 1, 0, 1)},
         {"vertex_index":(0, 1, 5, 0, 4), "normal":(0,-1,0),
          "colour":(0, 1, 1, 1)}]
array_spec = glesutils.ArraySpec("vertex_attrib:3f")

vertex_glsl = array_spec.glsl() + """
uniform mat4 transform_matrix;
uniform vec3 light_position;
uniform float ambient_light;
uniform vec3 face_normal;

varying float brightness;

void main(void) {
  gl_Position = transform_matrix * vec4(vertex_attrib, 1.0);
  vec4 spun_face_normal = normalize(transform_matrix *
                                    vec4(face_normal, 1.0));
  float distance = length(vec4(light_position, 1.0) - gl_Position);
  vec4 light_direction = normalize(vec4(light_position, 1.0) -
                                   gl_Position);
  float light_amount_angle = max(dot(spun_face_normal,
                                     light_direction), ambient_light);
  float light_distance_drop = 1.0/(distance * distance);
  brightness = light_amount_angle * light_distance_drop;
  gl_PointSize = 2.0;
}
"""

fragment_glsl = """
uniform vec4 color;

varying float brightness;

void main(void) {
  gl_FragColor = brightness*color;
}
"""

class MyWindow(glesutils.GameWindow):

    def init(self):
```

```
        self.angle = 10
        self.framerate = 20

        self.vertex_shader = glesutils.VertexShader(vertex_glsl)
        self.fragment_shader =
                        glesutils.FragmentShader(fragment_glsl)
    self.program1 = glesutils.Program(self.vertex_shader,
                                      self.fragment_shader)
    self.program1.use()

    glClearDepthf(1.0)
    glDepthFunc(GL_LESS)
    glEnable(GL_DEPTH_TEST)
    glFrontFace(GL_CW)

    glClearColor(0.5, 0.5, 0.5, 1)

    self.program1.uniform.light_dir.value = ((0, 1, -1))

    self.verteces_buffer =
             array_spec.create_buffer(vertex_attrib=vertices)
    for face in faces:
        face["element_buffer"] =
             glesutils.ElementBuffer(face["vertex_index"])

    self.outer_matrix = transforms.compose(
            transforms.rotation_degrees(20, "z"),
            transforms.rotation_degrees(20, "y"),
            transforms.rotation_degrees(20, "x"),
            transforms.scaling(1.2))

    self.points_matrix = transforms.compose(
                    transforms.stretching(0.1, 1, 1.5),
                    transforms.translation(-0.5, -0.5, -0.5))

def on_frame(self, time):
    self.angle = self.angle + time*0.02
    self.redraw()

def draw(self):
    self.program1.uniform.light_position.value = (0,0,-1)
    self.program1.uniform.ambient_light.value = 0.3

    rotation_matrix = transforms.compose(
            transforms.rotation_degrees(self.angle, "z"),
            transforms.rotation_degrees(self.angle, "y"),
            transforms.rotation_degrees(self.angle, "x"))

    self.program1.uniform.transform_matrix.value =
                                        rotation_matrix
```

```
        for face in faces:
            self.program1.uniform.color.value = face["colour"]
            self.program1.uniform.face_normal.value =
                                                face["normal"]

        self.verteces_buffer.draw(elements=face["element_buffer"],
                                  mode=GL_TRIANGLE_STRIP)

MyWindow(200, 200, pygame.RESIZABLE).run()
```

结果如图 6-2 所示。

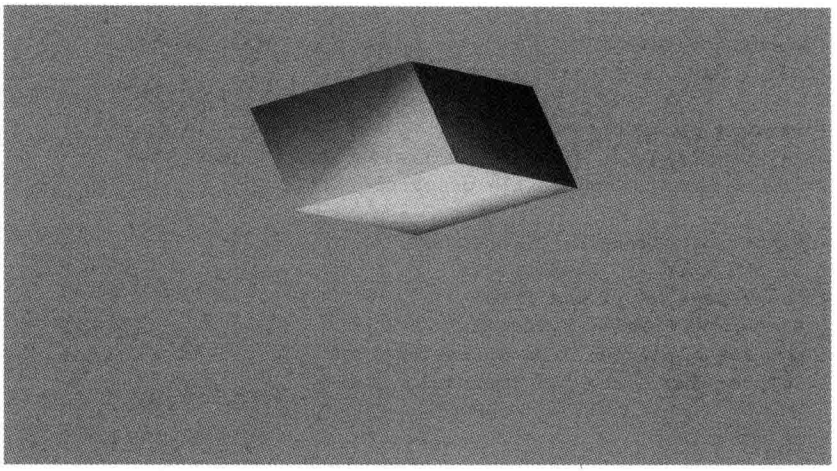


图 6-2　和第一个程序不同，光线增加了现实感

6.3　让屏幕起舞

至此，你见识过如何在屏幕上绘制立方体并照亮它们，但仅限于此。在下一个项目中，你将看到如何创建一个随音乐起舞的 3D 模型。大多数计算机音乐播放器都有相似的功能，为了听众增加点视觉效果而在屏幕上显示出声音效果。

因为本章主要介绍 3D 图形而不是音频处理，为了简单起见我们只处理 WAV 文件。也就是说我们可以使用 Python 的 wave 模块从文件中读取声音数据。如果你的音乐是 MP3 格式，你需要先把它们转换成 WAV 格式，可以在 LXTerminal 下通过 mpg123 命令行工具完成转换。使用之前需要先安装：

```
sudo apt-get install mpg123
```

然后就可以用下面命令完成转换：

```
mpg123 -w output-filename.wav input-filename.mp3
```

如果没有合适的音乐，可以从这个网站上下载一些免费音乐：http://freemusicarchive.org/。

第一步是播放音乐。和第 5 章中播放音乐的方法一样，也是使用 PyGame 合成器。它将音乐通过树莓派的音频通道播放出来，并不会给你提供用来控制 3D 图形的声音数据。为此，我们将使用第二个模块 wave。

```
print("opening file")
sound_file = wave.open("test.wav",';rb')

print("getting parameters")
(channels, sample_size, frame_rate, frames, compression_type,
  compression_name) = sound_file.getparams()

print("Number of channels: ", channels)
print("Sample size: ", sample_size)
print("Frame rate: ", frame_rate)
print("Number of Frames: ", frames)
print("Compression type: ", compression_type)
print("Compression name: ", compression_name)

print("readframes")
data = sound_file.readframes(channels*sample_size*frames)
print(len(data))
```

这里创建了变量 data，声音以字节形式存储在里面。声音编码是两个字节一组。为了读取一组数据，需要使用 int 类中的 from_byte() 方法将两个字节组合在一起。

```
sound_data = int.from_bytes(data[i:i+1],
                            byteorder='little', signed=True)
```

声音以帧的形式存储，每帧就是上面提到的一组两个字节，其取值范围从 -32 768 到 32 767。如果文件是立体声的，每个时间点有两帧数据。通常每秒有 44 100 帧（该数值存储在变量 frame_rate 中）。

你可以用很多种方式展示这些数据。唯一的需求就是当声音播放时，输出需要以娱乐化的方式展现出来。我们将使用两种方法。当音量变化时从屏幕底下绘出一组变化的 3D 条带，和一组闪光点（如果你喜欢诗意，可以使用星形）用来点缀音乐。为了增加一点花式，我们将整个可视化部分沿纵轴旋转起来，并把 3D 条的颜色设置成从底端的蓝色渐变到顶端的红色。