这本书将向你介绍面向对象范型的术语，通过一步步的例子，专注于面向对象的设计。它将带你从简单的继承开始，这在面向对象程序员工具箱里是最有用的工具之一，到最复杂之一的合作继承。你将能提高、处理、定义以及操作异常。

你将能够把Python面向对象和不是那么面向对象的方面结合起来。通过研究高级的设计模式，你将能够创建可维护的应用程序。你将学习Python复杂的字符串和文件操作. 以及如何区分二进制和文本数据。将会介绍给你不止一个，而是两个非常强大的自动化测试系统。你将会理解单元测试的喜悦以及创建单元测试是多么简单。你甚至会学习像数据库连接和GUI「. 具包这样的高级库，以及它们是如何应用面向对象原则的。

这本书讲了什么

第1章，面向对象设计覆盖了重要的面向对象概念。它主要处理关于抽象、类、封装和继承。在建模我们的类和对象时，我们也简要地看了下UML。

第 2 章， Python对象讨论了类和对象，以及它们是如何在Python中使用的。我们将学习Python对象中的属性和行为，以及把类组织成包和模块。最后我们将看到如何保护我们的数据。

第3章，当对象是相似的Lt我们从更深层次的视角来看继承。它覆盖了多重继承以及向我们展示了如何从内置来继承。这一章还包括了多态以及鸭子类型。

第4章，异常处理讲解异常和异常处理。我们将学习如何创建自己的异常。它还涉及了把异常用于程序流程控制。

第5章，何时使用面向对象编程主要处理对象，什么时候创建和使用它们。我们将看到如何使用属性来包装数据，以及限制数据访问。这一章也会讨论DRY原则以及如何不重复代码。

第6章，Python数据结构後盖了数据结构的面向对象特征。这一章主要处理元组、字典、列表和集合。我们也会看一看如何扩展内置对象。

第7章，Python里面向对象的快捷方式顾名思义，在Python中如何省时。我们将看到很多有用的内置函数，然后继续在列表、集合和字典中使用解析ˆ我们将学习关于生成器、方法重载，以及默认参数的内容。我们还会看到如何把函数当成对象来用。

第8章，Python设计模式1第一次向我们介绍了 Python设计模式。然后我们将会看到装饰器模式、观察者模式、策略模式、状态模式、单例模式以及模板模式。这些模式将会和在Python中实现的合适的例子和程序一起讨论。

第9章，Python设计模式2讲解上一章剩下的内容。我们将通过Python中的例子看到适配器模式、外观模式、享元模式、命令模式、抽象模式以及组合模式。

第10章，文件和字符串学习字符串和字符串格式化。也会讨论字节和字节数组。我们也将会学习文件以及如何从文件读取数据和往文件里写数据。我们将学习存储和pickle对象，最后本章会讨论序列化对象。

第丨丨章，测试面向对象的程序以使用测试和为什么测试如此重要开头。重点是测试驱动开发。我们将看到如何使用unittest模块，以及py. test自动化测试套件。最后我们将使用coverage .py来看代码的测试覆盖率。

第12章，常用Python 3库集中介绍库以及它们在程序构建中的利用率。我们将使用SQLAlchemy来构建数据库，使用Tklnter和PyQt开发用户界面。这一章会继续讨论如何构建XML文档以及我们应该如何使用ElementTree和lxml。最后我们将使用CherryPy和Jinja来创建一个Web应用。

对于这本书你需要什么

为了编译运行在本书中提到的例子，你需要下面的软件：

- Python 3.0或者更高的版本
- py.test
- coverage.py
- SQLAlchemy
- pygame
- PyQt
- CherryPy
- lxml

谁需要这本书

如果你是面向对象编程技术的新手，或者你有基本的Python技巧，并且希望深人地学习如何以及什么时候在Python中正确地应用面向对象编程，这本书适合你。

如果你是一个其他语言的面向对象编程人员，你也会发现这本书是对Python的一个有用的介绍，因为它使用了一些你已经熟悉的术语。

那些寻求迈人Python 3新世界的Python 2程序员也将会发现这本书的好处，但是其实你不需要了解Python 2。

惯例

在这本书中，你将发现一些用于区分不同种类信息的文本风格。这里有一些这些风格的例子，并且介绍了它们的意义。

文本形式的代码会按照下面显示：“我们可以通过使用import语句来访问Python的其他模块”。

一个代码块会像下面这样显示：

```
class Friend(Contact):
      def ____init____(self, name, email, phone):
            self.name = name
            self.email = email
            self.phone = phone
```

当我们希望你对一个代码片段里特殊的部分引起注意的时候，相应的行或者元素会设置成粗体：

```
class Friend(Contact):
      def ____init____(self, name, email, phone):
            self.name = name
            self.email = email
            self.phone = phone
```

任何一个命令行的输人或输出都是下面这样的形式：

```
»> e = EmailableCon tact ("John Smith" , " j smith@exan^>le. net")
»> Contact.all contacts
```

新的术语或者重要的词将会显示成黑体。例如你在屏幕、菜单和对话框中看到的词，会以文本的形式这样显示：“你每一次点击Roll!按钮的时候，我们会通过这个特性来给标签更新一个新的随机值’

警告或者重要的笔记将会这么显示。

小窍门和诀窍会这么显示。

读者反馈

我们随时欢迎来自读者的反馈信息。请告诉我们你对本书的评价——你喜欢或者不喜欢的地方。读者的反馈对于我们是非常重要的，这会帮助我们为你创作更有价值的作品。

请将你的反馈意见通过邮件feedback@packtpub. com发送给我们，并在邮件的主题中标明相关图书的名称。

如果你需要一本书并且希望我们出版相关图书，请在www. packtpub. com用SUGGESTATITLE^^^；—suggest@packtpub. com^诉我们。

如果你具有关于某个主题的专业知识，并且有兴趣参与图书的编写，请查看我们的作者指南，网址： www. packtpub. com/authors。

客户支持

你现在是拥有Packt书的尊贵主人，购买本书我们会从以下方面帮助到你。

下载本书的示例代码

_ i 你可以通过你在http: //www . packtpub. com的账号来下载你购买图书的所有示例代码文件。如果你从其他地方购买的本书，你可以访问http://www. packtpub. com/support并且注册，我们会直接通过电子邮件发给你文件。

勘误

虽然我们已尽力确保我们内容的准确性，但是错误还是会发生。如果你发现本书的一个错误——文字或者代码错误——如果你能报告给我们，我们将非常感激。通过这样做，你可以把其他读者从挫败中拯救出来并且帮助我们改善这本书的后续版本。如果你发现任何错误，请通过访问http://packtpub.com/support来报告它们，选择你的书，点击 | etus know链接，并且输人错误的细节。一旦你的勘误被核实，你提交的信息将会被接收并且将会把勘误上传到我们的网站，或者在添加到一个已有的勘误列表的相应主题部分。从http://www.packtpub.com/support选择我们的信息标题将看到任何一个存在的勘误。

盗版

互联网上著作权侵害是在所有媒体中持续存在的问题。在Packt, 我们会非常重视对版权和许可证的保护。如果你在互联网上发现了对于我们著作的任何形式的任何非法复制，请立即告诉我们地址或者网站名字，以便我们能进行补救。

请通过copyright@packtpub.com联系我们，可附带一个可疑链接的盗版材料。

我们非常感谢你在保护我们的作者以及我们能带给你有价值内容的能力方面给予我们的帮助。

问题

如果你对这本书的任何方面有问题，你可以通过questions@packtpub.com联系我们，我们将尽力解决这个问题。

Object Oriented Programming

Python 3

[redacted]

[加]Dusty Phillips 著
肖 鹏 常 贺 石 琳 译

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

一一一. c⑽ . ,„专⑽点• • ITas 版臓口。口 w
c 力 • $ 出

Python 3
面向对象编程

在现代编程语言中. 面向对象编程是非常重要的一块. 面向对象编程的基本原则学起来也相对简单但是把它们联合起来用到设计工作中是个挑战

通过使用Python 3强大的面向对象特性. 这本书使得编程变得更加有趣而不至于累它清晰地展示了面向对象的核心原则. 以及如何在Python中正确应用它们，面向对象编程在Python支持的众多模型中重要性排名较高：然而，很多开发者不愿意花时间去学习这个能让语言面向对象的强大特性. 这本书教我们什么时候以及如何正确地应用面向对象编程. 它强堝的不止是Python中简单的面向对象编程语法. 还有如何把这些对象组合起来，成为精心设计的软件：

这本书为谁而著

■ 如果你是面向对象编程技术的新手. 或者你有基本的Python技巧. 并且希望深入学习如何以及什么时候在Python中正确地应用面向对象编程，这本书适合你。

■ 如果你是一个其他语言的面向对象编程人员，你也会发现这本书是对Python的一个有用的介绍，因为它使用了一些你已经熟悉的术语(，

■ 那些寻求迈入Python 3新世界的Python ?程序员也将会发现这本书的好处，但是其实你不詣要了解 Python 2。

[PACKT]
PUBLISHING

你格从本书学到

- 通过一步步的教程来学习如何在Python中应用面向对象编程。
- 通过抽象、封装和信息隐藏来设计公共接口。
- 通过学习Python语法把你的设计变成可以工作的软件。
- 通过使用特定的错误对象来触发、处理、定义和操纵异常。
- . 通过实际例子在Python中实现面向对象编程。
- 理解什么时候使用面向对象特性，更重要的是，什么时候不用。
- -学习什么是设计模式以及为什么它们在Python中是不同的。 ，
- 揭示单元测试的简单以及为什么它在Python中如此重要。

策划编辑：张春雨付訾
责任编辑：付容
封面设计：李玲

上架建议：编程语言>Python

ISBN 978-7-121-26246-3
9 787121 262463
定价：79.00元

Python 3

Object Oriented Programming

Python 3

面向对象编程

[加]Dusty Phillips 著

肖 鹏 常 贺 石 琳 译

专 子 X f出 铖 此

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

Python是一种面向对象的解释型语言，面向对象是其非常重要的特性。本书通过Python的数据结构、语法、设计模式.从简单到复杂，从初级到高级.一步步通过例子来展示了 Python中面向对象的概念和原则。

本15不是Python的入门书籍.适合具有Python基础经验的开发人员阅读。如果你拥有其他面向对象语&的经验，你会更容易理解本书的内容。

版权贸易合同登I己号图字：01-2015-0634

图书在版编目（CIP)数据

卩^11〇113面向对象编程/(加）菲利普斯（？1^丨冲5,0.)著;肖鹏，常贺，石琳译.一北京;电子工业出版社，2015.7
t?名原文 Python 3 Object Oriented Programming
ISBN 978-7-121-26246-3

Ⅰ.①P…丨丨.①菲…②肖…③常…④石…II丨.①软件工具一程序设计IV.①TP311.56

中国版本图书馆CIP数据核字(2015)第122883号

策 划 编 辑 ： 张 春 雨 付 睿
赏任编辑：付睿
印 刷：北京中新伟业印刷有限公司
装　汀：三河市华成印务有限公司
出版发行：电子工业出版社
北京市海淀区万寿路173信箱邮编：100036
开 本：787x980丨/16　印张：23.75　字数：478千字
版　次：2015年7月第1版
印　次：2015年7月第I次印刷
定　价：79.00元

凡所购买电子工业出版社图书有缺损问题.请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.ciu

服务热线：（010）88258888。

关于作者

Dusty PhHIips是一名加拿大自由软件开发人员、教师、武术家以及开源爱好者。他和Arch Linux社区以及其他开源社区有着紧密联系。他维护着Arch Linux的门户网站以及编译了流行的Arch Linux手册。Dusty持有计算机科学人机交互方向的硕士学位。他目前在自己的计算机里安装了 6种不同的Python解释器。

我要感谢我的编辑Steven Wilding和Mayuri Kokate对我适时的鼓励和反馈。感谢我的朋友及导师Jason Chu比我开始学习Python, 数年来耐心地回答我关于Python、GIT和生活中的各种问题。感谢我的父亲C. C. Phillips, 激励我在编辑他了不起的小说作品的同时进行写作。最后感谢每一个说迫不及待想买我的书的人；你们的热情对我是一股巨大的激励力量。

关于审校

Jason Chu是Oprius软件公司联合创始人，CTO。他有超过8年的软件开发经验。Chu从2003年版本2.2时就开始使用Python。当他不开发个人或者专业软件的时候，会在空闲时间教空手道、下围棋，以及在他的家乡加拿大维多利亚BC玩。你会经常发现他在Christie马车房子里喝Back Hand of God Stout这种酒。

Michael Drisco丨丨用Python编程已近4年，从20世纪90年代末开始涉足其他语言。他从大学毕业获得科学学士学位，主修信息系统管理。Michael为了兴趣和利益编程。他的爱好包含圣经教义，写关于 Python 的博客 http://www.blog.pythonlibrary.org/，以及学习摄影。Michael目前在当地政府工作，一个能让他尽可能使用Python编程的地方。这是他作为技术审校的第一本书。

我要感谢我的妈妈，没有她我不可能成长成现在这样热爱学习的我。我还想感谢Scott Williams迫使我学习Python,没有他，我甚至不知道这个语言的存在。最重要的是，我要感谢耶稣让我救赎自己。

Dan McGee是目前生活在伊利诺伊州芝加哥的一名软件开发人员，主要在芝加哥地区全职做**java Web**开发，并且有多年的经验；当然，他也使用其他语言。**Dan**也从事一些自由项目。2007年，**Dan**成为**Arch Linux**发行版的一名开发者，并且从那以后做了很多和**Arch Linux**相关的项目，包括破解包管理器代码，兼职管理系统，以及帮助维持和改善网站，

Lawrence O丨uyede是一名26岁的**Python**和网络编程方面的软件开发专家。他非常乐见编程并行化以及函数式编程语言成为主流。他是由出版的意大利第一本**Ruby**书（沿咖**per vve**6 ）的意大利版的共同作者和审校。他在过去也为其他的书做过贡献，*i$-Python* **Coo/rfeooA:** （ **http://www. amazon . com/Python-Cookbook-Alex-Martelli/dp**/0596007973/） *VIJSc. The Definitive Guide to Django* （**http://www. amazon. com/Definitive-Guide-Django-Development-Right/dp**/1590597257 ）〇

这本书将向你介绍面向对象范型的术语，通过一步步的例子，专注于面向对象的设计。它将带你从简单的继承开始，这在面向对象程序员工具箱里是最有用的工具之一，到最复杂之一的合作继承。你将能提高、处理、定义以及操作异常。

你将能够把Python面向对象和不是那么面向对象的方面结合起来。通过研究高级的设计模式，你将能够创建可维护的应用程序。你将学习Python复杂的字符串和文件操作. 以及如何区分二进制和文本数据。将会介绍给你不止一个，而是两个非常强大的自动化测试系统。你将会理解单元测试的喜悦以及创建单元测试是多么简单。你甚至会学习像数据库连接和GUI「. 具包这样的高级库，以及它们是如何应用面向对象原则的。

这本书讲了什么

第1章，面向对象设计覆盖了重要的面向对象概念。它主要处理关于抽象、类、封装和继承。在建模我们的类和对象时，我们也简要地看了下UML。

第2章，Python对象讨论了类和对象，以及它们是如何在Python中使用的。我们将学习Python对象中的属性和行为，以及把类组织成包和模块。最后我们将看到如何保护我们的数据。

第3章，当对象是相似的Lt我们从更深层次的视角来看继承。它覆盖了多重继承以及向我们展示了如何从内置来继承。这一章还包括了多态以及鸭子类型。

第4章，异常处理讲解异常和异常处理。我们将学习如何创建自己的异常。它还涉及了把异常用于程序流程控制。

第5章，何时使用面向对象编程主要处理对象，什么时候创建和使用它们。我们将看到如何使用属性来包装数据，以及限制数据访问。这一章也会讨论DRY原则以及如何不重复代码。

第6章，Python数据结构後盖了数据结构的面向对象特征。这一章主要处理元组、字典、列表和集合。我们也会看一看如何扩展内置对象。

第7章，Python里面向对象的快捷方式顾名思义，在Python中如何省时。我们将看到很多有用的内置函数，然后继续在列表、集合和字典中使用解析ˆ我们将学习关于生成器、方法重载，以及默认参数的内容。我们还会看到如何把函数当成对象来用。

第8章，Python设计模式1第一次向我们介绍了 Python设计模式。然后我们将会看到装饰器模式、观察者模式、策略模式、状态模式、单例模式以及模板模式。这些模式将会和在Python中实现的合适的例子和程序一起讨论。

第9章，Python设计模式2讲解上一章剩下的内容。我们将通过Python中的例子看到适配器模式、外观模式、享元模式、命令模式、抽象模式以及组合模式。

第10章，文件和字符串学习字符串和字符串格式化。也会讨论字节和字节数组。我们也将会学习文件以及如何从文件读取数据和往文件里写数据。我们将学习存储和pickle对象，最后本章会讨论序列化对象。

第丨丨章，测试面向对象的程序以使用测试和为什么测试如此重要开头。重点是测试驱动开发。我们将看到如何使用unittest模块，以及py. test自动化测试套件。最后我们将使用coverage .py来看代码的测试覆盖率。

第12章，常用Python 3库集中介绍库以及它们在程序构建中的利用率。我们将使用SQLAlchemy来构建数据库，使用Tklnter和PyQt开发用户界面。这一章会继续讨论如何构建XML文档以及我们应该如何使用ElementTree和lxml。最后我们将使用CherryPy和Jinja来创建一个Web应用。

对于这本书你需要什么

为了编译运行在本书中提到的例子，你需要下面的软件：

- Python 3.0或者更高的版本
- py.test
- coverage.py
- SQLAlchemy
- pygame
- PyQt
- CherryPy
- lxml

谁需要这本书

如果你是面向对象编程技术的新手，或者你有基本的Python技巧，并且希望深人地学习如何以及什么时候在Python中正确地应用面向对象编程，这本书适合你。

如果你是一个其他语言的面向对象编程人员，你也会发现这本书是对Python的一个有用的介绍，因为它使用了一些你已经熟悉的术语。

那些寻求迈人Python 3新世界的Python 2程序员也将会发现这本书的好处，但是其实你不需要了解Python 2。

惯例

在这本书中，你将发现一些用于区分不同种类信息的文本风格。这里有一些这些风格的例子，并且介绍了它们的意义。

文本形式的代码会按照下面显示：“我们可以通过使用import语句来访问Python的其他模块”。

一个代码块会像下面这样显示：

```
class Friend(Contact):
    def __init__(self, name, email, phone):
        self.name = name
        self.email = email
        self.phone = phone
```

当我们希望你对一个代码片段里特殊的部分引起注意的时候，相应的行或者元素会设置成粗体：

```
class Friend(Contact):
    def __init__(self, name, email, phone):
        self.name = name
        self.email = email
        self.phone = phone
```

任何一个命令行的输人或输出都是下面这样的形式：

```
»> e = EmailableCon tact ("John Smith" , " j smith@exan^>le. net")
»> Contact.all contacts
```

新的术语或者重要的词将会显示成黑体。例如你在屏幕、菜单和对话框中看到的词，会以文本的形式这样显示：“你每一次点击Roll!按钮的时候，我们会通过这个特性来给标签更新一个新的随机值’

警告或者重要的笔记将会这么显示。

小窍门和诀窍会这么显示。

读者反馈

我们随时欢迎来自读者的反馈信息。请告诉我们你对本书的评价——你喜欢或者不喜欢的地方。读者的反馈对于我们是非常重要的，这会帮助我们为你创作更有价值的作品。

请将你的反馈意见通过邮件feedback@packtpub. com发送给我们，并在邮件的主题中标明相关图书的名称。

如果你需要一本书并且希望我们出版相关图书，请在www. packtpub. com用SUGGESTATITLE^^^；—suggest@packtpub. com^诉我们。

如果你具有关于某个主题的专业知识，并且有兴趣参与图书的编写，请查看我们的作者指南，网址： www. packtpub. com/authors。

客户支持

你现在是拥有Packt书的尊贵主人，购买本书我们会从以下方面帮助到你。

下载本书的示例代码

_ i 你可以通过你在http：//www . packtpub. com的账号来下栽你购买图书的所有示例代码文件。如果你从其他地方购买的本书，你可以访问http://www. packtpub. com/support并且注册，我们会直接通过电子邮件发给你文件。

勘误

虽然我们已尽力确保我们内容的准确性，但是错误还是会发生。如果你发现本书的一个错误——文字或者代码错误——如果你能报告给我们，我们将非常感激。通过这样做，你可以把其他读者从挫败中拯救出来并且帮助我们改善这本书的后续版本。如果你发现任何错误，请通过访问http://packtpub.com/support来报告它们，选择你的书，点击 | etus know链接，并且输人错误的细节。一旦你的勘误被核实，你提交的信息将会被接收并且将会把勘误上传到我们的网站，或者在添加到一个已有的勘误列表的相应主题部分。从http://www.packtpub.com/support选择我们的信息标题将看到任何一个存在的勘误。

盗版

互联网上著作权侵害是在所有媒体中持续存在的问题。在Packt, 我们会非常重视对版权和许可证的保护。如果你在互联网上发现了对于我们著作的任何形式的任何非法复制，请立即告诉我们地址或者网站名字，以便我们能进行补救。

请通过copyright@packtpub.com联系我们，可附带一个可疑链接的盗版材料。

我们非常感谢你在保护我们的作者以及我们能带给你有价值内容的能力方面给予我们的帮助。

问题

如果你对这本书的任何方面有问题，你可以通过questions@packtpub.com联系我们，我们将尽力解决这个问题。

目录

第1章面向对象设计 1

..... 1

雜艄 3

指定属性和行为 5

数据描述对象 5

行为是动作 7

隐藏细节并且创建公共接口 8

组合和继承 10

^7|C 12

細判 15

练习 22

总、结 23

第2章 Python对象 24

创建Python类 24

添加属性 26

让类实际做一些事情 26

对象的初始化 29

解释你自己 32

讎她 34

顧艱 36

谁可以访问我的数据 41

細料 43

..... 52

总结 53

第3章当对象是相似的.....54

縣縣.....54

扩展内置类.....56

重写和Super.....58

.....59

钻石的问题.....61

不同的参数集合.....66

多态.....68

額判.....71

练习.....84

^.....85

第4章异常处理.....86

■娜.....86

抛出一个异常.....88

当一个异常产生时发生了什么.....90

##細.....91

.....97

定义自己的异常.....98

异常不是例外.....99

類判.....102

练习.....112

.....113

第5章何时使用面向对象编程.....114

把对象当作“对象”来对待.....114

使用property为类中的数据添加行为.....118

property是怎样工作的.....121

装饰器：创建property的另一种方法.....123

何时该使用property诚性.....125

M又橡.....127

移除重复的代码. 130
. 131
或者我们可以使用组合. 135
糊判. 137
练习. 145
总结. 146

第6章Python数据结构. 147
. 147
元组和命名元组. 148
命 SjE 组. 150
. 152
何时应该使用字典. 155
使用 defaultdict. 156
. 157
对列表排序. 160
齡. 162
扩展内置数据类型. 166
案例学习. 171
练习. 178
ˆ. 178

第7章Python里面向对象的快捷方式. 180
Python内置函数. 180
Len. 180
Reversed. 181
Enumerate. 182
Zip. 183
. 185
解析. 186
酿撕. 186
集合和字典解析. 188

生成器表达式..... 189
..... 191
方法重载的另一种选择..... 194
默认参数..... 195
可变参数列表..... 197
参数拆分..... 201
函数也是对象..... 202
使用函数作为属性..... 206
可调用对象..... 207
麵 M..... 208
练习..... 212
总结..... 213

第 8 章 设 计 模 式 1..... 214

册献..... 214
雜器模@..... 215
装饰器实例..... 216
Python中的装饰器模式..... 219
酿者模K..... 221
观察者实例..... 222
..... 224
策聽伊J..... 225
Python中的策略模式..... 226
..... 227
從态獅..... 227
状态和策略模式的对比..... 233
单件模式..... 234
单件的实现方式..... 234
模块变量能够校仿单件..... 235
..... 238
觀銷..... 238

. 242
. 243

第9章 设计模式2 244

瓶器模4. 244
夕卜观模t. 247
. 250
命令赋. 254
抽象工厂模式. 259
. 263
. 267
总结. 268

第10章文件和字符串 270

. 270
字符串操作. 271
字符串格式化. 274
字符串是Unicode的. 281
可变字节字符串. 285
I/O. 286
把它放在上下文. 287
. 289
#储[雜. 290
獅J pickle. 292
序列化Web对象. 294
练习. 297
. 299

第11章测试面向对象的程序. 300

为什么要测试. 300
测试驱动开发. 302
. 303

断言方法. 304
减少样板和清理. 306
组织和运行测试. 308
忽略失败的测试. 309
用 py.test 测试 311
一个处理安装和清理的方法. 313
一种完全不同的变量设S方式. 316
用py. test跳过测试. 320
py.test 的补充. 321
多少测试才算够 323
. 326
. 327
练习 332
总、结 . . . . . : 333

第12章 常用Python 3库 334

数据库访问 335
引人 SQLAlchemy. 336
漂亮的用户界面 340
Tklnter. 341
PyQt. 345
选择一个GUI工具包. 347
XML. 348
ElementTree. 349
Lxml. 353
CherryPy. 354
一个完整的Web堆栈. 357
练习. 363
Pig. 364

第1章面向对象设计

在软件开发过程中，通常认为在编程之前要完成设计。这种说法并不准确；在实际中，分析、编程和设计趋向于重叠、结合和交织。通过本章内容，我们会学到：

- 面向对象是什么意思。
- 面向对象设计和面向对象编程的不同。
- 面向对象设计的基本原则：
- 基本统一建模语言并且当它不是一个麻烦的时候。

面向对象

所有人都知道对象是什么：某种我们可以感知、触摸和操纵的有形的东西。我们接触到的最早的对象是典型的婴儿玩具。木质积木、塑料形体及过M的拼图都是常见的第一对象。婴儿可以很快地学到，特定的对象》丨以做特定的事情。角形物体能准确地放到三角形的孔里。铃可以响、按钮可以按，以及手柄可以扳。

在软件开发中，对象的定义也没有太大的差别。对象通常不是你可以捡起、感知或者触摸的有形的东西，而是这些东西的模型，这些模塑可以做特定的事情并且特定的事情可以作用于它们身上。正式地讲，对象是一个数据以及相关行为的集合。

现在我们知道了什么是对象，那么什么是面向对象呢？面向简单的意思就是指向。所以面向对象简单的意思就是“功能上指向建模对象”。通过数据和行为来描述交互对象的集合，这是用于对复杂系统建模的众多技术之一。

如果你读过仟何的宣传资料，那么你可能已经碰到了面向对象分析、面向对象设计、

面向对象分析设计以及丨w向对象编程这些术语。在一般的面向对象范畴里，这些都是高度相关的概念。

事实上，分析、设计和编程是软件开发的整个过程。称它们是面向对象是简单指定了软件开发所追求的风格

面向对象分析（OOA)是着眼于一个问题、系统或者任务的过程，一些人想要把这个过程变成一个应用，并且识别这些对象以及对象间的交互。分析阶段全是关于需要做些什么的。分析阶段的输出是一套盖求ˆ如果我们想一步就完成分析阶段.我们需要把一个任务，比如“我需要一个网站”，变成这样一套需求：

访问这个网站的人需要可以做以下事项（以楷体字代表行为，黑体代表对象）：

- 回顾历史。
- 申请工作。
- 浏览、比较以及订购我们的产品。

面向对象设计（OOD)足把k面这些需求转化为实践规范的过程。设计者必须命名这些对象、定义行为并且正式地指定这些对象可以作用于其他对象的具体行为。设计阶段全是关于事情应该如何做的:,设计阶段的输出是实践规范。如果我们想一步就完成设计阶段，我们需要把需求转化成一套类和接n,这些类和接口可以通过任何（理想情况）面向对象编程语言来实现。

面向对象编程（**OOP**)是把这个完美定义的设计转化成一个可以**T**作的程序的过程，这个程序恰好做了 CEO最初要求做的事情,[5]

是啊，没错！如果这个世界符合我们的想象，并且就像旧教科书告诉我们的那样，可以并井有条地一步接一步遵循这些阶段，那就太美好了。通常，现实世界要残酷得多。不管我们多努力地区分这些阶段，我们总是发现当我们准备设计的时候，事情还需要进一步分析。当我们开始编程的时候，发现在设计中，一些特性需要澄清。在这个快节奏的现代世界，大部分开发刚好是一个迭代式开发模型。在迭代式开发中，会先模块化、设计和编程实现任务中一个小的部分，然后评估这个程序并且通过扩展来改善每一个特性，包括在一系列短的周期里增加新的特性。

这本书剩下的内容是关于面向对象编程，但是在本章，我们会讲解在设计背景下一些基本的面向对象原则[3]这将会让我们理解这些相肖简单的概念，并且无须争论软件语法或者解折器。

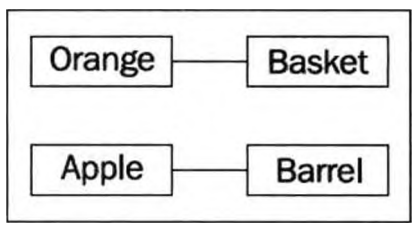
对象和类

因此，一个对象是一个有着相应行为的数据的集合。我们如何区分两类对象呢？苹果和橘子都是对象，但是它们没法比较却是一个常识。在计算机编程中，很少会建模苹果和橘子，但是假装我们正在为一个水果农场做一个货物清单的应用，作为一个例子，我们假设苹果会放到桶里，橘子会放到篮子里。

现在我们有4种对象：苹果、橘子、篮子、桶。在面向对象建模中，用于各种对象的术语叫作类。所以从技术术语来讲，我们现在有4个对象的类。

对象和类有什么不同呢？类描述了对象。它们就像是创建对象的模板。可能会有3个橘子放在你面前的桌子上。每一个橘子都是一个不同的对象，但是所有这3个对象却有着关联同一个类的属性和行为：橘子的通用类。

可以用统一建模语言 Unified Modeling Language ）(因为通过首字母缩写非常酷，一般简称UML ）类视图，来描述在我们货物清单里的4个类的对象之间的关系。下面是我们的第一个类视图：



这个视图简单地展示了一个橘子以某种方式和一个篮子有关系，一个苹果以某种方式和一个桶有关系。关联是把两个类联系起来最基本的方式。

UML这个T. 具在管理者之间非常流行，但偶尔会被编程人员蔑视。一个UML视图的语法通常是相当明显的；你通常（大多数情况）不需要通过去阅读一个指南来理解你看到的UML的意思是什么 _UML也是非常简单和直观的。毕竟，很多人在描述类以及它们之间关系的时候，会自然而然地画一些用线连起来的方框。对于这些直观的视图有一个标准，会让程序员和设计者、老板，以及其他程序员之间交流起来非常简单。

但是，一些程序员认为UML是浪费时间。以迭代式开发为例，他们会争辩说，在实施之前，这些正式的规范以花哨的UML视图来实现是多余的，并且维护这些正式的视图只会浪费时间，不会有任何ˆˆ处。

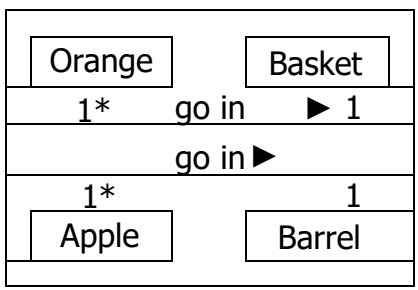
在一些组织机构里这是正确的，在一些其他公司的文化里这就是猪食。然而，每一个编程团队由不止一个人组成，偶尔需要坐下来并且讨论解决他们正在工作的系统的部分细

节，在这些头脑风暴会议中，为了快速和容易沟通，UML是非常有用的。即使那些嘲笑这种正式的类图的组织机构，在他们的设计会议或者团队讨论里，也倾向于使用一些非正式颁布的UML。

此外，你需要交流的最重要的人是你自己，我们都认为可以记住我们做出的设计决策，但是总是会有“为什么我要这么做？”的时刻，隐藏在我们的将来。当我们开始设计的时候，如果我们保存了写有我们初始设计图表的纸片，我们最终会发现它们是一个有用的参考。

然而，本章并不是想成为一个UML的教程。在互联网t有很多材料，以及众多相关话题的书籍可用。UMLM盖的远比类和对象阁多；它也有语法、部署、状态变化以及活动的用例。在这个面向对象的设计讨论中，我们会处理一些常陷的类视图的语法。你会发现，你可以通过例子获取结构，并且在你自己的团队或者个人的设计会议中，你会下意识地选择UML风格的语法。

我们最初的阉虽然正确，但是并没有提醒我们苹果可以放到桶里或若一个苹果时以放到多少个桶里。类之间的关系通常是明显并且不需耍进一步解释的，但是进一步澄清的选项总是存在的。UML的美妙之处在于大部分的东西是可选的。我们只需要在一个丨¥|里指出尽可能多的信息，来使它在当前情况下是有意义的。在一个快速的白板会议上，我们可能只是快速地_一些条条框框。我们可能会在正式文档里涉及更多的细节，耑要6个月才有意义。在这个苹果和桶的例子里，我们可以相当确信的关系是，〞很多苹果可以放到一个桶里”，但是为了确保没布人将其与“一个苹果占据一个桶”相混淆，我们可以通过下面的图强调这一点：



这个阁传诉我们橘子放进篮子里，通过一个小箭头显示什么放进什么。它也告诉我们双方关系的多样性（可以关联在一起的对象的数量L 一个篮子可以放很多（通过一个*表示）橘子对象。任何一个橘子可以放到一个篮子里3

很容易混淆多样性是哪一边的关系。一个类可以通过关系中其他端的任何一个对象关联在一起，这个类的对象数M就表示多样性。对于苹果放进桶这个关联，从左读到右，很

多个苹果类的实例（很多苹果实例）可以放进任何一个桶里。从右往左读，就是一个桶可以与任何一个苹果关联。

指定属性和行为

现在我们已经掌握了些基本的面向对象术语。对象是类的实例，可以彼此相互关联a一个对象实例是一个带有它自己数据集合和行为的特定对象；一个特定的橘子放到我们面前的桌子上，说明它是一般的橘子类的一个实例。这非常简单，但是，什么是与每一个对象关联的数据和行为呢？

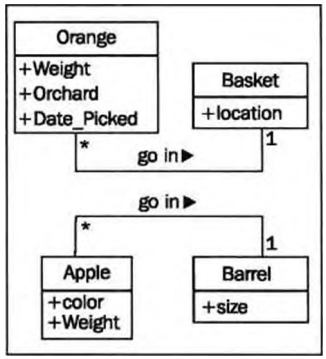
数据描述对象

让我们先从数据开始。数据通常代表了一个给定对象的个体特征。一个类的对象可以定义这个类的所有实例所共享的一些特定特征。对于给定的这些特征每一个实例可以给它们不同的值。例如，放在桌子上（如果我们还没有吃掉它们）的3个橘子可能各有不同的重Jfl。橘子类可能会有一个重量的属性。橘子类的所有实例都有一个重量属性，但是每一个橘子的重量可能会不同。属性不必是唯一的，任何两个橘子可能重量是相同的。作为一个更现实的例子，代表不同客户的两个对象可能第一个名字属性的值相同。

属性通常称为特征_。一些作者认为这两个词具有不同的含义，通常属性是可变的，特征是只读的。在Python中，“只读”的概念并没有真正使用，所以在本书中我们将会看到两个术语交叉使用。此外，在第5章中，对于一类特殊的属性，特征关键词在Python中有特殊的意义。

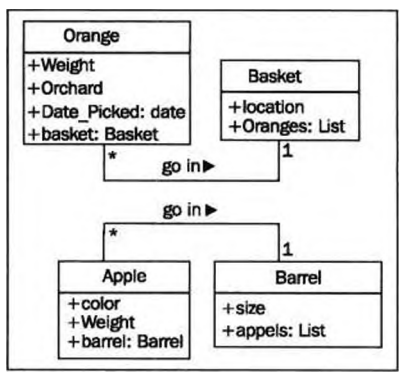
在我们的水果清单的应用程序里，果农可能想知道橘子是从哪个果园来的，什么时候采摘以及它多重。他们可能也想追踪每一个篮子都存放到哪里。苹果可能会有一个颜色的属性，同时桶可能会有不同的大小。其中一些属性也可能属于多个类（我们也可能想知道苹果是什么时候采摘的），似是对于第一个例子，让我们只是给类图添加一些不同的属性：

[1] **property**和**attribule**都有域性的总思在编裡丨•中.**p，p**$_{crt}$**y**通常指对私钉变丨丨(的丨力问控制.包含**g**$_{etter}$和$_{seUer}$方法；auribine通常指一个类中的数据成记本文中，在编程语言相关的部分.aiiribmc都译作诫性.property根据语境译为特征.属性.property丨4性，或不择.————择者注



根据我们设计所需的具体程度，我们也可以给每一个属性指定类型。属性类型通常是多数编程语言的标准说法，如整型、浮点型、字符串字节或者布尔型。然而，它们也可以代表一些数据结构，像列表、树或者图，或者最值得注意的是其他的类。这也是一个设计阶段和编程阶段可以重叠的领域。在一种编程语言里可用的对象或者基本类型可能和其他编程语言里可用的对象或基本对象有些许不同。通常在设计阶段我们不需要关心这些，因为在编程阶段会选择具体实现的细节。使用通用的名字就可以了。如果我们的设计需要一个列表容器类型，当实现的时候，Java程序员可以选择使用LinkedList或者ArrayList，而Python程序员（就是我们！）可以选择内置的list或者tuples

在我们的水果种植例子中，到目前为止，我们的属性都是一些基本的原始类型。但是也有我们可以明确的隐式属性：关联。对于一个给定的橘子，我们可能有一个属性，就是哪个篮子里放着这个橘子。另外，一个篮子里可能会放很多个橘子。下面这张图，就像对我们目前这些特征进行的类型描述一样，会把这些属性添加进去：



List是列表，Tuple是元组，这两个都是Python里的基本数据类SL——译者注

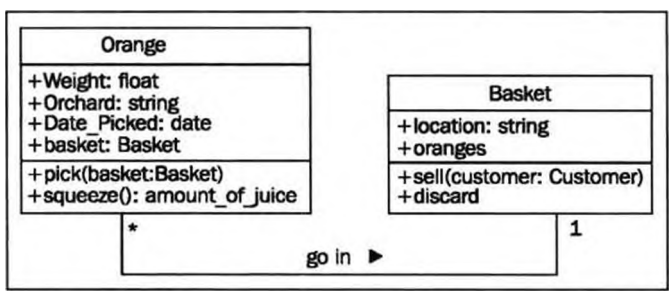
行为是动作[3]

现在我们知道了什么是数据，但是什么是行为呢？行为是可以发生在对象身上的动作。一个特定的类的对象里可以执行的行为被称为方法。在编程层面，方法就像结构化编程里的函数一样，但是它们可以神奇地访问和这个对象关联的所有数据。方法可以像函数那样接收参数并返问值

方法的参数是一个对象的列表，需要传递给被调用的方法。方法会使用这些对象来执行任何它想要做的行为或者任务。返回的值是这个任务的结果。这里有一个具体的例子；如果我们的对象是数字，数字类可能会有一个add方法，这个方法接收第二个数作为参数。当第二个数传人的时候，第一个数字对象的add方法会返回这两个数的和。给定一个对象和它的方法名，调用对象的时候也可以调用这个目标对象的方法。在编程层面匕调用一个方法就是通过传给它所必需的参数来使这个方法执行的过程。

在这个基本的清单应用里“比较苹果和橘子”，我们会遇到闲难。让我们进一步来扩展它并且看看是否能突破。一个可以和橘子关联在一起的动作就是摘。如果你想一下实现，摘就是通过更新橘子的篮子属性把橘子放进篮子，并且把橘子添加到篮子里的橘子列表。所以摘这个动作要知道它要处理哪一个篮子。因为果农也要销售果汁，我们可以给橘子添加一个挤压方法。当挤压的时候，挤压方法会返回获取的橘汁的量，同时也会把这个橘子从篮子里清除掉。

篮子可以有一个出售的行为。当一个篮子卖出去以后，我们的清单系统可能需要更新一些数据，这些数据是尚未指明的用于会计和利润计算的对象。或者.我们这篮子里的橘子可能会在卖出去之前坏掉，所以我们需要添加一个丢弃方法。让我们把这些方法添加到我们的图里：



给单个对象添加模块或者方法允许我们创建一个交互对象的系统。系统里的每一个对

这 | II原 Si 是 Behaviors are actions .-----译者注

象都是特定类的成员这些类指明了这个对象可以容纳什么样的数据类型以及什么方法可以凋用它。每一个对象的数据相比于同一类的其他对象可以有不同的状态，并且因为这个不丨FT],每一个对象对于方法调用的反应也是不同的$_3$

面向对象分析和设计就是找出这些对象是什么以及它们该如何交互。下一节描述了一些原则，这些原则可以用于使这些交互尽可能简单和直观a

隐藏细节并且创建公共接口

在面向对象设计中模块化一个对象的主要目的是决定该对象的公共接口:>接口是其他对象可以和该对象交互的属性和方法的集合。它们不需要且通常也不允许去访问对象的内部T作一个常见的真实世界的例子就是电视。我们电视的接口是遥控器。遥控器上的每一个开关代表了一个可以由电视对象调用的方法。当我们作为调用对象去访问这些方法的时候，我们不知道也不关心这个电视是从天线、电缆还是卫星天线获取信号的。我们不关心发送什么电子信号去调节音量，或者声音是输出到扬声器还是耳机。如果我们打开电视i方问它的内部T作，例如将输出信号分为去往外部扬声器以及耳机，我们的保修就失效了。

隐藏一个对象实现或者功能细节的过程，通常称为信息隐藏。它有时也被称为封装，但是实际h封装是一个更加包罗万象的术语。封装数据并不一定是隐藏。封装，夸张点说，是创建一个胶旗，所以想一下创建一个时间胶囊..如果你把一堆信息放到时间胶囊里，锁住并埋了它，它被封装并且信息被隐藏了。另一方面，如果没有把时间胶幾埋了并且没有上锁，或者胶•輿是用透明塑料制成的，里面的东西同样被封装了，但是信息没有被隐藏。

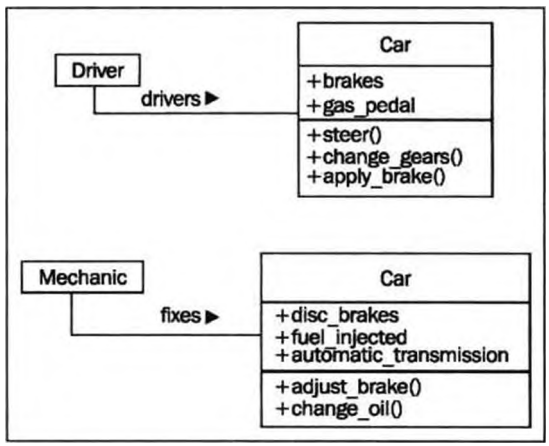
封装和信息隐藏的区别不是那么重要.特别是在设计层面。很多实际中的引用使用的术语是可替换的。作为Python程序员，我们不需要或者不必要真正的信息隐藏（我们在第2章里会讨论为什么），所以对封装更多的包装定义是合适的。

然而，公共接口是非常重要的。因为在将来很难改变它，所以需要精心设计。改变接口会导致中断任意客户对象对它的调用。我们可以随意改变内部机制，例如，让它效率更高，或者就像访问本地一样访问网络上的数据，客户对象无须改变就可以使用公共接口与之交互。另一方面，如果我们通过改变公共访问的属性名字，或者改变一个方法可以接收的参数名字或类锢来改变接口，那么所有客户对象也将必须修改。

记住，程序对象代表了真正的对象，但是它们不是真正的对象。它们是模型。模型化最好的天赋就是可以忽视一切不相关的细节。一个模■汽车外表可能看起来像一个真正的1956雷鸟，但是它不能跑并且传动轴也不转，因为这些细节过于复杂并且与年轻人组装模

型毫无关系。模型是一个真实概念的抽象。

抽象是另外一个和封装以及信息隐藏相关的面向对象词汇。简而言之，柚象意味着要对一个给定任务以最合适的水平来处理细节。它是一个把公共接口从内部细节里抽取的过程。一辆车的司机需要与方向盘、油门和刹车交互。而电动机、传动系统以及制动子系统的工作原理对于司机来讲是无关紧要的，另一方面，对丁机械师，优化引擎以及刹车就会有不同水平的抽象。下面是对一辆汽车在两个水平上的柚象：



现在我们有了一些指向相似概念的新术语。所有这些术语凝结成一句话就是，抽象就是通过把公共和私有接口分开而封装信息的过程。私有接口可以用于信息隐藏。

所有这些定义带来的最重要的事情就是使得与我们模沏交互的对象能够理解我们的模型。这就意味着要注意小细节。确保方法和属性有合适的名字„当分析一个系统的时候，在原始问题里对象通常代表名词，而方法通常代表动词。属性通常可以作为形容词，尽管如果属性指向了属于当前对象一部分的另一个对象，它仍然坷能是一个名词。相应地命名类、属性和方法 不要试图给将来可能有用的对象或者行为建模。准确地建模那些系统需要执行并a设计的任务，建模自然而然会走向一个合适的抽象层次。这并不是说我们不应该考虑未来可能的设计修改。我们的设计应该以开放式结尾，这样可以满足未来的耑求，然而，当抽象接口的时候，试着准确地建模那咚需要建模的而不要管其他。

在设计接口的时候，试着把自己放到对象里并且想象这个对象对隐私有很强的偏好，不要让其他对象可以访问你的数据，除非你认为这样做对你有最大的好处$_{S}$不要通过给它们一个接口来强制你肉己去执行一个特定的任务，除非你确定你希望它们可以这么做。

对于确保你的社交账户隐私，这也是一个很好的实践。

组合和继承

到目前为止，我们已经学会了把系统作为一组交互式对象来设计，每一个交互都是在适当的抽象层次上查看对象。但是我们尚不知道如何创建这些抽象层次。做这个有很多方法；我们会在第8章和第9章讨论一些高级的设计模式。但是大多数设计模式都会依赖两个基本的原则，称之为组合和继承。

组合是把一些对象收集在一起组成一个新对象。当一个对象是另一个对象的一部分的时候，组合是一个好的选择。在机械师的例子里，我们已经看到了第一个组合的暗示。一辆汽车是由引擎，传动、启动器、车灯和挡风玻璃等很多其他部件组成的。接下来，引擎是由活塞、曲轴以及阀门组成的。在这个例子里，组合是提供抽象层次非常好的方式。汽车对象可以提供给司机所耑的接口，同时提供/访问它的组成部件的能力，这个能力为机械师提供了一个更深的抽象层次。当然，如果机械师需要更详细的信息来诊断问题或者优化引擎，这些组成部分可以进一步分解。

这是关于组合的第一个常见例子，但是当它用于设计计算机系统的时候，不是一个好的例子。物理对象很容易分解成组件对象。至少从古希腊人最初假定原子是物质的最小单位开始，人们已经这样做了（当然，他们没有使用过粒子加速器）。计算机系统通常没有物理对象那么复杂，然而在这样的系统里识别部件对象也不会很自然地发生。在一个面向对象的系统里，对象偶尔代表物理对象，比如人、书或者电话。然而更常见的是，它们会代表抽象的概念。人有名字，书有标题，电话是用来拨打的。电话、标题、账号、名字、预约以及支付，通常不认为是物理肚界中的对象，但是在计算机系统里它们经常被建模为组件。

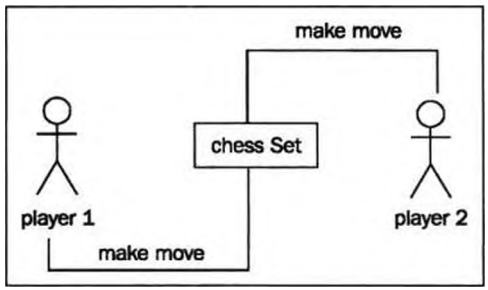
让我们通过建模一个更面向计算机的例子来看看实际中的组合。我们将看一下一个计算机象棋游戏的设计。这是一个19世纪80年代和90年代学术界非常流行的消逍游戏。人们预测有一天计算机能够击败人类的象棋大师。当这发生在1997年时（丨BM的深蓝击败了象棋冠军加里•卡斯帕罗夫），尽管仍有计算机和人类象棋手之间的竞赛，并且程序没有达到能够在所有时间内都打败人类象棋大师，但是在这个问题上人类兴趣索然。

作为一个基本的、高层次的分析：象棋游戏是两个选手之间玩的，在一个8X8的网格里包含64个位置的棋盘上使用一副棋子。棋盘有可以移动的两套16个棋子，两个选手交替以不同的方式移动。每一个棋子可以吃掉其他的棋子。每一次移动，棋盘都要在计算机屏幕上画出它自己。

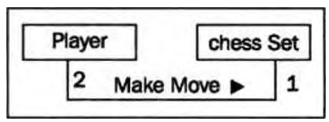
在描述中，我已经通过使用楷体字确定了一些可能的对象，以及使用黑体字确定了一些关键方法：在一个把面向对象分析变成设计的过程中，这是常见的第一步。在这一点上，

为了强调组合，我们将关注于棋盘，而不会关心太多有关玩家或者不同类型棋子的问题.

让我们从可能的最高层抽象开始。我们有两个玩家，通过轮流移动一副棋子来做交互。



那是什么东西？它看起来不像我们之前的类图。这是因为它根本不是一个类图！这是一个对象视图，也称为一个实例图。它描述了一个特定状态、特定时间的系统，描述的是对象的特定实例，而不是类之间的交互。记住，这两名玩家是同一个类的成员，所以类图看起来会有点不同。



这个图确切地显示了两个玩家可以和一套棋子做交互。它也表明，任何玩家一次只能玩一套棋子。

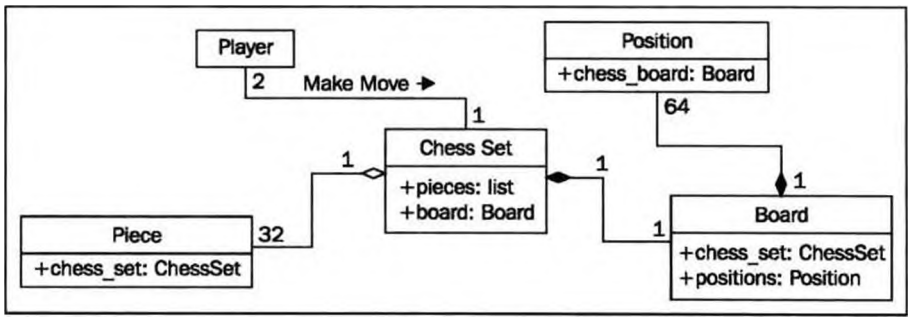
但是我们要讨论组合而不是UML,所以让我们想一下象棋是由什么组成的„当前我们不关心玩家的组成„我们可以假设在玩家的其他器官中有心脏和大脑，但是这和我们的模型无关。事实上，没有人禁止该玩家是深蓝自己，深蓝没有心脏也没有大脑。

然后，象棋是由一个棋盘和32个棋子组成的。棋盘进一步由64个位置组成。你可能会争论说，棋子不是象棋的一部分，因为你可以在象棋里用其他一副棋子把棋子替换掉。虽然在一个计算机版本的象棋里，这是不可能的，但是它给我们引人了聚合。聚合几乎和组合一样$_3$不同的地方是，聚合对象可以独立存在。把一个位置和不同的象棋棋盘相关联是不可能的，所以我们说，象棋棋盘是由位置组成的。但是对于棋子，它可以独立于象棋而存在，而被认为和一套象棋是一种聚合关系$_3$

另外一种区分聚合和组合的方法是考虑对象的寿命。当（内部）创建和销毁相应对象的时候，如果组合（外部）对象控制着它们，那么组合就是最合适的。如果可以独立于组合对象去创建相关对象，或者可以对比对象，一个聚合关系可能会有意义；聚合是组合的

一个更一般的形式。任何组合关系同时也是聚合关系，但也不尽然。

让我们描述一下我们目前的象棋组合，并且给对象添加一些属性来保持组合关系：



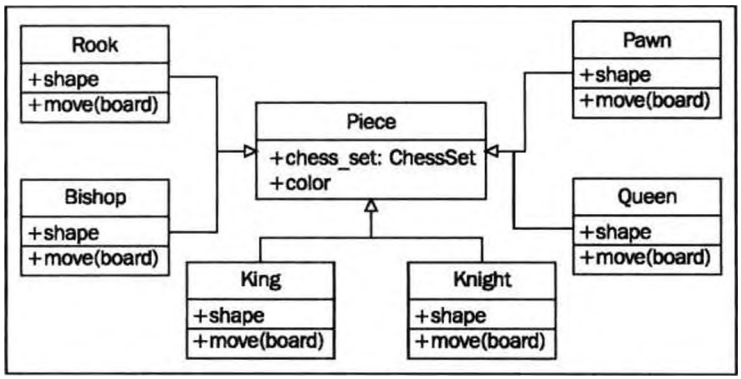
在UML图里，一个实心菱形代表了组合关系。空心菱形代表了聚合关系。你会注意到棋盘和棋子作为象棋的一部分来存储，它们的引用也以同样的方式作为一个属性存储在象棋里。在实践中，再次表明一旦迈过设计阶段，聚合和组合的不同通常是无关紧要的。当实现的时候，它们几乎也表现得一样。然而，当你的团队在讨论不同对象如何交互时，它可以帮助你们区分它们的不同。通常你可以把它们当成同样的东西对待，但是，当你需要K分它们的时候，也要知道它们的不同（这是在丁_作中的柚象）^

继承

我们已经讨论了对象之间3种类型的关系：关联、组合和聚合。但是我们没有完全定义我们的象棋，并且这些丨:具似乎并没有给我们所需要的能力。我们讨论过玩家有可能是人或者具有人工智能的软件的可能性。所以说一个玩家和一个人关联，或者人工智能是玩家对象的一部分，看起来都不正确。我们真正需要做到的是能够说“深蓝是一位玩家”或者“加里•卡斯帕罗夫是一位玩家”。

这是一种由继承形成的关系。继承是众所周知最著名的用于面向对象编程的一种关系。继承有点像家庭树。我爷爷的姓是菲利普，我父亲继承了这个姓，我又从父亲那里继承过来（同时还有蓝色的眼睛以及写作的嗜好在面向对象编程中，一个人除了有继承特性和行为，一个类也可以从其他类继承属性和方法。

例如，在我们的象棋中有32个棋子，但是只有6种不同的类型（兵、车、象、马、王和后），每种类型当它们移动的时候会有不同的行为。所有这些棋子都有如颜色、属于哪副棋的厲性，但是它们在棋盘上也有独特的形状、不同的移动。看一下这6种棋子如何从一个Piece类继承：



当然，空心箭头指明从Piece类继承来的单个棋子类。所有子类都会ft动从基类继承一个chessjset和coloi•厲性。每一个棋子提供了一个不同的形状属性（当渲染棋盘时会在屏幕上丨Bi出这个形状），以及一个不同的move方法，每一轮在棋盘上移动这个棋子到一个新的位置。

我们实际上知道所有Piece的子类都需要一个move方法，否则当棋盘试图移动棋子的时候会感到闲惑。有可能我们想要创建一个新版本的象棋游戏，它有一个额外的棋子（向导）。目前我们的设计允许我们不崙要给它一个move方法就能设计这个棋子。当棋盘要求棋子移动它自己的时候，棋盘就死掉了。

我们可以在Piece类里创建一个虚拟的移动方法来实现。子类可以用更具体的实现来重写这个方法。例如，默认的实现可能会弹出一个错误消息，提示这个棋子不能移动。在子类里重写方法允许实现强大的面向对象的系统，例如，如果要实现一个带有人T：智能的玩家类，我们可能需要提供一个叫$calcul_at_{e_}m_{\bigcirc}v_e$的方法，这个方法接收一个棋盘对象，然后决定哪个棋子要移动到哪里去。一个非常基础的类可能会随机地选择一个棋子和方向，然后移动。然后我们可以在子类里重写这个方法，来实现深蓝。第一个类可能比较适合跟初学选手比赛. 其后的可以挑战一位大师。重要的是，这个类的其他方法，比如通知棋盘已选择的移动，并不需要改变；这些实现可以在两个类之间分享。

考虑到象棋棋子，给它提供一个默认的移动方法其实没有什么意义。我们只需要在任意子类里指定它需要的移动方法，这可以通过使Piece成为一个抽象类，并且卢明一个抽象的移动方法实现。抽象方法基本会说“我们在子类里需要这个方法，但是我们拒绝在这个类里指明一个实现方法”。

事实上，创建一个没有实现任何方法的类是可能的。这样一个类会简单地告诉我们这个类该做什么，但是绝对没有提供任何建议告诉你如何去做。在面向对象的世界里，这样

的类称之为接口。

继承提供了抽象

现在是时候讨论另外一个长流行词了。多态是依据一个类的不同子类的实现而区别对待这个类的能力。我们已经在我们描述的象棋系统里实际看到过它了。如果我们想让设计走得远一点，我们可能会看到棋盘对象可以从玩家那里接收一个移动请求，并且调用这个棋子的move函数。棋盘不需要知道与之交互的棋子是什么类型的。它所需要做的只是调用move方法，然后适当的子类会去考虑是要按照车还是兵移动它。

多态非常酷，但是它是一个在Python编程中很少使用的字眼。Python更进了一步，它允许像父类那样对待一个对象的子类。在Python中一个棋盘的实现可能会让任何一个对象具有移动的函数，不管它是象、车或者鸭子。当调用move的时候，象会在棋盘上斜着走，车会开到一个地方，而鸭子根据自己的心情可以游泳或者飞。

在Python中这种多态性通常称为鸭子类型：“如果它像鸭子一样走路，像鸭子一样游泳，那么它就是一只鸭子”。我们不关心它是否真的是一只鸭子（继承），只要它可以游泳或者走路。鹅和天鹅很容易能够提供我们正在寻找的像鸭子那样的行为。这允许未来的设计师可以创建新种类的鸟，而不需要真正指定水生鸟类的继承层次结构。这也允许他们创建完全不同的插人式行为，这些行为传统的设计师从没有计划过。例如，未来的设计师可能可以创建一个能走路游泳的企鹅，它具有和鸭子相同的接口而不需要任何建议。

多重继承

当我们在自己的家庭树上考虑继承的时候，我们可以看到我们不仅仅从一个父亲那里继承特性[3]当陌生人告诉一位骄傲的母亲，她的儿子“有他父亲那样的眼睛”时，她通常将回应说，“是的，但是他有我的鼻子”。

面向对象设计同样可以有类似多重继承的特性，这允许一个子类从多个父类那里继承功能。在实践中，多重继承是比较棘手的事情，并且有些编程语言（最明显的就是**Java**）严格禁止它。但是多重继承也有它的用处。多数情况下，它可以用来创建一个具有两组不同行为的对象。例如，一个对象被设计用于连接扫描仪，并且把扫描文件发送传真出去，这种情况下，对象可以从两个单独的**scanner**和**faxer**传真机对象那里继承过来。

只要两个类有不同的接口，对于一个从这两个类继承过来的子类通常来讲是无害的。但是如果我们从两个提供重叠接口的类继承就会变得一团糟。例如，如果我们有一个摩托车类，它奋一个**move**方法，并且还有一个船类，它也有一个**move**方法，然后我们想把

它们聚合成一个两栖车辆. 当我们调用move方法的时候，这个合成类如何知道该怎么做？在设计层面，这需要解释，在实现层面，每种编程语言都有各自不同的方式来决定该调用哪一个父类的方法，或者以什么样的顺序去调用。

通常，处理这个的最好方法就是避免它。如果你有一个像这样的设计，你可能是做错了。退一步，再分析一下系统，看看是否可以把多重继承给删掉，而采用其他类似关联或者组合设计。

为了扩展行为，继承是一个非常强大的工具。它也是早些时候面向对象设i卜最激动人心的进步之一。因此，它通常是面向对象程序员拿到的第一个1：具。然而，认识到拥存一把锤子并不能把螺丝变成钉子是很重要的。对于明显的“是一个[4]”这样的关系来说，继承是完美的方案，但是它可以被滥用。程序员通常在两种对象间使用继承来分享代码，而这两个对象没有直接关系，看到的不是″是一个”关系。这未必是一个糟糕的设计，而是一个非常好的机会，问一下他们为什么这么设计，以及是否不同的关系或者设计模式更合适。

案例学习

让我们通过几个现实世界中迭代的面向对象设计例子来把我们新的面向对象知识联系到一起。我们将要建模的系统是一个图书馆目录。图书馆通过使用卡片目录来追踪它们的藏书已经儿个世纪了，然而最近电子藏书更流行。现代图书馆有一个基于Web的目录，我们可以在A己家里查询藏书$_3$

让我们开始分析。当地图书馆要求我们写一个新的卡片目录程序，因为他们古老的基于DOS的程序很难用并且过时了$_3$这并没有给我们太多细节，但是在我们开始要求更多信息之前，让我们想一下我们已经知道的关于图书馆目录的知识。

目录包含书的列表人们搜索这个目录来寻找特定主题、特定标题，或者特定作者的书$_3$书可以通过一个唯一的国际标准书号 ISBN ）区分。每一本书都有一个杜威十进制数(DDS ）来帮助我们在特定书架上找到它。

这个简单的分析告诉丫我们在这个系统里的一些明显的对象。我们快速地识别出书是最重要的对象，它有一些前面已经提到的属性，比如作者、标题、主题、 I SBN、DDS号及目录，作为书的一种管理方式。

我们也注意到有一些其他对象可能需要也可能不需要在系统里建模。出于编写目录的

[4]所if彳“是•个″这种关系. 就坫A是一个B这种关系可以丨丨丨宋简单判断是否可用继承. 比如苹果坫一种水么苹果类就可以继承水果类——译#注

R的，所有我们需要的就是通过一本书里的auth○r_name属性来搜索一本书。但是作者也适对象，并且我们可能需要存一些关于作者的其他数据。当我们考虑这个问题的时候，我们″J■能还记得有些书的作者不止一个。突然间，对象只有一个author_name属性的想法看起来似乎旮点螽，与每本书关联一个作者列表显然是一个更好的主意。作者和书的关系很显然是关联，W为你绝对不会说“书是一个作者”（它不是继承关系），并且说“书有一位作者”，虽然语法楚对的，但是这并不意味着作者是书的一部分（这不是聚合关系）。事实上，任何一位作者都荇吋能跟多本书相关联。

我们应该注意名词（名词总是好的候选对象）书架。在一个目录系统里，书架是一个耑要建模的对象吗？我们如何识别一个书架。如果有一本书存放在一个书架末尾，然后因为另一本书插人到这个B架1:而导致之前那本书移动到下一个书架的开头，这会发生什么？

DDS &在帮助我们在图书馆里定位实体书架这样，给书存储一个DDS属性就应该足够找到这本丨5,而不管它放到f哪个书架上。所以至少P前可以把书架从我们的竞争对象列表里删除了。

在这个系统里 外一个可疑对象是用户。我们需要知道关于一个特定用户的任何信息吗？他们的名字、地址，或者过期的图书列表？到目前为止，只有图书管理员告诉我们他们想要一个目录；关于追踪汀阅或者逾期通知他们并没有说什么。在我们的脑海里，我们也汴意到作者和用户都足特定类型的人；将来这里可能会有一种有用的继承关系。

至于I丨录的H的，丨1前我们决定我们不需要识别用户。我们可以假设一个用户将会搜索目录，但是除r提供一个允许他们搜索的接口外，我们不必要在系统里给他们建模:，

我们已经确定了儿个关于书的厲性，但是目录会有什么属性？任何一个图书馆都会有多个0录吗？我们WfT要唯-地识别区分它们吗？显然，S录有一个它所包含的书的列表，但是在某种程度上，这个列表可能不是公共接口的一部分。

行为呢？很显然目录需要一个搜索方法，可能会区分作者、标题及主题。书有什么行为吗？它耑要有一个预览的方法鸣？或者预览可以通过第一页属性来表示，而不是一个方法？

前面〖、丨论的问题都是而向对象分析阶段的一部分。但是混杂着这些问题，我们已经确定了设汁的部分关键对象事实上，你刚才#到的是分析和设计之间的微观迭代。有可能这些迭代会发生在刚开始的图书管理员会议上。然而在这个会议之前，我们已经为我们具体识别的对象勾勒出了一个最基本的设计：

Catalog
+Search○

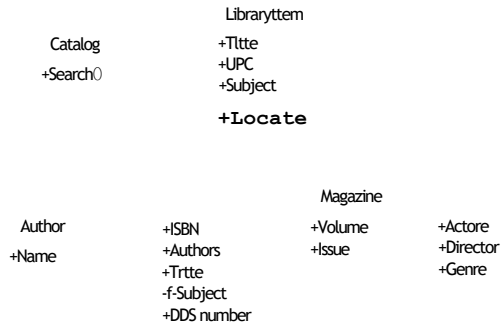
Author

+ISBN
+Authors
+Title
+Subject
+DDS number

我们和图书管理员见面，带着这张基本图以及一个铅笔来交互式地改善它。他们告诉我们这是一个好的开始，但是图书馆并不只摆书，他们也有DVD、杂志及CD, 这些都没有ISBN或者DDS号。然而所有这些类型的东西只能通过一个UPC[5]号来唯一标识。我们提醒管理员，他们必须在书架上找到这些东西，并且这些东西可能没有按照UPC组织分类。图书管理员解释说，每种类型的组织方式不同。CD大多数是有声读物，只有几打库存，所以它们按照作者的姓来组织分类。DVD分为不同类型并且进一步通过标题来组织分类。杂志通过标题以及精确的卷和发行号组织分类。书，好像我们已经猜到了，是通过DDS号组织分类的。

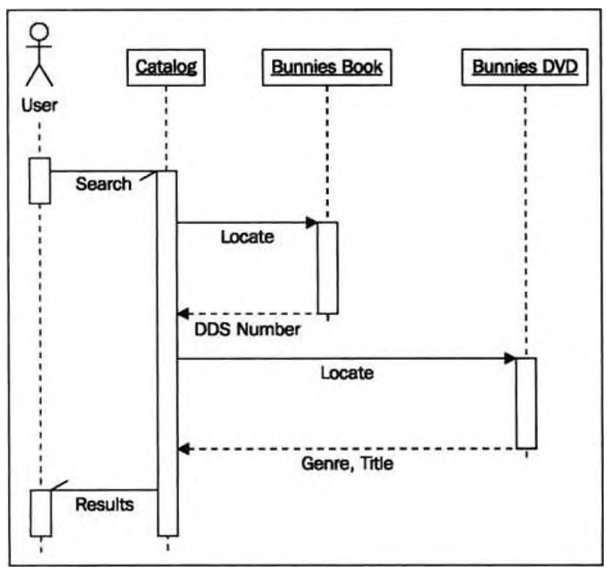
由于之前没有面向对象设计的经验，我们可能会考虑给DVD、CD、杂志和书在我们的目录里添加单独的列表，并且轮流搜索每一项。问题是，除了特定的扩展属性以及标识这些东西的物理位置以外，这些东西的行为方式都是一样的。这就是一个继承T. 作！我们快速地更新下我们的UML图：

[5]通用产品代码 Universal Produci Code). 通常简称UPC码. 是关国统一编码协会制定的一种商品条码•主要在J6丨*i以及加拿大使ML——译者注



图书管理员需要理解我们画的阁的要点，但是对于定位功能有一些疑惑。我们通过一个特定的用例来解释，有一位用户想要搜索“小兔子”这个词。用户首先给检索目录发送了一个搜索请求。目录查询其内部的项目列表，并且找到在标题里带有“小兔子”的一本书和一个DVD。这时，B录并不关心它找到的是DVD、书、CD或者杂志；对目录而言，所有的东西都是一样的。但是用户想知道如何找到物理的东西，如果目录只返回一个标题列表就有些疏忽了。所以它对找到的两个东西调用了定位方法。书的定位方法返回了一个DDS号，这个号可以用于找到放着这本书的书架。DVD通过返回类型和标题来定位。然后用户就可以问DVD部分，找到包含这个类型的部分，并且通过标题找到具体的DVD。

就像我们解释的，我们勾勒出了一个UML序列图来解释不同对象是如何通信的：

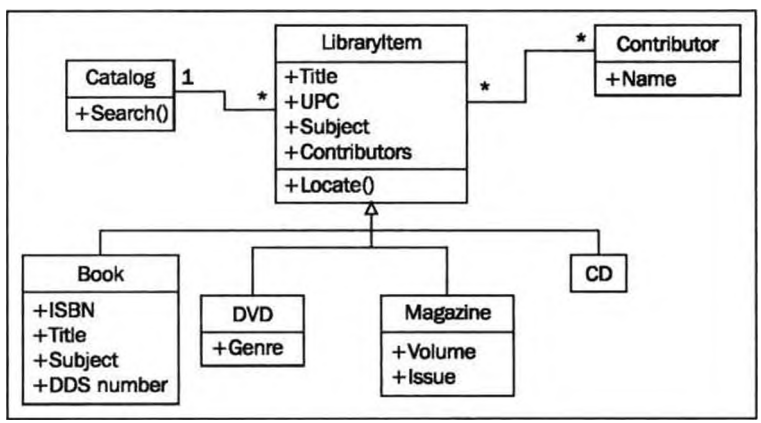


类图描述类之间的关系，序列图描述了对象之间的特定的消息传递。从每个对象出来的虚线是生命线，它描述了对象的生命周期。每个生命线上的宽方格代表了这个对象上的有效处理（没有方块的对象基本是闲置的，在等待事件发生）。生命线之间的水平箭头显示特定信息。实箭头表示被调用的方法，虚线箭头代表方法返回的值。半箭头表示发生或者从一个对象返回的异步消息。一个异步消息通常意味着第一个对象调用了第二个对象的方法并立即返回。经过一些处理，第二个对象调用第一个对象的一个方法来给它一个值。这和正常的方法调用相反，正常的方法调用会在方法内部做处理，然后立即返回一个值。

像所有的UML图一样，序列图最好在需要的时候使用。为了画一个图而画一个UML图是毫无意义的。但是当你需要在两个对象间交互通信的时候，序列图是一个非常好的工具。

不幸的是，目前我们的类图仍然是一个混乱的设计。我们注意到DVD里的演员 CD里的艺术家都是各种类型的人，但是却和书的作者区别对待。图书管理员也提醒我们，他们大多数CD是音频书籍，它们有的是作者而不是艺术家。

我们如何处理贡献一个标题的不同类型的人？ 一个明显的实现就是创建一个Person类，带有人的名字以及其他相关细节，然后为艺术家、作者及演员创建子类。但是这里真的需要继承吗？为了搜索和检索B录的目的，我们不在乎表演和写作是两个完全不同的活动。如果我们在做一个经济模拟，分别给演员和作者单独一个类以及不同的calculate_income和perform_job方法是有意义的，但是为了检索目录的目的，知道这个人贡献了这个东西可能就足够了。我们认识到，所有的东西都有一个或者多个贡献者对象，所以我们把作者关系从书中移到了它的父类里：



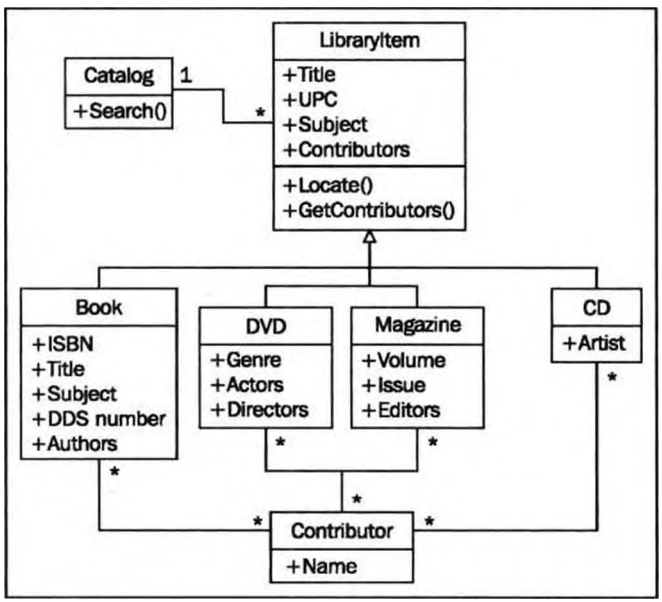
Contributor/Library | tem关系是多对多的多样性关系，就像在每一个关系的结尾通过*

表示一样。任何一个图书馆项目可能不止一个贡献者（例如，一个DVD视频中有多名演员和一名导演）。并且很多作者写了很多书，所以他们会被附到多个图书馆项目里。

这个小改变，虽然看起来更加简洁和简单，但是丢失了一些重要信息。我们仍然可以说出那些贡献一个特定图书馆项目的人，但是我们不知道他们是如何贡献的。他们是导演还是演员？他们写了音频书，还是用声音叙述了那本书？

如果我们可以给Contributor类添加一个Contributor_type属性就好j″，但是当处理那些既写书又导演了电影的多才艺的人时，这将会崩溃。

有一个选择是给我们每一个Libmryltem子类添加属性，它会保存像书的作者、CD的艺术家这种我们需要的信息，然后让这些属性都指向Coiitributoi•类。这样做的问题是，我们失去了优雅的多态。如果我们要列出一个项目的贡献者，我们需要在这个项目里查找特定的属性，就像作者或者艺术家。我们可以通过给LibraryUem类添加一个GetContributors方法来实现，Libranylten类的子类可以重写这个方法。然后目录永远不会知道对象正在查找什么属性；我们抽象了的公共接口：

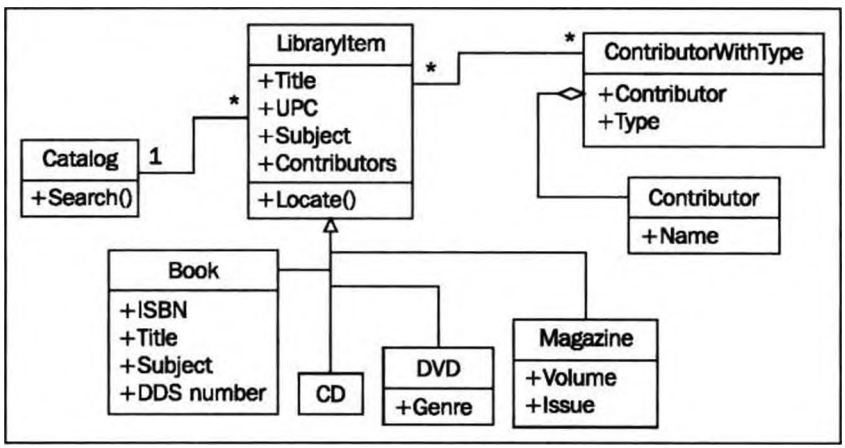


看着这个类阁，感觉我们好像做错了什么。这很笨重和脆弱。它可能会做我们需要的一切，但是感觉很难维护或者扩展。这里有太多的关系，修改任何一个类，太多的类会受到影响。它看起来像意大利面条和肉丸子。

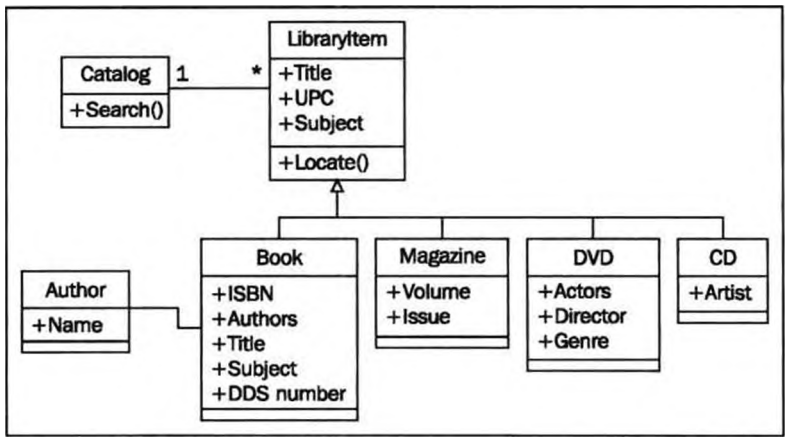
既然我们已经讨论了继承作为一个选项，并且发现它想要成为这个选项，我们可能要

回顾下我们之前的基于组合的图表，那里的Contributor直接附在Library | tem上[a]再想一想，我们可以看到我们其实需要给全新的类再添加一个关系来区分贡献者的类型。在面向对象设计中这是一个很重要的步骤。现在我们给我们的设计添加一个类，旨在支持其他的对象，而不是给初始需求的任何部分建模。我们正在重构设计，以便于帮助系统中的对象而不是现实中的对象。重构是维护或设计一个程序的重要过程。重构的目的是通过移动代码，删除重复代码或者复杂关系来改善设计，有利于形成一个更简单、更优雅的设计。

这个新类由一个Contributor和额外的识别给定Library | tem贡献者的属性组成[b]对于一个特定的Library | tem可以有多个贡献者，一个贡献者可以以同样的方式贡献不同的项目。这个图传达设计得非常好：



起初，这个组合关系看起来不如基于继承的关系那么自然。但是它的优势是允许我们添加新类型的贡献者，而不需要给我们的设计添加新的类。当子类有一些专业性的东西时，继承是最有用的。专业性是在子类里创建或者改变属性或者行为，来使它和父类有所不同。为了识别不同种类的对象，创建一堆空类看起来很愚蠢（在Java和其他“一切都是对象”的程序员看来，这种属性不是很流行，但是对于更实际的Python设计者看来这很普通）。如果我们看看继承版本的图，可以看到一堆实际上不做任何事情的子类：



当不使用面向对象原则时，有时候意识非常重要。什么时候不要用继承的例子是一个很好的提醒，提醒我们面向对象只是丁.具，不是规则。

练习

这个一本实用的书，而不是教科书^正是这样，我不会把一些假的面向对象的分析问题布置给你然后去创建设计。我想给你一些东两去思考，思考能用到你自己项目里去的东西。如果你以前有面向对象的经验，你不需要放太多精力在这上面。如果你已经使用Python一段时间了，但是没有真正关心过所有与这些类相关的东西，那么这本书是有用的。

首先，想一想最近你已经完成的编程项目。确定设计中最突出的对象。尽可能多地为这个对象想一些属性。它有：颜色？重量？大小？利润？费用？名字？身份证号？价格？风格？考虑一下这些属性的类型。它们是原始语言还是类？其中一些属性其实是行为在伪装？有时候看似是数据，实际上是从对象的其他数据计算而来的，并且你可以用这个方法去做其他计算。这个对象的其他方法或者行为是做什么的？什么对象调用这些方法？对于这些对象它们有什么关系？

现在想想即将到来的项目。不管是什么项目；它可能是一个有趣的免费时间项目，或者是一个数百万美元的合同。它不必要是一个完整的应用；它可以只是一个子系统。执行一个基本的面向对象分析。确定需求和交互对象。勾勒出一个描述这个系统最高抽象层次的类图。确定最主要的交互对象。确定最小的支持对象。对于它们中最有趣的一些，想一下它们的属性和方法的细节。不同的对象采取不同的抽象层次。寻找你可以使用继承或者组合的地方。寻找你可以避免使用继承的地方。

目标不是设计一个系统（尽管很欢迎你这么做，如果你倾向于满足野心和可用时间的话）。目标是思考面向对象设计。重点关注你参与或者预计在将来会参与的项目，这样会让它更真实。

现在访问你最喜欢的搜索引擎并且查找一些UML的教程。有几十个，所以找到一个适合你的首选方法去研究。为先前确定的对象画一些类图或者序列图。不要花太多时间去记语法（毕竟，如果很重要，你总是可以再查它一次），只是了解下这个语言。有时候它会继续存在于你的大脑，如果你可以快速地勾勒出你下次OPP讨论的图表，会让你的交流更简单。

总结

在这一章，我们快速地浏览了面向对象的术语，专注于面向对象设计。我们学会了如何把两个不同的对象分成一个不同类的分类，并且通过类的接口描述了这些类的属性和行为。特别的是，我们讲了：

- 类和对象。
- 抽象、it装，以及信息隐藏。
- 设计一个公共接口。
- 对象关系：关联、组合和继承。
- 为了娱乐和交流的基本的UML语法。

下一章，我们将看一下在Python中如何实现类和方法。

2

第2章 Python对象

通过第1章的内容，现在我们手上已经有了一份设计并且准备把这份设计变成一个可以工作的程序！当然，通常并不是这么简单，但是本书将以Python编程的方式来实现它。通过本书的学习，我们将通过一些例子和提示看到什么才是一个优秀的软件设计，但是我们的关注点是面向对象编程。那么就让我们来看一下能够编写出面向对象软件的Python语法吧。

通过完成本章内容，我们将会学到：

- 在Python中如何创建类并实例化对象,,
- 如何给Python对象添加属性和行为。
- 如何把类组织成包和模块。
- 如何建议别人不要错误使用我们的数据。

创建Python类

我们不需要写过多代码就可以意识到Python是一门非常“简洁”的语言。当我们想做一件事情的时候,直接做就可以，不必要预先做很多的设置。一个普通的例子,就是用Python实现“hello world”，你会看到，只需要一行代码。

同样，在Python 3中一个最简单的类是这样实现的：

```
Class MyFirstClass:
    Pass
```

这是我们第一个面向对象的程序！类的定义以关键字class开头。之后跟着一个名字

（用户定义）用来标识这个类，并且以冒号结尾

ˆ＜类的命名必须符合标准的Python变量命名规则（必须以字母或者下画线开头，名
•字中只能包含字母、下画线或者数字）。同时，Python代码风格指南（在网页搜索“PEP 8”）建议类的名字应该使用驼峰式记法（以大写字母开头，并且随后紧跟的任意一个单词都要以大写字母开头）。

紧跟类定义的，是这个类的内容，并且做了缩进。和其他Python构建一样，缩进用来划清这个类的界限，这和很多其他语言使用花括号或者中括号不同。使用4个空格来代表缩进，除非你有强有力的理由不这么做（比如配合其他那些使用Tab键来缩进的代码）。任何像样的代码编辑器都可以配置成一旦按下rw键即可插入4个空格的功能。

因为我们第一个类并没有实际做什么事情，所以在第2行，我们简单地使用了 pass 这个关键词来表示不需要采取进一步的行动。

我们可能会想，对于这么一个最基本的类，我们可能不能对它做什么，但是它确实允许我们实例化这个类的对象。我们可以在Python 3的解释器里加载这个类，然后就可以交互式地操作它了。为了实现交互，我们把前面提到的这个类定义保存到一个叫first一class.py 的文件中，然后运行命令 python -i first_class.py〇 参数-i 告诉Python‘‘运行这段代码然后抛向交互式解释器”。下面这个解释器会话演示了与这个类的基本交互。

```
»> a = MyFirstClass ()
»> b = MyFirstClass ()
»> print (a)
<  main__.MyFirstClass object at 0xb7b7faec>
»> print (b)
<  main__.MyFirstClass object at 0xb7b7fbac>
»〉
```

这段代码从这个新类里实例化了两个对象，名字为a和b。通过键人类的名字并紧跟一对小括号这种简单方式，就可以创建一个类的实例。这看起来像一个普通的函数调用，但是Python知道我们退在“调用”一个类而不是函数，所以它理解它的任务是创建一个新对象。当打印的时候，这两个对象会告诉我们它们是哪个类，以及在内存中的存放位置。在Python代码中不常使用内存地址，但是在这里，它证实了包含两个明显不同的对象。

添加属性

现在我们有了一个甚本的类，但是它毫无用处。它不包含任何数据，并且也不做任何事情。我们如何做才能为一个给定对象赋予一个属性呢？

原来，在类的定义中，我们不必要做任何特殊的操作。我们可以通过点记法给一个实例化的对象赋予任意属性：

```
class Point:
    pass

pi = Point()
p2 = PointO

pl.x = 5
pi.y = 4

p2.x = 3
p2.y = 6

print(pi.xf pl.y)
print(p2.x, p2.y)
```

如果我们运行这段代码，最后两条打印语句会告诉我们这两个对象的新属性值。

```
5 4
3 6
```

这段代码创建了一个没有任何数据和行为的空的Point类。然后它创建了这个Point类的两个实例，并且给每个实例赋予一个x坐标和y坐标来标识、一个二维空间的点。我们需要做的只是通过<O/?/££7>.<|3/^/62//£>=<^0(/«^>这个语法来为属性赋值。这种方式有时被称为点记法。这个值可以是任意的：一个Python原始的内置数据类型、其他的对象，甚至可以是一个函数或者另外一个类！

让类实际做一些事情

现在，有一个带属性的对象确实是很棒的，但是面向对象编程真正的是对象之间的交互。我们感兴趣的是通过激发一些行为来引起这些属性的改变，是时候给我们的类加人一

些行为了。

让我们来模拟这个Point类的一些行为。我们可以从一个叫reset的方法开始，这个方法用来把点移到原点（原点就是x坐标和y坐标都是0的点）。这是一个非常好的可以用来介绍的行为，因为它不需要任何参数：

```
class Point:
    def reset(self):
        self.x = 0
        self.y = 0

p = Point()
p.reset()
print(p.x, p.y)
```

打印语句显示了两个属性值都变成了 o:

Python中的方法（method)和定义一个函数（fbnction)相同，以关键字def开头，紧跟一个空格和方法名，方法名后紧跟一对小括号，括号内包含参数列表（我们会在稍后讨论self这个参数），然后以冒号结尾。下面的行通过缩进方式包含了这个方法内部的语句，这些语句可以是任意的Python代码，它可以操作对象本身和任意传人的参数，当然只要这个方法认为这个参数合法。

方法和普通函数有一点不同，就是所有方法都有一个必需的参数，这个参数通常被称为self, 从来没有程序员采用其他名字来称呼这个变量（习惯的力量很强大)。即使如此，如果你想称它为this或者Martha, 也不会有人阻止你。

一个方法中的self参数，是对调用这个方法的对象的一个引用。我们可以和其他对象一样访问这个对象的属性和方法$_3$当要改变self对象的x和y属性值时，正是我们通过调用内部reset方法实现的$_3$

你会注意到，当我们调用p. reset ()方法时，并没有给它传人self参数 Python自动帮我们做了，它知道我们正在调用P对象的一个方法，所以它自动地把这个对象传给了这个方法。

然而，方法真正只是一个函数而已，只不过它恰巧出现在类中。除了可以直接调用一个具体对象的方法以外，我们也可以在类中调用这个函数，并且明确地把这个对象作为self参数传给对象：

```
p = Point ()
Point.reset(p)
print(p.x, p.y)
```

输出和前面的例子一样，因为在内部做了同样的处理。

如果我们在类定义中忘记包含self参数会怎么样呢？ Python会返回错误消息：

```
»> class Point:
...     def reset():
...         pass

»> p = Point ()
»> p.reset()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: reset() takes no arguments (1 given)
```

这个错误消息并不是那么清晰（“你这个傻瓜！你忘记self参数了”会得到更多的教训）。只要记住当你看到一条错误信息提示你缺少参数时，第一件事情就是去检查在方法定义时你是否忘记了带self参数。

那么我们如何给方法传递多个参数呢？让我们添加一个新的方法，这个方法允许我们把点移动到任意位置，而不只是移动到原点。我们也可以添加另外一个方法，它接收另一个Point对象作为输入，然后返回这两个对象之间的距离：

```
import math

class Point:
    def move(self, x, y):
        self.x = x
        self.y = y

    def reset(self):
        self.move(0, 0)

    def calculate_distance(self, other_point):
        return math.sqrt(
                (self.x - other_point.x)**2 +
                (self.y - other_point.y)**2)
```

```
#如何使IH它：
pointl = Point()
point2 = Point ()

pointl.reset ()
point2.move(5,0)
print(point2.calculate_distance(pointl> >
assert (point2.calculate_distance(pointl)
        pointl.calculate_distance(point2))
pointl.move(3,4)
print(pointl.calculate_distance(point2))
print(pointl•calculate_distance(pointl>)
```

最后的打印语句给了我们下面的输出：

```
4.472135955
0.0
```

上面这个例子做了很多事情。这个类现在拥有3个方法。move方法接收x和y两个参数，并且给self对象赋值，这和前面例子中的reset方法很像。之前的reset方法现在叫作move, 因为reset只不过是一个移动到特定位置的move。

calculat^distance方法使用了不是很复杂的勾股定理来计算两个点的距离。我希望你能懂点数学（**是平方的意思，math，sqrt是计算平方根），但是现在我们的关注点是学习如何写一个方法，所以数学我们不做要求。

最后几行代码展示了如何调用带参数的方法，简单地把参数包含在小括号里，用同样的点记法调用这个方法。我这里只是选了一些随机的位置来测试这些方法。这些测试代码调用了每一个方法并把结果打印在了控制台上。asser函数是一个简单的测试工具，如果assert后面的语句是False（0、空或者None）的话，这个程序就会异常退出。这里，我们用它来确保不管哪个point类调用另一个point类的calculate_distance方法，得到的距离是一样的。

对象的初始化

如果不清晰明确地给Point对象赋予x坐标和y坐标，也不使用move方法或者直接访问它们，那么你将得到一个没有实际位置的无意义的点。在这种情况下，我们试着访问

这个对象会发生什么呢？

好的，我们不妨尝试做一下，看会如何。对于学习Python来讲，“尝试做一下，看会如何”是一个非常有用的学习工具。打开你的交互式解释器，键人要试的代码。下面这个交互式会话展示了如果你尝试访问一个不存在的属性会发生什么。如果你把之前的例子存成一个文件或者使用随书携带的代码文件例子，那么你可以通过命令行python _i filename .py把代码加载到Python解释器。

```
»> point = Point ()
»> point.x = 5
»> print (point .x)

»> print (point .y)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Point' object has no attribute 'yf
```

好处是，运行这段代码会至少抛出一个有用的异常。我们会在第4章详细讲解异常。在这之前你也可能会看到过它们（特别是普遍存在的语法错误异常SyntaxEiror, 出现这个异常意味你键人了一些错误的代码！）。但此时，你只需要简单地意识到发生了错误即可。

这些错误输出对于调试是有帮助的。在t面的交互式解释器中，错误输出告诉我们错误发生在第1行，这不完全正确（因为在一个交互式的会话中，一次只执行一行代码，所以它总会提示错误发生在第1行）。如果我们运行一个写在文件中的脚本，错误输出会告诉我们精确的行数，这样找到出错代码会变得容易。此外，它告诉我们这个错误是一个AttributeErroi•，并且通过一个有用的消息告诉我们这个错误是什么意思。

我们可以捕捉错误信息并修复错误，但是现在这个情况，好像我们应该指定一些默认值。可能是每一个新的对象都应该使用默认值调用reset（>方法，或者当用户创建新对象时候，如果能强制用户告诉我们这些Point对象的位置，这样会比较好3

大部分面向对象编程语言都有一个叫构造函数的特殊方法，当它被创建的时候会创建和初始化对象。这点, Python和它们有一点点不同, Python有一个构造函数和一个初始化函数。正常情况下，构造函数很少能用得到，除非你想做一些特别另类的操作。所以我们下面开始讨论初始化方法。

除了有一个特殊的名字_init_以外，Python的初始化方法和其他方法没什么不同。开始和结尾的双下画线的意思是：“这是一个特殊的方法, Python解释器会特殊对待它”。对于你自己定义的函数，名字一定不要使用双下画线开头结尾。Python解析器不会识别它，

但是有一种可能，就是Python的设计者们出于特殊目的，在将来添加了一个如此命名的函数，如果他们这么做了，你的代码就会异常退出。

让我们在Point类里开始添加一个初始化函数，这个函数要求用户在实例化Point对象的时候提供x和y坐标值：

```
class Point:
    def __init__(self, x, y):
        self.move(x, y)

def move(self, x, y):
    self.x = x
    self.y = y

def reset(self):
    self.move(0, 0)

#构造一个Point
point = Point(3, 5)
print(point.x, point.y)
```

现在，我们的点再也不会没有y坐标了！如果我们构造一个point对象时，没有包含合适的初始化参数，程序会报一个“没有足够参数”的错误，与之前当我们忘记self参数时收到的那个错误类似.，

如果不想让这两个参数是必需的，我们该怎么办？那么我们可以通过不改变Python函数的语法，而通过提供参数默认值来实现，语法就是在每一个变fi名后通过等号赋予参数默认值。如果调用对象时没有提供那个参数，那么就会使用默认参数值；此时对于这个函数来讲，这个变量仍然是可用的，但是这些变M将会使用参数列表里的默认值。这里有一个例子：

```
class Point:
    def —init—(self, x-0, y=0):
        self.move(x, y)
```

大多数情况下，我们会把初始化的语句放到_^：11：_函数里。但是，就像之前提到过的一样，除了初始化函数，Python还有一个构造函数，你可能永远不会用到这个函数，但是下面的讲解会帮助我们认识到它的存在，所以这里我们稍微提一下。

和_1ˆ1：_不同，构造函数名叫_1ˆ>ˆ_，并且它只接收一个参数，就是这个将要被构造的类本身（它会在对象被构造之前调用，所以这里也就没有self参数），同时它会返回刚被创建的对象。提到复杂的元编程技术时，这个可能会比较有趣，但是在日常编程工作中，它不是很有用。在练习时，你几乎不需要去使用_1ˆ以_，因为_丨心_的功能已经足够了。

解释你自己

Python是一门非常简单易读的编程语言，有些人可能认为它是文档型语言（self-documenting）。然而，与我们进行面向对象编程时，清晰地总结每一个对象是什么，每一个方法是做什么的，并把这些内容写成AP〖文档是很重要的。做到文档的实时更新很闲难，最好的方式就楚把这些文档写到代码里[3]

Python的docstring提供了对这种文档方式的支持。在每一个类、函数、方法的开头，紧接着它们的定义（以冒号结尾那行）〞丨以有一行Python的标准字符串。这行字符串也需要和下面的代码一样有缩进。

docstring是用单引号（‘）或#双引号（“）标注的Python字符串。通常，docstring比较长并且跨越多行时 PEP 8风格指南建议行的长度不应该超过80个字符），就可以格式化成多行string, 用（ ‘”）或者标注起来。

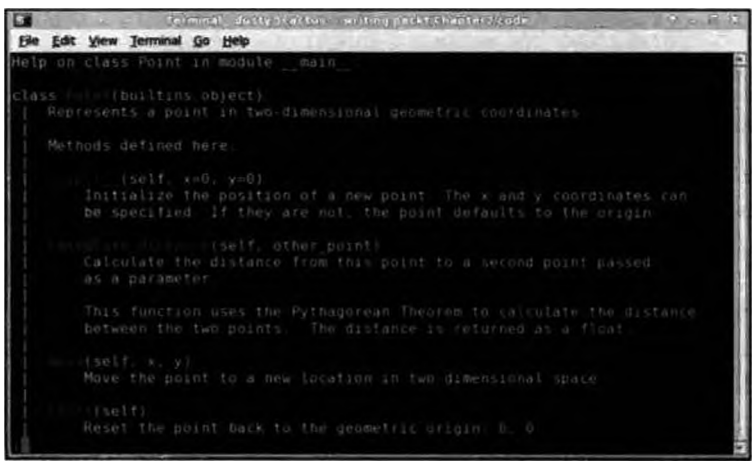
docstring应该能清晰准确地总结出它所描述的类或者对象的用途，应该能解释任何用法不是那么明显的参数，并且也包含如何使用这些AP丨的简单例子。任何一个AP丨使用者应该知道的注意事项或者问题，都应该标注在这里。

在这一节的结尾，我们将通过一个完整的具有文档的Point‘类来展示docstring的用法：

```
import math

class Point:
     'Represents a point in two-dimensional geometric coordinates

    def _init—(self, x=0, y=0):
    '''Initialize the position of a new point. The x and y
       coordinates can be specified. If they are not, the point
       defaults to the origin.1''1
    self.move(x, y)
```

```
def move(self, x, y):
    "Move the point to a new location in two-dimensional space.
    self.x = x
    self.y = y

def reset(self):
    'Reset the point back to the geometric origin: 0, 0*
    self.move(0, 0)

def calculate_distance(self, other_point):
    '''"""Calculate the distance from this point to a second point
       passed as a parameter.

    This function uses the Pythagorean Theorem to calculate
    the distance between the two points. The distance is returned
    as a float.'''

   return math.sqrt(
           (self.x - other_point.x) **2 +
           (self.y - other_point.y)**2)
```

试着键入或者加载这个文件到交互式解释器（记住，用python -i filename.py）。然后在Python提示符后输入help（Point）<enter>，你应该会看到这个类的漂亮的格式化文档，如下面的截图所示：



模块和包

现在我们知道了如何创建类和实例化对象，是时候想一下该如何组织它们了。对于小程序来讲，我们可以把所有类都放到一个文件里，并且在文件最后通过一些代码来使它们相互调用。然而，随着我们项目的发展，在我们定义的众多类中找到想要编辑的那个类变得很闲难。模块Module就这么产生了。模块是非常简单的Python文件，仅此而已。在小程序里，单个Python文件就是一个模块，两个文件就是两个模块。如果你在同一个文件夹里卉两个文件，我们町以通过从这个模块加载一个类的方式来使用其他模块3

比如，如果我们正在构建一个电子商务系统（e-commerce system)，我们可能会在数据库里存储大量数据我们可以把所有关于数据库访问的类和函数放到一个单一文件里（给它起一个有意义的名字：database.py)。然后其他的模块（如客户模型、产品信息和财产清单）为了访问数据库，可以从这个模块里导人那些关于数据库访问的类。

import语句是用来导人模块或者从模块里导入特定的类或函数的。在上一节的Point类中，我们已经通过一个例子看到了 import的用法。我们通过import语句导入了一个Python的内置math模块，从而可以使用sqrt函数来计算距离。

这里有一个具体的例子。假如我们有一个叫database.py的模块，它包含一个叫Database的类，第2个模块叫products.py,它负责产品相关的查询。此时，关于这两个文件的具体内容，我们不需要想太多。只需要知道 products.py需要从database .py里实例化一个Database类，然后就可以在数据库产品表里执行查询了。

这M有一些导人语句语法的变量可以用来访问这个类。

```
import database
db = database.Database ()
#数据库搜索操作
```

上面这个版本，把database模块导人到了 products命名空间（在一个模块或者函数内部可以访问到的名称列表），这时候任何在database这个模块里的类或者函数. 都可以通过database. 〈something; ˆ种记法来访问。或者，我们可以用from... import语法来导人一个类：

```
from database import Database
db = Database()
#数据库搜索操作
```

假如出于某种原因，products自己已经有了一个叫Database的类，并且我们不想

让这两个名字冲突，那么在products模块里使用这个类的时候，我们可以重新命名它：

```
from database import Database as DB
db = DB()
#数据库搜索操作
```

我们也可以在同一行语句里导人多项。如果database模块同时包含一个叫Query的类，可以通过下面的语句导入这两个类：

```
from database import Database, Query
```

有一些教程说，我们甚至可以用下面的语法从database模块里导入它所有的类和函数：

```
from data base import *
```

千万不要这么做！任何一个有经验的Python程序员都会告诉你绝对不要用这种语法，他们会用“这会使命名空间混乱”这种模糊不清的理由，对于初学者来说，这个理由并没有多大意义。有一种方式可以学习到为啥要避免这种语法，那就是两年以后，你再尝试使用这种语法并理解你的代码，但是在这里，我们会快速地解释一下原因，以便节省时间和两年糟糕的代码！

当我们在文件开头用from database import Database明确地导人Database类的时候，可以清楚地看到database类是从哪里来的。我们可以在文件400行以后使用db = Database（），并且快速地看一下导入语句就可以知道Database类是从哪里来的。然后如果需要阐明如何使用Database类的时候，你可以去查看源文件（或者在交互式解释器里导人这个模块，并且使用help（database. Database)命令)。然而，如果我们使用from database import *这个语法，寻找这个类的位置将会花费长一点的时间。代码维护会变成疆梦。

同时，很多编辑器能提供一些额外的功能，像可靠的代码补全，或者如果使用正常的导入语句，跳到类定义地方的功能。import *语法通常会完全破坏它们可靠地做这些事情的能力。

最后一点，使用import *语法会把一些意想不到的对象导人到我们本地的命名空间。肯定的一点是，它会把这个模块里定义的所有类和函数导人进来，但是，这样也会把这个模块自己导人的任意类和模块导人到当前文件中。

尽管有这些警告，你可能会想，“如果我只是为一个模块使用了 from X import *

语法，我可以假设任何未知的导人都来源于这个模块”。从技术层面来讲这是对的，但在实际中它会发生问题我敢保证，如果你使用这种语法，你（或者其他那些试图理解你的代码的人）将会遇到极其让人沮丧的时刻，“这个类到底是从哪里来的？ ”一个模块中所有用到的名字，都应该来自一个特定的空间，不论它是在这个模块中定义的，还是从其他模块明确导人的。不应该有奇怪的变量凭空出来。我们应该总是能够立即就确定，在当前的命名空间里，这个名字来自哪里。

组织模块

随着项目发展成越来越多模块的集合，我们可能会发现，需要增加另外一层抽象，基于模块水平的某种层次模型。但是我们不能把模块放到模块里面，一个文件只能持有一个文件，毕竟，模块也只不过是Python文件而已。

然而，文件可以放到文件夹里，模块也是可以的。一个包（package)就是放到一个文件夹里的模块集合。包的名字就是文件夹的名字。我们需要做的只是告诉Python这个文件夹是一个包，并且把一个名为_init_.py的文件（通常是空的）放到这个文件夹里。
如果我们忘记创建这个文件，就没法从这个文件夹里导人那些模块。

在我们的T.作目录里，把我们的模块放到了一个叫ecommerce（电子商务）的包里，这个目录同样包含一个main .py文件用来启动程序。此外，在ecommerce包里再增加一个叫payments的包用来管理不同的付款方式。文件夹的层次结构看起来像这样：

```
parent_directory/
    main.py
    ecommerce/
            _init__ .py
            database.py
            products.py
            payments/
                    —init__ .py
                    paypal.py
                    authorizenet.py
```

当在包之间导人模块或类的时候，我们要注意语法。在Python 3中，导人模块有两种方式：绝对导人和相对导入。

绝对导入

绝对导入需要指明这个模块、函数的完整路径，或我们希望导入的路径。如果我们需要访问products模块里的Product类，我们可以使用下面这些语法做绝对导人：

```
import ecommerce.products
product = ecommerce.products.Product()
```

或者

```
from ecommerce.products import Product
product = Product()
```

或者

```
from ecommerce import products
product = products.Product()
```

import语句使用点号作为分隔符来分隔包或者模块。

对于任何模块，这些语句都可以运行。我们可以在main. py里、在database模块里，或者任意一个payment模块里使用这样的语法实例化一个Product对象。确实，只要这些包在Python里是可用的，就可以导人它们。比如，这些包还可以安装到Python的site packages文件夹里，或者可以通过自定义PYTHONPATH环境变fi来动态地告诉Python，该如何搜索它即将要导人的包或者模块。

有这么多方式，我们该选择哪种语法呢？它取决于个人口味及你手头的应用程序。如果我想用的这个products模块里有几十个类和函数，我通常会使用from ecommerce import products语法来导人模块名字，然后通过products . Product来访问单一的类。如果我只需要products模块里的一两个类，我会用from ecommerce.products import Product语法来直接导人它们。我个人不太常使用第一种语法，除非有某种命名冲突（比如，我需要访问两个名字都叫products但完全不同的模块，需要把它们分开>。你可以做任何能让你的代码看起来更优雅的事情。

相对导入

当处理一个包里的相关模块时，详细指明完整路径看起来有点蠢，因为我们知道父模块的名称。相对导入就这么产生了。相对导人基本上就是这么一个意思，“找出一个类、函数或者模块，它的定位要相对于当前模块”。比如，如果当前我们在products模块下T.作，想从“隔壁”的database模块里导人Database类，我们可以使用相对导人：

```
from .database import Database
```

database前面这个点号说明，“使用当前包里的database模块”。在这种情况下，当前的包指的是包含目前我们正在编辑的product.py文件的这个包，这个包就是ecommerce 包O

如果我们正在编辑ecommerce.payments包里的paypal模块，我们可能会说，“使用父包里的database包”，改成两个点号就可以轻松做到这点：

```
from ..database import Database
```

我们可以通过使用更多的点号来访问层级的更上层。当然，一方面可以往下层访问，另一方面可以往上层访问。我们没有一个层级足够深的例子来恰当地展示，但是如果我们有一个ecommerce .contact包，这个包里有一个email模块，我们要把这个模块中的send_mail函数导入到我们的paypal模块中，下面的导入语句是有效的：

```
from ..contact.email import send_email
```

这个导人语句使用了两个点来告诉我们.“payments包的父包”，然后使用普通的package .module语法，访问上层的contact包。

在任何一个模块里，我们可以指定要访问的变量类或者函数。这是一个很方便的方法，可以用来存储没有命名空间冲突的全局状态。比如，我们已经把Database类导人到了不同的模块里，并且实例化了它，但是，在database模块里，有且只有一个全局的database对象会更有意义一些。此时的database模块看起来应该是这样的：

```
class Database:
    #数据库实现
    pass

database = Database()
```

这样，我们就可以使用任意一种讨论过的导入方法来访问database对象，例如：

```
from ecommerce.database import database
```

上面这个类有个问题，就是在第一次导人这个模块的时候，就立即创建了 database对象，通常对象的创建应该在程序启动的时候。事情总不那么理想，因为数据库连接需要一段时间，这会减缓程序启动，或者也许得不到数据库连接信息。我们可以减缓database对

象的创建，直到真正需要它的时候，通过调用initialize_database函数来创建模块级别的变量：

```
class Database:
    #数据库实现
    pass

database = None

def initialize_database():
    global database
    database = Database()
```

global关键字告诉Python，我们刚刚在initialize_database里定义了一个模块级别的database变ffl ，如果我们没有指明这个变量是全局的 global ），当initialize_database方法执行完返回的时候，Python在这个方法内部新创建的这个database变量会被抛弃，剩下那个模块级别的database变_虽：，值不会变（None）。

就像在这两个例子中展示的一样，当导入模块的时候，模块里的所存代码都会被立即执行。但是如果模块里的是一个方法或者函数，会创建这个函数，但函数里的代码直到函数被调用的时候才会执行。这对执行脚本来说比较狡猾（就像在e-commerce例子里的main脚本）。通常，我们会写一个程序让它来做一些有用的事情，过后发现，我们想从另一个程序里的一个模块导人一个函数或者类。但是只要我们导人了它，这个模块里的所有代码都会被立即执行。当我们只是想访问那个模块里的一些函数的时候，如果不小心，可能会把当前正在运行的程序终止掉。

为了解决这个问题，我们应该总是把启动代码放到一个函数里(通常叫作main函数），并且只有肖我们知道这是在执行脚本的时候，才去执行这个函数，而不是在其他脚本导入我们的代码的时候。但是我们如何知道呢？

```
class UsefulClass:
    1'fThis class might be useful to other modules.* 1'
    pass

def main ():
    '''creates a useful class and does something with it for our module.'''
    useful = UsefulClass ()
    print(useful)
```

```
if —name_ == "—main_":
    main()
```

每一个模块都有一个特殊的变量_name_(记住，Python使用双下画线命名一些特殊变量，像一个类里方法），当导人这个模块的时候，这个变量指明了模块的名字。但是当这个模块直接通过python module.py执行的时候，就不会导人这个变而这时_nam_变量就赋值给一个字符串"_mai_"，而不再是模块名了。把你所有的代码都包在if _name_ = 里面便成了一个策略，你就会发现它是非常有用的，可以防止万一有一天你写了一个函数，代码里导人了其他的代码。

方法fli现在类里，类出现在模块里，模块出现在包里，所有的都会这样吗？

事实不是这样的，当然这确实是Python程序里的典塑顺序，但并不是唯一可能的布局方式。类可以定义在任何地方。它们通常是在模块级别定义的，但是它们也可以在一个函数或者方法内部定义，像：

```
def format—string(string, formatter=None):
    '''Format a string using the formatter object, which
    is expected to have a format() method that accepts
    a string.[1]'''
    class DefaultFormatter:
        *'[1] Format a string in title case.'*'
        def format(self, string):
            return str (string) •title○

    if not formatter:
        formatter = DefaultFormatter()

    return formatter.format(string)

hello_string = "hello world, how are you today?"
print (" input: ” + hello_string)
print ("output: •’ + format—string (hello_string))
```

输出：

```
input: hello world, how are you today?
output: Hello World, How Are You Today?
```

format_string函数接收一个字符串和一个可选的格式化对象作为参数，然后执行格式化字符串操作。如果没有提供格式化方法，函数会自己创建一个格式化方法，作为一个本地的类并实例化它。既然是在函数内部创建的，这个类就不能访问这个函数外面的任何地方。类似地，函数也可以定义在另一个函数里面;总之，任何时候都可以执行任何Python语句。这种“内部”类或者函数是有用的，特别对于那些在模块级别不需要或值得保留向己作用范围的“一次性”项目，或者只有在一个荤一方法里有意义的项目。

谁可以访问我的数据

大多数面向对象编程语言会有一个“访问控制”的概念。这个和抽象有关。对象里的某些属性和方法会被标记为“私有的（private〉”，意思是只有这个对象可以访问它们。另外一些会被标记为“受保护的（protected)”，意思是只有这个类和它的子类可以访问。剩下的会被标记为“公共的（public)”，意思是允许任何其他对象访问它们。

Python不会这样做、Python不相信强制制定规则这种方式会让人严格遵守。相反，它提供了一个不强制的指南和最佳实践。在技术层面上，一个类里的所有方法和属性都是公共可访问的。如果我们想建议某个方法不应该能被公共访问，我们可以通过在docstring里放一个提示来表明是否这个方法只是内部使用的(解释面向公共的API如何T作会更好！）。

按照惯例，我们也可以给一个属性或者方法加一个下画线的前缀，大部分Python程序员会把这个解释为，“这是一个内部变M，在直接访问它之前请三思”。但是如果别人认为访问这个变fi能给他们的程序带来最大的帮助，那么什么也阻止不了他们去访问。是的，如果他们这么想，为什么我们要阻止呢？我们不会想到将来这个类会如何使用，

你可以用另外一种方式，强烈建议外部对象不能访问某个属性或者方法，就是给它添加一个双下画线的前缀。这就是所谓的对问题中的属性做“名称改编 name mangling)”。基本的意思就是，外部对象如果真的想访问的话，还是仍然可以调用这个方法的，但是你需要做额外的工作，并且它是一个很强的指示器，指示你想到你的属性应该保持私有性。例如：

```
class Secretstring:
    *'*A not-at-all secure way to store a secret string.[11]'

   def __init__(self, plain—string, pass_phrase):
      self.__plain_string = plain—string
      self._pass_phrase = pass_phrase
```

```
    def decrypt(self, pass_phrase):
        *'''Only show the string if the pass_phrase is correct.'[1]'
        if pass_phrase == self._pass_phrase:
            return self.__plain_string
        else:
            return "
```

如果我们在交互式解释器里加载并测试这个类的话，我们可以看到那段明文字符串对外隐藏了：

```
»> secret string = Secretstring(nACME: Top Secret", "antwerp")
»> print (secret string • decrypt ("antwerp•' }
ACME: Top Secret
»> print (secret string._____plain_text)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError:    * Secretstring[1] object has no attribute
'_plaintext'
»〉
```

看起来它生效了。没有密码的话没人可以访问我们的plain_text属性，所以它一定是安全的[3]不过，在我们太过兴奋之前，让我们看看想破解我们的密码是多么容易：

```
>» print(secret string• Secretstring_plain  string)
ACME: Top Secret
```

不要啊！有人已经破解r我们的密码字符串。好处是我们查到了，这就是Python的“名称改编 name mangling)”起作用了。当我们使用双下画线开头定义一个属性时，这个属性会自动加上一t_<classname>的前缀。这时候这个类的方法在内部访问变设，它们没有被A动转换、当外部的类想要访问它时，它们必须要自己做名称改编[2]。所以名称改编不能保证隐私，它只会强烈建议要隐私。大部分Python程序员不会在其他对象中碰触双下画线开头的变fi,除非他们有极其强制性性的理由这么做。

然而，如果没有强制性的原因，大部分Python程序员也不会使用单下画线开头的变量。大部分情况下，在Python代码中没有什么好的理由说一定要用“名称改编”的变M,这样

就是还没有加_<classname>W的缀，可以£常访问■——译#注
就是要程序员it己加那个前缀一-译者注

做会导致不幸。比如，一个“名称改编”的变量可能对于子类有用，并且它需要自己做改编[3]。其他的对象如果想访问你的隐藏信息，只要让它们知道，你认为使用单下画线前缀或者一些清晰的docstring，都不是好主意。

最后，我们可以直接从包里导入变量，这和从包里导入模块截然不同。在我们之前的例子中，我们有一个ecommerce包，它包含两个模块，一个叫database.py，一个叫products.py。database模块里有一个db变量，这个变量在很多地方都会被访问。如果我们用 import ecommerce • db 取代 import ecommerce • database . db，这样会不会方便一些？

还记得_init_.py文件定义目录为包吗？只要我们愿意，这个文件里可以包含任意变量或者类的声明，而且它们会作为这个包的一部分被我们使用。在我们的例子中，如果ecommerce/_init_.py文件里包含这么一行：

```
from .database import db
```

这样我们就可以用下面的导入语句，在main . py或者其他文件中访问db这个属性了：

```
from ecommerce import db
```

如果你能记起导致ecommerce.py这个文件是一个模块而不是包的原因就是_init_.py文件，会很有帮助。如果你把所有代码都放到了一个单独的模块里，过后又决定把它拆分成一个包里的多个包，_init_.py文件同样会对你有帮助。其他模块想要访问这个新包，_init_.py文件仍然是主要的切入点，但是在内部，代码仍然可以被组织成许多不同的模块或者子包。

案例学习

把前面讲的所有都结合在一起，让我们构建一个简单的命令行笔记本应用（command-line notebook application）。这是一个非常简单的任务，所以我们不会用到多个包。然而在这个例子中，我们会看到类、函数、方法和docstring的基本使用方法。

让我们从一个简单的分析开始：备注（notes）是存在笔记本（Notebook）里的短的备忘录（memos）。每一个备注（note）都应该记录下它被创建的时间，并且为了查询方便，可以添加标签（tag）。备注（note）应该可以修改。我们也需要能够搜索备注。所有的这些

[3] 就是加前缀„ —译者注

功能都应该通过命令行实现。

很明显的对象就是Note。不太明显的就是要有一个容器对象Notebook。标签和日期看起来也是对象，但是我们可以使用Python标准库里的日期对象，以及逗号分隔的字符串作为标签。这时候为了避免复杂，对于这些对象我们不会去定义单独的类。

Note对象有memo本身、tags和creation_date这几个属性。每一个备注也需要一个唯一的整数id，这样用户就可以通过菜单接口选择它们。备注有一个方法可以用来修改它的内容，另外一个方法来修改标签，或者我们可以直接让笔记本访问这些属性。为了让搜索更简单，Note对象需要一个match方法。这个方法以一个字符串作为输入参数，不需要直接访问Note的M性，就能告诉我们是否有一条备注与输入的字符串匹配。这样的话，如果我们想修改搜索参数（比如，搜索标签而不是搜索备注的内容，或者让搜索结果大小写敏感），只需要在一•个地方修改即可。

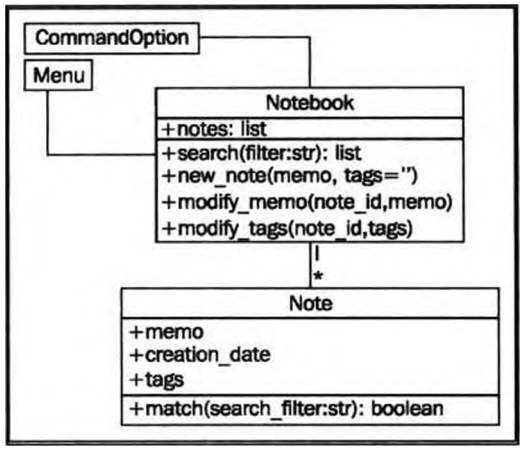
很明显，Notebook对象会有一个notes列表作为它的属性，也需要一个能返回过滤后的notes列表的search方法,:，

但是，我们如何和这些对象交互呢？我们已经指明了一个命令行的应用，当要运行这个程序的时候，这个应用可以让我们用不同的选项添加修改命令，或者我们会有各种菜单，这些菜单允许我们对这个笔记本对象做各种不同的操作。如果我们这样设计，那么上面提到的这些接口都是允许用的，或者在将来，我们可以添加基于Web页面的接口或者图形开发工具（GU｜toolkit)的接口。

作为一个设计决定，我们现在就要实现菜单接口，为了做到我们设计的Notebook类是可扩展的，基于命令行选项的接口设计我们也要记在脑子里。

如果我们有两个命令行接口，每一个都和Notebook做交互，那么Notebook为了能和它们交互，就需要一些方法。我们需要可以add新备注的方法，基于id modify已存在的备注的方法，同时还有我们已经讨论过的search方法。这个接口同样需要可以列出所有的备注，但是它们也可以通过直接访问notes列表属性获取。

我们可能会丢失一些细节，但是给了我们需要写的代码的一个很好的概述。我们可以在一个简单的类图里总结一下：



在你写任何代码之前，让我们先为这个项目定义一下文件夹的结构。菜单接口应该清晰地存在于它自己的模块里，因为它将是一个可执行的脚本，并且将来我们可能有其他可执行脚本来访问笔记本。Notebook和Note对象可以在一个模块里。这些模块可以放到同一个顶级目录里而无须把它们放到一个包里5〉一个空的command_option .py模块在将来可以帮助提醒我们计划添加的新的用户接口。

```
parent_directory/
    notebook.py
    menu.py
    command_option.py
```

现在，回到代码。让我们开始定义Note类，因为它看起来比较简单。下Ifi丨的例子展示丫整个Note类。例子里的docstrings解释了它们之间是如何适用的。

```
import datetime

#为所有新的备注存储下一个可用的id
last_id = 0

class Note:
    '*'Represent a note in the notebook. Match against a
    string in searches and store tags for each note.''[1]

    def —init___(self, memo, tags='*):
```

```
        'f * initialize a note with memo and optional
        space-separated tags. Automatically set the note's
        creation date and a unique id.* 1'
        self.memo = memo
        self.tags = tags
        self. creation__date = datetime . date. today ()
        global last_id
        last_id += 1
        self.id = last_id

    def match(self/ filter):
        1 *'Determine if this note matches the filter
        text. Return True if it matches, False otherwise.

        Search is case sensitive and matches both text and
        tags•|*
        return filter in self.memo or filter in self.tags
```

在继续之前，我们应该快速地启动交互式解释器并且测试我们目前的代码。经常频繁测试，因为事情总不会以我们期待的方式去工作。事实上，当我测试这个例子的第一个版本时，我发现在match函数里我忘记了 self参数！我们会在第丨0章讨论自动化测试；现在，用解释器来检查一些东西就足够了：

```
»> from notebook import Note
»> nl = Note ("hello first")
»> n2 = Note ("hello again")
»> nl. id
1
>» n2.id
2
»> nl .match ('hello ')
True
»> n2 . match (' second')
False
>»
```

看起来一切都正常。接下来让我们创建我们的笔记本：

```
class Notebook:
    * * 'Represent a collection of notes that can be tagged,
    modified, and searched.'''

    def _init—(self):
        '''Initialize a notebook with an empty list.''*
        self.notes =[]

    def new_note(self, memo, tags='');
        ''*Create a new note and add it to the list.*''
        self.notes.append(Note(memo, tags))

    def modify- memo(self, note_id, memo):
        1,1 Find the note with the given id and change its
        memo to the given value.1'*
        for note in self.notes:
            if note.id == note_id:
                note.memo = memo
                break

    def modify_tags(self, note_id, tags):
        '''Find the note with the given id and change its
        tags to the given value.1 *'
        for note in self.notes:
             if note.id == note_id:
                 note.tags = tags
                 break

    def search(self, filter):
        '''Find all notes that match the given filter
        string.…
        return [note for note in self.notes if
                note.match(filter)]
```

我们快速清理一下，酋先让我们测试并确保它能工作：

```
»> from notebook import Note, Notebook
»> n = Notebook ()
```

```
>» n. new note ("hello world")
>» n.new note ("hello again")
»> n.notes
[<notebook.Note object at 0xb730a78c>, (notebook.Note object at
0xb73103ac>]
»> n.notes [0] . id
1
»> n. notes [1] . id
2
>» n. notes [0] .memo
'hello world'
»> n. search ("hello")
[(notebook.Note object at 0xb730a78c>, (notebook.Note object at
0xb73103ac>]
»> n. search ("world")
[(notebook.Note object at 0xb730a78c>]
»> n• modify—memo (1, "hi world")
»> n• notes [0] .memo
'hi world'
>»
```

它确实可以丁.作.iii然代码有点混乱；我们的modify_tags和modifyjmemo方法几乎是相同的，这不是良好的编程实践。让我们看看是否可以修复这个问题。

在对一个备注做一鸣事情之前，两个方法都试图通过给定的丨D来区分不同的备注。因此让我们添加一个通过给定丨D来定位备注的方法。我们让这个方法的名字以下_线开头，这样这个方法就仅供内部使用，当然.如果我们想.我们的菜单接口也逛可以访问这个方法的。

```
    def _find—note(self, note_id):
        *_'Locate the note with the given id" * *
        for note in self.notes:
            if note.id == note_id:
                return note
        return None

    def modify_memo(self, note—id, memo):
        ''* Find the note with the given id and change its
```

```
        memo to the given value.*''
        self._find_note(note_id).memo = memo
```

现在应该可以工作了；让我们看看菜单接口。接口只需要简单地提供一个菜单并且允许用户输人他们的选择。下面是第一次尝试：

```
import sys
from notebook import Notebook, Note

class Menu:
    * *'Display a menu and respond to choices when run.
    def __init__(self):
        self.notebook = Notebook()
        self.choices ={
                  T:  self.show_notes,
                u2u :  self.search_notes,
              •'3,,:  self.add_note,
                "4":  self .modify__note,
                "5":  self.quit


    def display_menu(self):
        print (•"•••
Notebook Menu

1. Show all Notes
2. Search Notes
3. Add Note
4. Modify Note
5. Quit
""")

    def run(self):
        •"Display the menu and respond to choices•***
        while True:
            self.display_menu()
            choice = input("Enter an option:")
```

```
            action = self.choices.get(choice)
            if action:
              action()
            else:
               print ("{0} is not a valid choice*'. format (choice))

    def show_notes(self, notes=None):
        if not notes:
            notes = self.notebook.notes
        for note in notes:
            print("{0}: {1}\n{2format(
                note.id_f note.tags, note.memo))

    def search_notes(self):
        filter = input ("Search for: ·)
        notes = self.notebook.search(filter)
        self.show notes(notes)

    def add-note(self>:
        memo = input("Enter a memo:")
        self.notebook.new_note(memo)
        print("Your note has been added.")

    def modify_note(self):
        id = input("Enter a note id:")
        memo = input("Enter a memo: H)
        tags = input ("Enter tags: ••)
        i f memo:
            self.notebook.modify_memo(id, memo)
        if tags:
            self . notebook.modif y__tags (id_f tags)

    def quit(self):
        print("Thank you for using your notebook today.")
        sys.exit(0)

if
```

```
Menu().run()
```

这段代码首先通过一个绝对导人语句导人了笔记本对象。相对导入无法工作，因为我们还没有把我们的代码放到一个包里面。Menu类的run方法会重复地显示一个菜单并且通过笔记本里的函数对输人做出响应。这是通过使用一个Python中特有的习惯用法来实现的。用户输人的选择是字符串。在菜单的方法里我们创建一个字典来把字符串映射到菜单对象本身的函数然后，当用户做出一个选择，我们从这个字典里检索这个对象。变M action实际上指向一个特定的方法，并且通过添加空括号（因为这些方法都不需要参数）给变量的方式调用这个方法。当然，用户可能做了一个不恰当的选择，所以在调用方法之前，我们会检查这个行为是否真的存在。

各种方法都要求用户输人与调用和Notebook对象相关联的方法。对于search的实现，我们注意到，在我们过滤完备注之后，我们需要显示它们。所以我们让show_notes函数执行双重任务；它接收一个可选的notes参数。如果提供了这个参数，它就只显示过滤后的备注，但是如果没有提供，它会显示所有备注。因为notes参数是可选的，所以仍然可以像一个空菜单项那样，不带任何参数地调用sh$_{OW}$n$_{O}$tes。

如果我们测试这个代码，我们会发现修改备注是无法T.作的。这里有两个错误，即：

- 当我们输人一个不存在的备注丨D的时候，备注本会崩溃。我们永远不能相信我们的用户会输人正确的数据！
- 如果我们输人一个正确的ID,因为丨D是整型数字，但是我们菜单传人的是一个字符串，所以笔记本还是会崩溃。

后一个错误可以通过修改Notebook类的_find_note方法来解决，这个方法会用字符串来比较而不是存于备注中的整数来比较，如下：

```
def _find_note(self, note_id):
    ''* Locate the note with the given id.[11]'
    for note in self.notes:
        if str(note.id) == str(note—id):
            return note
    return None
```

我们只是在比较它们之前，简单地把输人（input)和备注的ID转换成了字符串。我们也可以把输入转换成整型数据，但是那样的话，当用户输入一个字母“a”而不是数字“1”的时候我们会遇到麻烦。

用户输入的备注1D不存在的这个问题可以通过改变两个在笔记本里的modify方法

来完成，这个方法会检查是S_find_note会返回一个备注，像这样：

```
def modify_memo(self, note_id, memo):
    '1 * Find the note with the given id and change its
    memo to the given value.f * 1
    note = self._find_note(note_id>
    if note:
        note.memo = memo
        return True
    return False
```

这个方法已经更新,取决于一个备注是否能找到，这个方法会返回True或者False。如果用户输人一个无效的备注，菜单可以使用这个返回值来显示一条错误信息。这个代码看起来有点笨拙，相反如果它能抛出一个异常的话，看起来会更好一些。我们将在第4章讲解这些。

练习

编写一些面向对象的代码。目标是使用你在这一章学到的规则和语法，来保证你不只是阅读了它，更是使用了它。如果你已经进人一个Python项目里了，回去看看是否有些地方你可以创建对象并且给它添加属性和方法。如果项目非常大，试着使用这些语法把它分成一些模块或者包。

如果你没有这么一个项目，尝试开始一个新的。它不一定是一个你想要完成的事情，只需要一些基本的设计部分。你不需要完全实现一切，往往在整个设计阶段你所需要的只是一个Print ("this method will do somethivg")方法。当你要实现不同的交互以及在真正实现它们要做的事情之前描述一下它们该如何工作时，这就是所谓的自上而下设计。相反的叫自下而上设计，先实现细节，然后把它们所有的连接在一起。两种模式在不同的时间都是有用的，但是为了理解面向对象的原则,一个自上而下的工作流更合适一些。

如果你提出新想法有些困难，试着写一个TO DO的应用。（提示：和设计笔记本应用类似，但是有额外的数据管理方法。）它可以记录每天你想做的事情，并且允许你把它们标记为完成状态。

现在，尝试设计一个更大的项目；它不需要实际做任何事，但是确保你实验了包和模块的导人语法。给不同的模块添加函数并且尝试从其他模块和包导人函数。使用相对导人和绝对导人。看看差异之处，并且尝试想象一些场景，哪一个场景需要哪一种导入方式。

总结

在这一章，我们学习了在Python中创建类并且给类分配属性和方法是多么简单的事。我们同时还介绍了访问控制以及不同级别的范围（包、模块、类以及函数）。特别是，我们讲解了：

- 类的语法。
- 属性和方法。
- 初始化函数和构造函数。
- 模块和包。
- 相对导入和绝对导入。
- 访问控制以及它的局限性。

在下一章，我们将学习如何使用继承来分享实现。

3

第3章 当对象是相似的

在编程世界，重复代码被认为是非常不好的。我们不应该在不同的地方有多份相同或者类似代码的副本D

把具有相似功能的代码片段或者对象合并起来的方式有很多种。这一章，我们将介绍一个最著名的面向对象准则：继承ˆ正如第1章所讨论的，继承允许我们在两个或者更多的类之间创建一种“是一个”的关系.这种关系把共同的细节柚象到一个超类里，特有的细节存于子类里。我们将重点介绍下酣一些Python语法和准则：

- 基本继承。
- 从内置的类继承。
- 多重继承。
- 多态和动态类型。

基本继承

从技术上讲，每一个我们创建的类都使用了继承。所有的Python类都是一个叫作object的特殊类的子类。这个类提供了非常少的数据和行为（这些它提供的行为都是以双下岡线开头的方法.这些方法都只供内部使用），但是它确实使Python以同样的方式对待所有对象。

如果我们不具体指明我们的类从其他类继承，那这个类自动从object继承过来-不管怎样，通过下面的语法，我们可以公开声明我们的类从object继承：

```
class MySubClass(object):
```

```
    pass
```

这就是继承！从技术角度讲，这个例子和我们第2章的第一个例子没什么不同，因为在Python 3中，如果我们不给它具体提供一个不同的超类，这个类就会自动从object继承。超类，或者也叫父类，是一个被继承的类。子类是一个从超类继承过来的类。在上面的例子中，超类是object，MySubClass是子类。我们可以说一个子类来源于父类，或者这个子类扩展了父类。

你可能已经从这个例子发现，在一个基本的类定义中，继承需要非常少M的额外语法。仅仅需要在类名后面的一对括号里包含父类的名字就行，但是要在冒号终止这个类的定义之前。所有我们需要做的，就是告诉Python这个新的类应该来源于这个给定的超类。

在实际中我们如何应用继承呢？最简单、最明显的使用继承的方法就是，给一个已经存在的类添加一个功能。我们从一个简单的通信录开始，这个通信录可以记录一些人的姓名和电子邮件地址。contact类负责维护一个类变量中所有联系人的列表，并且初始化姓名和地址，来看一下这个简单的类：

```
class Contact:
    all – contacts =[]

    def __init—(self, name, email):
        self.name = name
        self.email = email
        Contact. all__contacts . append (self)
```

这个例子向我们介绍了类变量。这个叫all_contacts的列表，因为它属于类定义的一部分，实际上会被所有这个类的实例共享。这就意味着这里只有一个Contact .all_contacts列表，如果我们在任何一•个对象里调用self.all_contacts，它会引用这个列表。在初始化函数里的这段代码保证了无论何时我们创建一个新的contact类，这个列表都会自动把这个对象添加进来。要小心使用这个语法，因为如果你曾经使用self.all_contacts给这个列表赋过值，你将会在这个对象里创建一个新的实例变量，这个类变M将会保持不变并且可以通过Contact.all_contacts访问到。

这是一个非常简单的类，它允许我们记录关于联系人的一些数据。但是如果我们的某些联系人同时是需要从他们那里订购东西的供应商该怎么办？我们可以给Contact类添加一个order方法，但是这样会允许人们从我们的客户或者家庭朋友这些联系人那里订购东西。相反，我们创建一个新的叫Supplier的类，和Contact类一样，但是它有额外

的order方法：

```
    class Supplier(Contact):
        def order(self, order):
            print ("If this were a real system we would send •'
                   •• { } order to { } n . format (order, self .name))
```

现在，如果可以在我们可靠的解释器里测试这个类，我们将看到所有的联系人，包括供应商，在里都接收一个名字和电子邮件地址作为输入参数，但是只有供应商有一个order功能的方法：

```
»> c = Contact ("Some Body", " somebody @exan5>le. net'1)
»> s = Supplier ("Sup Plier", "supplier@exanqple.net">
»> print<c.name, c.email, s.name, s.email)
Some Body somebody0exainple.net Sup Plier supplier@exan^>le. net
»> c. all-contacts
[< main____.Contact object at 0xb7375ecc>,
< main____.Supplier object at 0xb7375f8c>]
»> c.order ("Ineed pliers")
Traceback (most recent call last):
  File ''<stdin>", line 1, in <module>
AttributeError:    'Contact' object has no attribute 1 order *
»> s. order ("I need pliers")
If this were a real system we would send I need pliers order to
Supplier
»〉
```

那么现在我们的Supplier类可以做任何Contact类可以做的事（包括把自己添加到all_contacts这个列表里）和一个供应商需要处理的所有特殊事情。这就是继承的魅力。

扩展内置类

这种继承最有趣的应用就是给内置类增加功能。之前看到的Contact类中，我们把联系人添加到所有联系人的列表里。如果我们想通过名字搜索怎么办？那么，我们可以给Contact类添加一个方法用于搜索，但是似乎这个方法实际上属于列表本身。我们可以使用继承来做：

```
class ContactList(list):
    def search(self, name):
        '1'Return all contacts that contain the search value
        in their name.*''
        matching_contacts =[]
        for contact in self:
            if name in contact.name:
                matching_contacts.append(contact)
            return matching—contacts

class Contact:
    all_contacts = ContactList()

    def _init__(self, name, email):
        self.name = name
        self.email = email
        self•all contacts.append(self)
```

我们创建了一个新的ContactList类来扩展Python内置的List,而不是实例化一个普通的列表来作为我们的类变量,,然后我们实例化这个子类来做我们的$_{a}$ll$_{-CO}$nt$_{aC}$t$_{S}$列表。我们可以像下面这样测试这个新的搜索功能：

```
»> cl = Contact ("John A", " j ohna6 exan^>le .net")
»> c2 = Contact ("John B" , '* johnb@exan5>le. net")
»> c3 = Contact ("Jenna C" , " jennac0exan^>le. net")
»> [c.name for c in Contact•all—contacts•search(' John1)]
['John A'f  John B,]
»>
```

你想知道我们是如何改变了内置列表语法[],使之成为我们可以继承的对象吗？用[]来创建一个空的列表实际上是用list（>创建空列表的简化形式，这两种语法是相同的：

```
»> [] = list()
True
```

所以,list这个数据类型像一个我们可以扩展的类，而不像objeot。

像第二个例子一样，我们可以扩展diet类，这种创建字典的方式是烦琐的（{:}语法）O

```
class LongNameDict(diet):
    def longest_key(self):
        longest = None
        for key in self:
            if not longest or len(key) > len(longest):
                longest = key
        return longest
```

在交互式解释器中可以简单测试一下：

```
»> longkeys = LongNameDict ()
»> longkeys [' hello ' ] = 1
»> longkeys [' longest yet' ] =5
»> longkeys [' hello2 ' ] = ' world'
»> longkeys . longest_key ()
'longest yet'
```

大部分内置数据类型都可以类似地这样进行扩展。常见的可扩展的内置数据类型是object、list、set、diet file以及str。像整塑int和浮点塑float这些数值类塑偶尔也会做继承。

重写和Super

给已经存在的类添加新的行为，继承是非常好的实现方式，但是如果要改变行为呢？我们的contact类只允许一个名字和一个电子邮件地址，这对大部分的联系人可能是足够的，但是如果我们想给我们亲近的朋友增加一个电话号码怎么办？

就像我们在第2章看到的，在联系人被构建之后，我们可以非常简单地通过给它增加一个电话号码的属性来实现a但是如果我们想ih这第3个变量在初始化类的时候就可用，就需要重写_in_函数。重写就是在子类里用一个（和超类相同名字）新的方法来改变或者覆盖超类里的这个方法做这个不需要特殊的语法，会自动执行子类新创建的方法而不是超类的方法。例如：

```
class Friend(Contact):
    def _init_(self, name, email, phone):
        self.name = name
        self.email = email
        self.phone = phone
```

不只有_init_，任何方法都可以被重写。在我们继续往下讲之前，无论如何我们需要纠正这个例子里的一些问题。我们的Contact和Friend类有重复的代码去建立name和email属性;这会导致维护复杂,因为我们需要在两个或者更多的地方更新代码。更值得警惕的是，我们的Friend类忽略了把自己加到all_contacts列表里，这个列表是我们在Contact类里创建的。

我们真正需要的是一种可以调用父类代码的方法。这就是super函数的功能；它返回一个父类的实例化对象，允许我们直接调用父类的方法：

```
class Friend(Contact):
    def __init__(self, name, email, phone):
        super().__init__(name, email)
        self.phone = phone
```

这个例子里首先通过super得到父类对象的实例，并且调用这个对象的_:1111七_方法，传递给它预期的参数。然后这个类做了白己的初始化，即设置phone属性。

super（>可以在任何方法里调用，不只是_;ˆ1七_方法。这就意味着通过重写和调用super,可以修改所有的方法。可以在方法内的任何位置调用super;我们不必要总是在方法内的第I行调用。例如，在把传进来的参数传给超类之前，我们可能需要操作它们。

多重继承

多重继承是一个敏感的主题。原则上来讲，它非常简单：一个从多个父类继承过来的子类，可以访问所有父类的功能。在实践中，它并没有听起来那么有用，并且很多专家程序员推荐不要使用它。因此，我们以一个警告开始：

/ 根据经验，如果你认为你需要多重继承，你有可能是错误的，但是如果你知道你需要它，那你可能是正确的。

多重继承最简单、最有用的形式叫作mixin，一个mixin通常是一个超类，这个超类不是因为自己而存在，而是通过被其他类继承来提供额外的功能。比如，假设我们想给我们的Contact类增加一个功能，允许给self .email发送一封电子邮件。发送电子邮件是一个非常常见的任务，我们可能会想在其他的类里使用这个功能。所以我们可以写一个简羊-的mixin类来为我们做这个发电子邮件的功能。

```
class MailSender:
  def send_mail(self, message):
      print("Sending mail to " + self.email)
      #在这里添加电子邮件逻辑
```

为简单起见.我们这里不会包括实际的电子邮件逻辑；如果你有兴趣去学习这是如何做的，看一下Python标准库里的smtplib模块吧。

这个类不做任何特别的事（实际上，它仅仅起到一个独立的类的作用），但是通过多重继承，它允许我们定义一个新的类，这个类既是Contact又是MailSender:

```
class EmailableContact(Contact, MailSenderJ:
    pass
```

多重继承的语法就像类定义里的参数列表。括号里可以包含两个（或者更多的）基类而不是一个，这些基类使用逗号分隔开来3我们可以测试一下这个新的混合体来看一下mixin是如何T.作的：

```
»> e = EmailableContact ("John Smith", " j smi th@exan5>le . netn)
»> Contact • all – con tacts
[< main_.EmailableContact object at 0xb7205fac>]
»> e. send_mail ("Hello, test e-mail here")
Sending mail to jsmith@example.net
```

Contact类的初始化函数仍然会把一个新的联系人添加到all_contacts这个列表里，并且mixin可以发送电子邮件给self, email,所以我们知道一切都正常丁.作。

这倒不是很难，可能你想知道关于多重继承的警告是什么。我们可以很快知道它的复杂性，但是让我们想想除了使用mixin,这里我们还有什么选择：

- 我们可以使用单一继承并且把send_mail函数添加到子类里。这种方式的缺点就是，这个发送邮件的功能会重复出现在其他任何需要电子邮件功能的类里。
- .我们可以创建一个独立的Python函数用来发送电子邮件，并且当需要发送电子邮件的时候，通过提供一个正确的电子邮件地址作为传人参数来调用它。
- 我们可以使用monkey-patch（我们会在第7章简单讲解monkey-patch）的方法让Contact类在其被创建以后有一个send_mail方法。可以通过定义一个函数让其接收self作为参数，并且作为一个属性把这个函数传递给一个已经存在的类来实现。

当混合不同类的方法时，多重继承可以很好地T.作，但是，当我们要调用超类的方法

时，这将会变得非常混乱。为什么呢？因为这里有多个父类。你怎么知道该调用哪个呢？你怎么知道调用它们的顺序呢？

通过给我们的Friend类增加一个家庭地址的方法来探讨一下这个问题。我们要实现这个可以通过哪些方法？地址是一些代表街道、城市、国家和其他联系人相关细节的字符串集合。我们可以把这些字符串作为参数直接传递给Friend类的_init_方法。我们也可以把这些字符串先存到一个元组或者字典里，然后再把它作为单一参数传递给_init_方法。如果没有额外的功能需要添加到地址里面，这应该是最好的方法了。

另外一个选择就是，可以创建一个新的Address类来专门保存这些字符串，并且把这个类的一个实例传给Friend类的_init_方法。这种解决方法的优势就是，我们可以给这些数据添加一些行为（比如，根据地址来指点方向或者打印地图的方法）而不只是静态地存储它们。这将会利用合成，也就是我们在第丨章讨论的“有一种”关系。对于这个问题，合成一个完全可行的解决方案，它允许我们在其他的实体，比如建筑、商业、组织中重用这个Address类。

然而，继承也焙一个可行的方案，并且这也是我们想去探索的，因此让我们增加一个新的类来存放地址。我们称这个新类叫AddressHolder而不是Address，因为继承定义了一个“是一种”的关系。说一个Friend是Address是不对的。但是因为一个朋友可以有一个Address,那么我们可以提出一个Friend是一个AddressHolder。后面，我们可以创建其他实体（公司、建筑），它们也可以持有地址。下面就是我们的AddressHolder 类：

```
class AddressHolder:
    def —init__ (self, street, city, state, code):
        self.street = street
        self.city = city
        self.state = state
        self.code = code
```

非常简单；我们只是把所有数据放入实例化变M的时候赋给了初始化方法。

钻石的问题

但是我们在这个已经存在的从Contact类继承过来的Friend类中该如何操作呢？当然是多重继承了。棘于-的部分是我们现在有两个父类的_init_方法需要被初始化。并且它们要通过不同的参数来初始化。我们该如何做呢？好吧，让我们从一个天真的方法

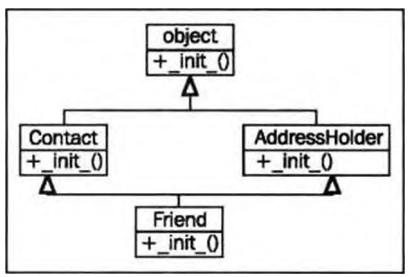
开始:

```
class Friend(Contact, AddressHolder):
    def __init__(self, name, email, phone,
                 street, city, state, code):
        Contact._init-(self, name, email)
        AddressHolder.__init-(
            self, street, city, state, code)
        self.phone = phone
```

在这个例子里，我们直接调用了每一个超类的_init_J^r法并且显式地传递了 self 参数。这个例子从技术层面上讲是可以工作的；我们可以在这个类里直接访问不同的变量。恒是这里存在一些问题。

首先，如果我们忽略显式地调用初始化函数可能会导致一个超类未被初始化。在这个例子里这还不是太大的问题，但是在一些常见的场景里这会导致程序崩溃。想象一下，比如，试图把数据插到一个未连接的数据库里。

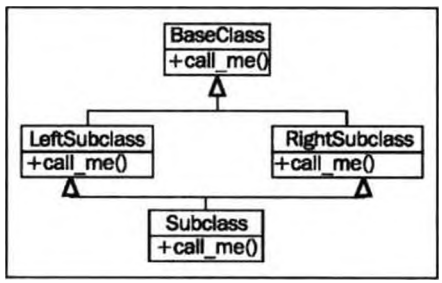
第二，更可怕的是，由于这个类的层次结构，可能会导致某个超类被调用多次。看下面的继承关系图：



Friend类的_init_方法首先调用了 Contact类的_init_方法，隐式初始化了object超类（记住，所有类都继承于object类）。Friend类然后调用了AddressHolder的_init_方法，又一次隐式初始化了 object超类。父类被创建了两次。在我们这个情况下，它是相对无害的，但是在一些情景里，它可能会带来灾难。想象一下，每次请求视图连接数据库两次！基类只应该被调用一次。一次，是的，那么在什么时候调用呢？是先调用Friend,然后Contact,接着Object,之后AddressHolder?还是先调用 Friend,然后 Contact,接着 AddressHolder,之后 Object?

从技术上来讲，每一个方法的调用顺序，可以通过修改这个类的_1111'0_（方1法解析顺序）属性来动态地实现修改，这已经超出了本书的讲解范围如果你*认为你需要理•解它，氣推荐 Expert Python Programming, Tarek Ziade.Packt*■，或者阅读Python官方文档的这个主题：http://www.python,org/download/releases/2.3/mro/。

让我们看第二个人为的例子来更清晰地说明这个问题。这里我们有一个基类，这个基类有一个方法叫call_me。有两个子类重写了这个方法，然后第3个类通过多重继承扩展了这两个方法=这称为钻石继承，因为这个类的视图是一个钻石的形状：



这个钻石的例子会让多重继承变得非常棘手。从技术上讲，在Python 3中所有多重继承都是钻石继承，因为所有的类都从object继承，前面的图例中，使用object._init_同样是一个钻石问题

把上面的图转化成代码，下面的例子展示了这些方法在什么时候被调用：

```
class BaseClass:
    num_base_calls = 0
    def call_me(self):
        print("Calling method on Base Class")
        self.num_base_calls += 1

class LeftSubclass(BaseClass):
    num_left_calls = 0
    def call_me(self):
        BaseClass.call_me(self)
        print("Calling method on Left Subclass")
        self.num_left_calls += 1

class RightSubclass(BaseClass):
```

```
        num_right_calls = 0
        def call_me(self):
            BaseClass.call - me(self}
            print("Calling method on Right Subclass")
            self.num_right_calls +=1

    class Subclass(LeftSubclass, RightSubclass):
        num_sub_calls = 0
        def call_me(self):
            LeftSubclass•call- me(self}
            RightSubclass•call- me(self)
            print("Calling method on Subclass")
            self.num_sub_calls +=
```

这个例子简单地保证了每一次重写call_me方法都是直接使用同样的名字调用父类里的方法。每一次它被调用，通过把信息打印到屏幕上来告知我们，并且在类里面更新一个静态变11来显示它被调用了多少次=如果我们实例化一个Subclass对象并且调用一次call_me方法，我们会得到下而的输出：

```
»> s = Subclass ()
»> s . call_me ()
Calling method on Base Class
Calling method on Left Subclass
Calling method on Base Class
Calling method on Right Subclass
Calling method on Subclass
»> print(s.num_sub_calls, s•num -left-calls, s.num -right -calls,
s•num— base-calls)
1 1 1 2
» >
```

基类的call_me方法被调用了两次。这不是我们所期待的行为，并且如果这个方法正在做实际的丁.作，这将导致非常严重的bug—-像往一个银行账户存款两次一样。

对于多重继承我们需要记住的是，在类的层次结构中，我们只想调用“下一个”方法，而不是“父类”的方法。实际上，下一个方法可能不属于当前类或者当前类的父类或者祖先类。关键字super再次拯救我们。确实，super最初被开发出来就是为了让形式复杂的多重继承成为可能a下面是用super写的相同代码：

```
class BaseClass:
    num_base_calls = 0
    def call_me(self):
        print ("Calling method on Base Class'*)
        self. num_base__calls += 1

class LeftSubclass(BaseClass):
    num_left_calls = 0
    def call-me(self):
        super()•call- me()
        print("Calling method on Left Subclass")
        self.num_left_calls += 1

class RightSubclass(BaseClass):
    num__right_calls = 0
    def call—me(self):
        super()•call- me()
        print("Calling method on Right Subclass")
        self.num_right_calls +- 1

class Subclass(LeftSubclass, RightSubclass):
    num_sub_calls = 0
    def call_me(self):
        super()•call—me()
        print ("Calling method on Subclass'*)
        self.num_sub_calls += 1
```

改变非常小；我们简单地把直接调用变成了调用super（>方法。这已经非常简单了，但当我们执行它时，让我们看看不同之处：

```
»> s = Subclass ()
»> s . call_me ()
Calling method on Base Class
Calling method on Right Subclass
Calling method on Left Subclass
Calling method on Subclass
»> print(s•num -sub-calls, s•num -left -calls, s•num -right -calls,
s . num-base-calls)
```

看起来很好，我们的基类只被调用丫一次。但是super（>方法在这里到底做了什么呢？因为在super调用之后我们执行了 print语句，输出被打印出来的顺序就是每一个方法实际上被执行的顺序。我们从后往前看一下输出来看看谁调用什么。

首先，Subclass的call_me方法调用了 super ().call_me()，其实就是引用了LeftSubclass.call_me( ) 方法。然后 LeftSubclass. call_ me( ) 调用了super（> . call一me ()，但是这时 super <)弓|用了 RightSubclass . call_me（> O 要特别注意；super调用并不是调用Lef tSubclass的超类（就是BaseClass ）的方法，它是调用RightSubclass，虽然它不是LeftSubclass的父类！这就是下一个方法，而不是父类方法。RightSubclass然后调用BaseClass，并且通过super调用保证了在类的层次结构中每一个方法都被执行一次。

不同的参数集合

当我们回到Friend多重继承的例子.你能看到这使事情变得复杂了吗？在Friend的_1^^1：_方法里，我们最初是用不同的参数集合，直接调用两个父类的_:^1七_方法：

```
Contact.___init__(self, name, email)
AddressHolder.____init__(self, street, city, state, code)
```

我们如何把它变成了使用super方法？我们不需要知道super尝试去首先初始化哪个类。即使我们需要知道，我们也需要一个方式来传递“额外”的参数，以便后续在其他子类里调用super苻正确的参数。

特别要说的是，如果第一次凋用super传递了 name和email作为参数给Contact._init_，jf-且 Contact. _init_接着调用了 super,它需要能够传递地址相关参数给“下一个”方法，也就是AddressHolder._init_。

当我们想用相同的名字但不同的参数集合，来调用超类方法的时候，会有一个问题。通常来讲，使用完全不同的参数集合来调用一个超类，唯一一次就是在_；^1亡_方法里，就跟现在我们做的一样即使对于普通方法，尽管我们可能想添加可选参数给它们，但是这只冇在子类或者一组子类里才有意义。

很不幸的是，解决这一问题唯一的方法就是从一开始就做好计划。我们必须设计好基类的参数列表，这样它们接收任何不是每一个子类实现所必需的参数作为关键字参数。我们也必须确保方法能接收在super调用中并不期望和传递的参数，以避免对于在继承顺序

上排在后面的方法需要它们。

Python函数的参数语法提供了我们要实现这些所需的所有工具，但是它使整个代码变得烦琐。il:我们看看Friend多重继承代码的正确版本：

```
class Contact:
    all_contacts =[]

    def __init__(self, name= * , exnail=' ' , **kwargs):
        super () .__init__(**kwargs)
        self.name = name
        self.email = email
        self.all_contacts.append(self)

class AddressHolder:
    def __init__(self, street:1*, city='1, state='*, code='',
                 **kwargs):
        super 0 .__init__(**kwargs)
        self.street = street
        self.city = city
        self.state = state
        self.code = code

class Friend(Contact, AddressHolder):
    def __init__(self, phone='', **kwargs):
        super () .__init__(**kwargs)
        self.phone = phone
```

通过设置空字符串为参数默认值，我们已经把所有参数变成了关键字参数。我们也保证包含了一个**kwargs参数，它可以捕获任何特殊方法不知道如何处理的额外参数.:，通过调用super方法，它把这些参数传递给了下一个类。

t 如果你不熟悉**kwargs语法，它主要是收集任何传递到方法但是没有在参数ˆ列表中显式列出的关键字参数。这些参数会被存于一个叫kwargs（我们可以随意称呼这个变量，但是通常叫kw或者kwargs)的字典里。当我们调用一个携带* *kwargs语法的不同方法（例如super._____ init__)，它会打开这个字典并且把结果以标准关键字参数的形式传给这个方法。我们会在第7章中讲解它的细节

前面的例子做了它应该做的。但是它开始变得混乱并且难以回答这个问题，“哪些参数我们需要传递给Friend.”对于任何想使用这个类的人来说这是最重要的问题，所以应该给这个方法添加一个docstring格式的注释来解释发生了什么。

进一步来讲，如果我们想在父类中“重用”这个变量，这种实现方式甚至是不够的。当我们传递**kwargs变量给super，这个字典并不包括任何包含在显式关键字参数中的变量。例如，在Friend._init_方法里，调用super方法并没有在kwargs字典里包含phone参数。如果任何其他类需要phone参数，我们需要保证它在传递的这个字典里。更糟糕的是，如果我们忘记这么做，这将很难调试，因为超类将不会报错，但是会简单地给这个变量赋一个默认值（在这个例子里，是一个空字符串）。

这里有一些方法来保证向上传递的变量。假设Contact这个类，出于某种原因需要在初始化的时候携带一个电话号码参数，同时Friend类也需要能访问它。我们可以做下面的事情：

- 不要把phone包含在S式关键字参数里。相反，把它放到kwargs字典里 Friend类可以通过kwargs ['phone M语法查找它。肖它把**kwargs传递给super调用时，phone参数也会包含在这个字典里。
- 让phone作为显式关键字参数，但是在把它传给super之前，使用标准的字典语法 kwargs [¹ phone ¹] =phone 来更新 kwargs 字典„
- 让phone作为陆式关键字参数，但是使用kwargs • update方法更新kwargs字典如果你布多个参数需要更新，这种方法是很有帮助的。你可以使用diet (phone=phone)构造闲数或者使用字典的语法{ 'phone ¹ : phone }来创建一个字典，并作为参数传递给update调用。
- 〖丨:phone作为显式关键字参数，但是通过语法super()._init_(phone=phone, **kwargs)显式地把它传给super调用。

我们已经介绍了很多在Python涉及多重继承的注意事项。当我们需要考虑所有的情况时，我们应该为它们做计划，我们的代码也会变得很乱。基本的多重继承是很方便的，但是在很多情况下，我们需要考虑一个更透明的方式来把两个不同的类结合起来，通常会使用我们在第8章和第9章将会介绍的设计模式之一或者组合。

多态

我们在第1章介绍了多态。这个有趣的名字描述了一个简单的概念；调用不同的子类

将会产生不同的行为，而无须明确知道这个子类实际上是什么。比如，想象一个可以播放音频文件的程序。媒体播放器可能需要加载一个AudioFile对象然后play它。我们把一个play<)的方法放到这个对象里，它负责解压或者提取音频，然后把音频引导到声卡或者扬声器。一个播放AudioFile的行为可以如下这么简单：

```
audio_file.play()
```

然而，对于不同类型的文件，解压和提取音频文件的过程是很不一样的。.wav文件存储未压缩的音频，.mp3、.wma和.ogg文件都有着不N的压缩算法。

我们可以使用多态和继承来让设计简单化。每种不同类型的文件都可以表示为一个AudioFile的不同子类，例如，WavFile、MP3File。每一个都会有一个play（>方法，但这个方面针对每个文件的实现是不同的，以确保后继流程既正确又准确。媒体播放器对象永远不需要知道它引用了 AudioFile的哪一个子类；它只是调用play(>方法然后多态地让对象去处理实际播放的细节=■让我们看一个快速的骨架程序如何展示这个：

```
class AudioFile:
    def —init___(self, filename):
        if not filename.endswith(self.ext):
            raise Exception("Invalid file format")

        self.filename = filename

class MP3File(AudioFile):
    ext = "mp3"
    def play(self):
        print("playing {} as mp3M.format(self.filename))

class WavFile(AudioFile):
    ext = "wav"
    def play(self):
        print("playing {} as wav".format(self.filename))

class OggFile(AudioFile):
    ext=Moggn
    def play(self):
        print ("playing { } as ogg,f. format (self. filename))
```

所有音频文件的检查确保了初始化的一个有效扩展。但是注意到，如何能让父类的_init_方法去访问来自不同子类的ext类变量？这就是多态性的工作。如果文件并没有以正确的名字结尾，就会抛出一个异常（异常的细节会在下一章介绍）。事实上，AudioFile没有存储ext变量的引用这一事实并不能阻止它在子类中访问。

此外，每一个AudioFile的子类会以不同的方式实现play ()方法（这个例子实际上并不播放音乐；音频的压缩算法真的可以值得单独一本书来讲）这也是多态在起作用。媒体播放器可以使用完全相同的代码来播放文件，无论它是什么类型；它不在乎它在使用AudioFile的哪一个子类。解压音频文件的细节被封装了。如果我们测试这个例子，它可以如我们希望的那样工作：

```
»> ogg = OggFile ("myfile. ogg")
»> ogg.playO
playing xoyfile. ogg as ogg
»> mp3 = MP3File ("myfile .mp3")
»> mp3. play ()
playing myfile.nqp3 as n^>3
»> not__an__n5>3 = MP3File ("myfile. ogg")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "polymorphic-audio .py" , line 4, in _____ init__
    raise Exception("Invalid file format">
Exception: Invalid file format
```

看到AudioFile.法在不知道是哪个子类在引用它的情况下，可以检查文件类型了吗？

多态实际上是面向对象编程里最酷的一件事，它使得一些编程设计变得非常显而易见，这在早期的范例中是不可能的。然而，因为鸭子类型 duck typing), Python使得多态不是那么酷D Python中的鸭子类型允许我们使用任何提供所需行为的对象，而不需要迫使它成为一个子类。Python的动态特性使这些微不足道。下面的例子没有扩展AudioFile,但是可以在Python中使用完全相同的接口与之交互：

```
class FlacFile:
    def __init__(self, filename):
        if not filename.endswith(".flac"):
        raise Exception("Invalid file format")
```

```
        self.filename = filename

    def play(self):
        print ("playing {} as flac'*. format (self. filename)
```

我们的媒体播放器可以播放这个对象，就像扩展AudioFile 一样简单。

在很多面向对象的上下文中使用继承，多态是最重要的原因之一。因为任何提供正确接口的对象都可以在Python中交替使用，它减少了多态的一般超类的需求。继承仍然可以用来共享代码，但是如果所有被共享的都是公共接口，鸭子类型就是所有所需的。这减少了继承的需要，同时也减少了多重继承的需要；通常，当多重继承似乎是一个有效方案的时候，我们只需要使用鸭子类型去模拟多个超类之一就可以了。

当然，仅仅因为一个对象满足特定的接口（通过提供必需的方法或者属性）并不意味着它就会在任何情况下只是T.作。它需要以一种在整个系统层面有意义的方式实现该接口。仅仅因为一个对象提供了一个叫play（>的方法并不意味着它就会自动地同媒体播放器工作。例如，我们第1章中的象棋A1对象，它有一个可以移动棋子的play（>方法。尽管它满足接口要求，如果我们试图把这个方法放进媒体播放器里，这个类还是町以以特殊的方式异常退出。

鸭子类型另外一个有用的特性就是，鸭子类型的对象只需要提供那些真正要访问的方法和属性。例如，如果我们需要创建一个假的文件对象来读取歌面的数据，我们可以创建一个新的有一个read 方法的对象；如果将要与对象交互的代码只会去读文件，那么我们不需要重写write方法。更简洁的是，鸭子对象不需要提供一个可用对象的完整接口，它只需要实现实际使用的接口就行了。

案例学习

让我们试着用一个更复杂的例子把我们学到的东西串起来。我们将设计一个简单的房地产应用程序，这个应用程序允许一个代理来管理用于购买或者租赁的房产。会有两种类型的房产：公寓和房子代理需要能够输入一些新房产的相关细节，列出P前所有手头的房产，并且把一个房产标记成售出或者已租。为了简便起见，我们不需要担心编辑房产的具体细节或者被出售之后重启激活房产。

这个项目将会允许代理通过Python解释器命令提示符来和对象交互。在这个阁像用户界面以及Web应用程序的世界里，你可能会想，为什么我们要创建看起来如此落后的项目。

简单来说，为了实现需求,窗口程序以及Web应用程序都需要大量的知识开销和样板代码。如果我们要用这些范例中的一种来开发程序，我们可能会迷失在"GUI编程"或者"Web编程"中，而忽视了我们试图去掌握的面向对象原则。

幸运的是，大多数GUI和Web框架都是利用面向对象的方法，并且我们现在学习的原则有助于将来我们去理解这些系统„我们会在第12章中简单讨论它们两个，但是完整的细节远远超出了一本书的范围。

我们的需求看起来包含了相当多的名同，这些名词可能代表了我们系统里的对象的类。显然我们需要代表一个房产。房子和公寓可能需要单独的类。出租和购买似乎也需要单独表示。因为我们现在专注于继承，所以我们将会寻找一些方法来分享使用继承和多重继承的行为。

显然，House和Apartment都是房产的类型，所以Property可以是它们两个类的父类。Rental和Purchase需要额外考虑；如果我们使用继承，我们就需要单独的类，例如，对于House Rental和HousePurchase,并且使用多重继承来组合它们。相比于组合/关联设计，这看起来有点笨，但是让我们跑跑看，看看我们能想出什么。

现在，哪些属性可以和Property相关联？不管它是一个公寓或者房子，大多数人都会想知道面积，卧室以及浴室的数量。（还有很多其他类似的属性，但是我们将会使我们的原型保持简单。）

如果房产是一所房子，它将需要告知仓库的数量，是否有车库（连接着、分离的，或者没有），以及院子是否有围墙。一套公寓需要显示它是否有一个阳台，并且如果是套间的话，朝向是什么。

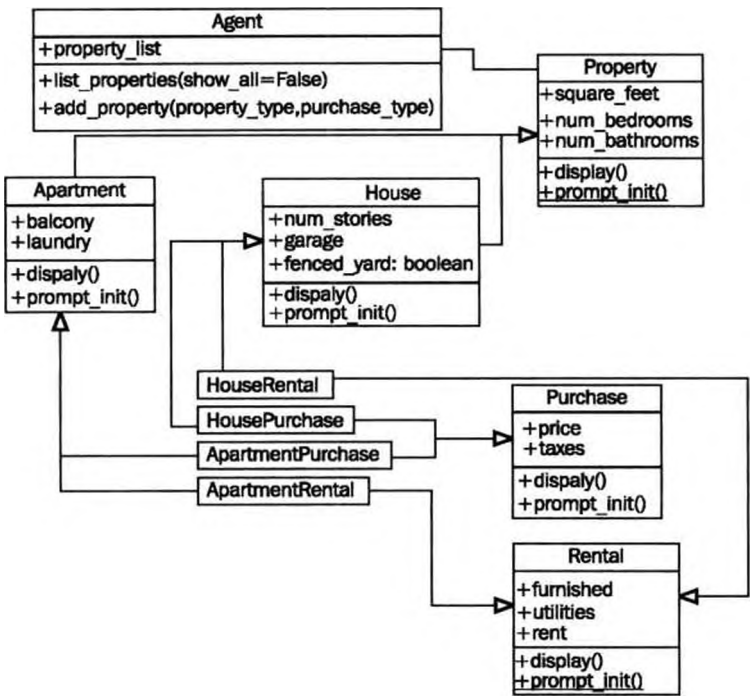
两种房产类型都需要显示房产特征的方法。目前，没有其他行为是显而易见的。

租赁房产需要存储铒月的房租，房产是否包含家具，以及是否包含公共设施，如果没有，需要估计些什么。用于出售的房产需要存储购买的价格以及估计每年的物业税。对于我们的应用来讲，因为我们只需要显示这些数据，所以就与其他的类相似，我们只需要添加一个display（>方法就可以了。

最后，我们需要一个Agent对象来占有所有房产的列表，显示他们的房产，并且允许我们创建新的房产。创建房产将需要提示用户每一个房产类型的相关细节。这可以通过Agent对象实现，但是然后Agent对象需要知道很多关于房产类型的信息。这不是多态的优势。另一个相关的地方可能是需要在每一个类的初始化函数或者甚至构造函数里放置这些房产提示信息，但是将来在GUI或者Web应用程序中，是不会允许类这样使用的。一个更好的主意就是，创建一个静态的方法，这个方法会做一些提示并且返回一个提示参数

的字典。然后所有Agent要做的就是提示用户房产的类型以及支付方式，并且让正确的类来实例化它们自己。

对于一个简单的应用，这个设计看起来有点复杂！下面这个类的视图可能会把我们的设计决策表现得更清晰一些：



哇，有这么多多重继承的箭头！如果没有箭头的穿插，我认为不会有其他方法可以添加另一个级别的继承关系。即使是在设计阶段，多重继承也是很复杂的事情。

显然，这些类里面最棘手的部分就是确保在继承层次结构里调用超类的方法。让我们从实现Property开始：

```
class Property:
    def __init__(self, square_feet='', beds='',
                 baths='', **kwargs):
        super().__init__(**kwargs)
        self.square_feet = square_feet
        self.num_bedrooms = beds
```

```
        self.num_baths = baths

    def display(self):
        print("PROPERTY DETAILS")
        print (''==========M)
        print("square footage: {}".format(self.square_feet))
        print("bedrooms: {}M.format(self.num_bedrooms))
        print ("bathrooms: { }f'. format (self.num_baths))
        print()

    def prompt_init():
        return diet (square_feet=input (''Enter the square feet: ·),
                beds=input ("Enter number of bedrooms: ••),
                baths=input ("Enter number of baths: •'>}
    pron^jt-init = staticmethod(proinpt- init)
```

这个类还是非常简单的。我们已经给_init_方法额外地添加了**kwargs参数，因为我们知道会在多重继承的情况下使用它。我们也包含了调用super（> 方法，以避免在多重继承链中我们不是最后一个调用。在这种情况下，我们在“消费”关键字参数，因为我们知道，在其他级别的继承层次结构中是不需要它们的。

在PrOmPt_init方法中我们看到了新东西。这个方法在创建之初就立即设置成了静态方法。静态方法只和一个类（就像类的变量）关联，而不是和一个具体的对象实例。因此，它们没有self参数。正因为如此，super关键字就行不通了（这里没有父类对象，只有父类），所以我们简单地在父类里直接调用这个静态方法。这个方法使用了 Python diet 构造函数来创建可以传递到法的字典的值。字典的每个键的值通过调用input方法来提示输入。

Apartment类扩展了 Property类，在结构上是类似的：

```
class Apartment(Property):
    valid_laundries = ("coin", "ensuite", "none")
    valid_bal conies = ("yes"."no", "solarium'*)

    def __init__(self, balcony='', laundry^*', **kwargs):
        super() ._init__(* * kwargs)
        self.balcony = balcony
        self.laundry = laundry
```

```
def display(self):
    super().display()
    print ('*APARTMENT DETAILS")
    print("laundry: %s" % self.laundry)
    print("has balcony: %s" % self.balcony)

    parent_init = Property.prompt_init()
    laundry = '*
    while laundry.lower() not in \
            Apartment•valid-laundries:
        laundry = input("What laundry facilities does "
                "the property have?( ".format (
                ","•join(Apartment•valid-laundries))
    balcony = 11
    while balcony.lower() not in \
            Apartment.valid—balconies:
        balcony = input(
             "Does the property have a balcony?"
             •' {}}". format (
             ",".join(Apartment,valid_balconies))
    parent_init.update({
        "laundry": laundry,
        "balcony": balcony
    })
    return parent-init
prompt_init = staticmethod(prompt_init)
```

display（）和_init_（>方法通过super（>分别调用T它们父类的方法来确保Property类正确地被初始化。

静态方法pr○mpt_init现在从它的父类获取了字典值并且之后添加了一些属于它自己的字典值。它通过调用diet. update方法把新的字典值合并到第一个字典里u然而，这个prompt_init方法看起来很丑陋；它会循环两次，直到用户通过使用结构类似的代码输人两次有效但是值不同的输人为止。如果把这个验证逻辑提取出来会比较好，那样我们可以只在一个地方维护它；对于后面的类也将会有帮助。

对于所有关于继承的讨论，我们可能会想这是使用mixin的好机会。相反，我们有机会学习到一种情况，在这种情况，继承并不是最好的解决方案。我们要创建的方法将会用

在一个静态方法里面。如果我们要从一个提供了验证功能的类来继承，那么这些功能也会作为静态方法提供给我们，而没有去访问这个类里的任何实例变量。如果它没有访问任何实例变A，那使其成为一个类有什么意义呢？为什么我们不把这个验证功能做成一个模块级的函数，这个函数接收一个输入字符串并且列出有效的答案，然后就是这样吗？

让我们探索一下这个验证函数会是什么样子：

```
def get_valid_input(input_string, valid_options):
    input_string += " ({}) ".format(", ".join(valid_options))
    response = input(input_string)
    while response.lower() not in valid_options:
        response = input(input_string)
    return response
```

我们可以在解释器里测试这个函数，独立于其他所有我们使用的类。这是一个好的迹象，这意味着我们设计的不同模块没有紧密耦合，而是可以在后期不影响其他代码片段的情况下独立改进。

```
>>> get_valid_input("what laundry?", ("coin", "ensuite", "none"))
what laundry? (coin, ensuite, none) hi
what laundry? (coin, ensuite, none) COIN
'COIN'
```

现在，让我们快速地更新一下我们的Apartment.prompt_init方法，使用下面这个新函数进行验证：

```
    def prompt_init():
        parent_init = Property.prompt_init()
        laundry = get_valid_input(
                  "What laundry facilities does"
                  "the property have?",
                  Apartment.valid_laundries)

        balcony = get_valid_input(
            "Does the property have a balcony? ",
            Apartment.valid_balconies)
        parent_init.update({
            "laundry": laundry,
            "balcony": balcony
```

```
        })
        return parent_init
    prompt_init = staticmethod(prompt_init)
```

相比于我们最初的版本，这个更容易阅激以及维护)。现在我们已经准备好构建House类。这个类和Apartment有着平行的结构，但是会引用不同的提示和变

```
class House(Property):
    valid_garage = ("attached", "detached", "none")
    valid_fenced = ("yes", "no")

    def _init_(self, num_stories='*,
            garage= * 1, fenced='',   kwargs):
        super().—init (  kwargs)
        self.garage = garage
        self.fenced = fenced
        self.num_stories = num_stories

    def display(self):
        super() .displayO
        print("HOUSE DETAILS")
        print (''# of stories : {} '*. format (self. num_stories))
        print ("garage: {} '*. format (self.garage))
        print("fenced yard: {}M.format(self.fenced))

    def prompt_init<>:
        parent_init = Property.prompt_init()
        fenced = get_valid_input("Is the yard fenced? ",
                    House.valid_fenced)
        garage = get_valid_input (*'Is there a garage? ",
                House .valid__garage)
        num_stories = input ("How many stories? ")
        parent_init.update({
            "fenced": fenced,
            "garage": garage,
            "num_stories": num_stories
    })
    return parent_init
```

```
prompt_init = staticmethod(prompt_init)
```

这里没有我们需要探索的新知识，所以让我们继续构建Purchase和Rental类。尽管有着明显不间的S的，但是就像我们讨论的，它们在设计上也是类似的：

```
class Purchase:
    def —init—(self, price=' 1, taxes= * ', **kv/args):
        super().—init (* * kwargs)
        self.price = price
        self.taxes = taxes

    def display(self):
        super() .displayO
        print ("PURCHASE DETAILS'1)
        print("selling price: {}".format(self.price))
        print("estimated taxes: {}".format(self.taxes))

    def prompt_init():
        return diet(
            price=input ("What is the selling price? '•),
            taxes=input("What are the estimated taxes?"))
    prompt_init = staticmethod(prompt_init)

class Rental:
    def _init—(self, furnished=' ', utilities25'',
            rent= *', **kwargs):
        super().—init—(  kwargs}
        self.furnished = furnished
        self.rent = rent
        self.utilities = utilities

    def display(self):
        super() .displayO
        print("RENTAL DETAILS")
        print("rent: {}H.format(self.rent))
        print("estimated utilities: {}M.format(
            self.utilities))
```

```
        print ("furnished: {} *'. format (self. furnished))

    def prompt_init():
        return diet(
            rent=input("What is the monthly rent?")),
            utilities=input(
                "What are the estimated utilities? ••},
            furnished = get_valid_input(
                "Is the property furnished? ",
                    ("yes", "no" n
    prompt_init = statiemethod(prompt_init)
```

这两个类没有超类（除了object这个默认超类以外），但是我们仍然调用了super（）._init_方法，因为它们会和其他类结合到一起，并且我们不知道super方法调用的顺序。用于House和Apartment的接口是类似的，当我们在单独的子类里把这4个类的功能组合到一起的时候，这个接口非常有帮助。例如：

```
class HouseRental(Rental, House):
    def prompt_init():
        init = House.prompt_init()
        init.update(Rental.prompt_init())
        return init
    prompt_init = statiemethod(prompt_init)
```

这有点奇怪，因为它既没有一个方法，也没有一个display方法！因为它的两个父类在它们的方法里恰当地调用了 super方法，我们只需要扩展这些类，并且以正确的顺序调用这些类就行了。当然这不是p_rOm_pt_init的情况，因为它是一个不会调用super的静态方法，所以我们明确地实现了这个方法。在我们写其他3个组合之前，我们应该通过测试来保证这些类将正常丁作：

```
»> init = HouseRental .pron5>t_init ()
Enter the square feet: 1
Enter number of bedrooms: 2
Enter number of baths: 3
Is the yard fenced? (yes, no) no
Is there a garage? (attached, detached, none) none
How many stories? 4
```

```
What is the monthly rent? 5
What are the estimated utilities? 6
Is the property furnished? (yes, no) no
»> house = HouseRental (^♦init)
»> house. display ()
PROPERTY DETAILS

square footage: 1
bedrooms: 2
bathrooms: 3

HOUSE DETAILS
# of stories: 4
garage: none
fenced yard: no

RENTAL DETAILS
rent: 5
estimated utilities: 6
furnished: no
```

它看起来工作正常。prompt_init方法会提示我们初始化所有超类，并且display ()方法也合作地调用了所有3个超类。

> 注意：在前面的例子中，继承类的顺序是非常重要的。如果我们写了 class HouseRental (House, Rental)而不是 class HouseRental (Rental, House)的话，display ()方法就不会调用Rental • display ()！在我们这个HouseRental版本中当调用display方法时，它指向了 Rental版本里的方法，这个方法会调用super .display ()来得到House版本，House又会调用super • display (>来得到正确的版本.•如果我们反过来，显示将会指向House类的display ()方法。当调用super方法的时候，它调用了 Property父类的方法但是Property在它的display方法里并没有调用super。这就意味着Rental类的display方法将不会被调用！按照我们的顺序放置继承列表，我们确保了 Rental调用super，来处理House这边的继承关系.：> 你可能会想，我们可以给Property .display ()方法添加一个super的调用，但是这样行不通，因为Property下一个超类是object， object并没有一个display方法〇

解决这个问题的另一个方法是允许Rental和Purchase 4广展Property类而不是直接从object继承。（或者我们可以动态地修改方法的解析顺序，但是这已经超出了本书的范围。）

现在我们已经完成了测试，可以准备创建剩下的组合子类了：

```
class J^partmentRental (Rental, Apartment):
    def prompt_init():
        init = Apartment,prompt_init()
        init.update(Rental.prompt_init())
        return init
    prompt__init = staticmethod (prompt__init)

class ApartmentPurchase (Purchase, i^artment):
    def prompt nit ():
        init = Apartment.prompt_init()
        init.update(Purchase.prompt_init())
        return init
    prompt_init = staticmethod(prompt_init)

class HousePurchase(Purchase, House):
    def prompt_init():
        init = House.prompt_init()
        init.update(Purchase.prompt_init())
        return init
    prompt_init = staticmethod(prompt_init)
```

这应该是我们的设计方式中最紧张的时候！现在我们需要做的是创建Agent,它负责创建新的列表并且显示现有的列表。让我们从简单的存储以及列清单开始：

```
class Agent:
    def __init__(self):
       self.property—list =[]

   def display—properties(self>:
       for property in self.property_list:
           property.display()
```

添加一个房产首先需要查询这个房产的类型和房产是否用于出售或者出租。我们可以通过显示一个简单的菜单来做这个。一旦已经确定，我们可以提取正确的子类并且通过使用我们已经开发的**pr○mpt_init**层次结构来提示所有细节。听起来简单？确实。让我们从通过给**Agent**类添加一个字典类变量开始：

```
type_map = {
    ("house"f ''rental"): HouseRental,
    ("house", "purchase"): HousePurchase,
    ("apartment'*, "rental"): ApartmentRental,
    ("apartment", "purchase"): ApartmentPurchase
```

这是一些非常有趣的代码。这是一个字典，它的键有两个不同字符串的元组，它的值是类对象。类对象？是的，类可以传递、重命名，以及可以像“正常”对象或者原始数据类型那样存到容器里。用这个非常简单的字典，我们可以简单地截取我们之前的**get_valid_input**方法来确保我们得到了正确的字典键，并且查找适当的类，像这样：

```
def add-property(self>:
    property_type = get_valid_input(
            "What type of property? ",
             ("house", "apartment*') ) . lower ()
    payment_type = get_valid_input(
            "What payment type? ",
             ("purchase", "rentalf,) ) . lower ()

    PropertyClass = self.type_map[
        (prope r ty-type, payment__type)]
    init_args = PropertyClass.prompt_init()
    self.property -list.append(PropertyClass(**init_args)>
```

这个看起来也很有趣！我们在字典里查找这个类并且把它存到一个名为PropertyClass的变M里。我们并不确切知道哪个类可用.但是类本身自己知道，所以nf以以多态的方式调用prompt_init方法来得到合适的传递到构造闲数的字典的值n然后我们使用关键字参数语法来把字典转换成参数并且构造新的对象来加载正确的数据。

现在我们可以使用这个Agent类来添加和查看房产列表。添加一些特性来使得一个房产可用或者不可用，或者编辑和删除房产，这不会需要太多的工作。现在我们的原型处于

一个足够好的状态，可以拿去给一个真正的房产经纪人并且演示它的功能„看看演示是如何丁_作的：

```
»> agent = Agent ()
»> agent. add_jproperty ()
What type of property? (house, apartment) house
What payment type? (purchase, rental) rental
Enter the square feet: 900
Enter number of bedrooms: 2
Enter number of baths: one and a half
Is the yard fenced? (yes, no) yes
Is there a garage? (attached, detached, none) detached
How many stories? 1
What is the monthly rent? 1200
What are the estimated utilities? included
Is the property furnished? (yes, no) no
»> agent. add_property ()
What type of property? (house, apartment) apartment
What payment type? (purchase, rental) purchase
Enter the square feet: 800
Enter number of bedrooms: 3
Enter number of baths: 2
What laundry facilities does the property have? (coin, ensuite,
one) ensuite
Does the property have a balcony? (yes, no, solarium) yes
What is the selling price? $200,000
What are the estimated taxes? 1500
»> agent. display_jproperties ()
PROPERTY DETAILS

square footage: 900
bedrooms: 2
bathrooms: one and a half

HOUSE DETAILS
# of stories: 1
garage: detached
```

```
fenced yard: yes
RENTAL DETAILS
rent: 1200
estimated utilities: included
furnished: no
PROPERTY DETAILS

square footage: 800
bedrooms: 3
bathrooms: 2

APARTMENT DETAILS
laundry: ensuite
has balcony: yes
PURCHASE DETAILS
selling price: $200,000
estimated taxes: 1500
>>
```

练习

看看你工作空间周围的物理对象并且看看你能否用一个继承层级来描述它们。几个世纪以来，人类已经把这个世界划分成了不同的种类，所以应该不会很难描述。对象类之间有不明显的继承关系吗？如果你要在计算机应用程序中建模这些对象，它们会分享哪些属性和方法？哪些必须要以多态的方式重写？它们之间哪些属性是完全不同的？

现在，来编写一些代码，不，不是对物理层次结构；那会是无聊的。比起方法，物理的东西有着更多的属性。想想在过去一年里你一直想解决但还没有开始的一个宠物编程项n。无论你想解决什么样的问题，试着想一下基本的继承关系3然后实现它们。也要确保你一定要注意各种关系，那些实际上不需要使用继承的关系！有没有什么地方你可以使用多重继承？你确定吗？你能看到有什么地方你想使用mixin吗？试着快速撞击出一个原型。这个原型不一定是有用的或者部分可以T.作的。你已经看到了如何使用Python3 -i做测试；只是写一些代码并且在交互式解释器里测试一下。如果能工作，继续写。否则，修复它！

现在我们看一下真实房地产的例子。这个被证明是使用多重继承非常有效的例子。不

过，我不得不承认，当我开始设计的时候，我也有我的怀疑。看一看最初的问题并且看看你能否想出另一个方案，只用单一继承就能解决这个问题。一种设计完全不使用继承会怎样呢？你认为这3个当中哪一个是最优雅的解决方案？优雅是Python开发的主要目标，但是每一个程序员对于什么是最优雅的解决方案会有不同的看法。有些人倾向于使用组合认识和理解问题是最清晰的，而另一些人则发现多重继承是最有用的工具。

最后，尝试给这3个设计添加新的特性。任何你喜欢的特性都可以。我希望看到一种可以区分可用以及不可用属性的方法，对于新人来讲，如果它已经租出去了，对我就没多大用处了！

哪一个设计最易于扩展？哪一个最难？如果有人问你你为什么这么想，你能够自己解释吗？

总结

我们已经从面向对象程序员工具箱里最有用的工具之一简单继承，走到了最复杂之一的多重继承。我们学会了如何：

•给现有的类添加新的功能并且内置使用继承。

•通过抽象到父类来让多个类共享相似的代码。

•把多个相关联的功能通过多重继承联合起来。

.使用**super**方法调用父类的方法。

•在多重继承里格式化参数列表，这样超类不至于阻塞。

下一章，我们将讨论处理异常情况的微妙艺术。

4

第4章 异常处理

程序是非常脆弱的。如果代码总能够返回一个有效的结果，这将非常棒，但是有时候并不能计算出一个有效的结果。比如零不能做除数，或者在一个只有5个元素的列表里去访问第8个元素。

在以前，严格地检查每一个函数的输人来确保它们有意义是唯一解决这个问题的方法。通常函数会有一个特殊的返回值来表示一个错误情况；比如，它们可以通过返回一个负数来表示无法计算一个正数的值。不同的返回值可能代表发生了不同的错误。任何调用这个函数的代码可以显式地检查错误条件并且采取相应的行为。大部分代码并没有费心思去做这些，所以程序很容易崩溃。

在面向对象的世界里不是这样的！本章我们将学习异常，一些特别的错误对象，这些对象只有当处理它们有意义的时候才去处理。特别是我们将介绍的这些：

- 如何导致异常发生。
- 当异常发生时如何恢复。
- 如何用不同的代码处理不同的异常。
- 异常发生时的清理操作。
- 创建新的异常。
- 使用异常语法做流控。

抛出异常

那么到底什么是异常呢？从技术上讲，一个异常只是一个对象。有很多不同的异常类

可用，并且我们可以很容易地定义更多我们自己的异常类。这些异常类有一个共同点，就是它们都继承于一个叫**BaseException**的内置类。

当在程序流控里处理这些异常类时，这些异常对象会变得很特殊。肖一个异常发生时，所有应该发生的没有发生，除非当一个异常发生时它应该发生。讲得通吗？别担心，会的！

那么，我们如何才能产生异常呢？最简单的就是做一些傻事！事实上你已经这么做了并且看到了异常输出。比如，Python任何时候在你的程序里遇到一行它不能理解的代码，它就会产生一个**SyntaxError**的异常，这是一种异常类型，下面是一个常见的例子：

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

在Python 2和之前版本中，print语句是一个有效的命令，但是在Python 3中，由于print是一个函数，我们必须带上括号中的参数。所以如果在Python 3的解释器中输入上面的语句，我们会得到一个SyntaxError异常。

SyntaxError实际上是一个特殊的异常，因为我们不能处理它。它告诉我们，我们输人了错误的语句，并且最好弄清楚是什么。其他一些我们可以处理的常见异常，如下面的示例所示：

```
»> x = 5 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero

»> 1st = [1,2,3]
»> print (1st [3])
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
IndexError: list index out of range

»> 1st + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list not "int") to list
```

```
»> 1st. add
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
AttributeError:    * list' object has no attribute 'add'

»> d = { • a• : 'hello’ }
»> dfb']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'b'

»> print (this_is_not__a__var)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name *this—is—not_a— var■ is not defined
»›
```

有些时候这些异常指出了我们程序的一些错误（在这种情况下，我们可以去查看它指定的行数并且修复它），但是它们也会在一些合法的情况下发生。ZeroDivisionError并不总是意味着我们接收到了无效的输人，只是不同的输人。用户可以由于粗心输人一个零，或者故意的，或者它可能代表一个合法的数值，比如一个空的银行账号或者一个刚出生的孩子的年龄。

你可能已经注意到上面所述的所有内置异常名字都以Error结尾。在Python中，“错误”和“异常”经常可以互换使用。有时候大家会认为错误比异常更可怕，但是处理它们的方式完全相同。事实上，上面所有的错误类都以Exception（扩展BaseException）作为它们的超类。

抛出一个异常

现在，如果我们写一个程序，这个程序需要让用户或者一个调用函数，知道输人是无效的，我们该做什么？如果我们可以使用和Python相同的机制将会非常好……并且我们可以！想要看看如何做吗？这里有一个简单的类，它把元素添加到一个列表里，而且只有这个元素是有限的整数的时候才能这么做：

```
class EvenOnly(list):
    def append(self, integer):
```

```
        if not isinstance(integer, int):
            raise TypeError("Only integers can be added")
        if integer % 2:
            raise ValueError("Only even numbers can be added")
        super().append(integer)
```

就像我们在第2章讨论的，这个类扩展了内置的list，并且重写了 append方法，这个方法通过核查两个条件来确保这个元素是一个偶数，我们首先检查输人是不是一个int类型的实例，然后使用模运算符来确保它可以被2整除。如果这两个条件都不满足，raise关键字用来抛出一个异常。简单地在raise关键字后面跟上要被作为异常抛出的对象即可。在上面的例子中，两个对象是刚刚从内置的TypeError和ValueError类构造出来的。抛出的对象可以简单地是我们A己创建的一个新异常类的实例（我们很快就会看到），可以是在其他地方定义的异常，或者甚至一个我们刚才抛出和处理的异常对象。

如果我们在Python解释器中测试这个类，我们可以看到，当异常发生时，它会输出有用的错误信息，就像之前：

```
»> e = EvenOnlyO
»> e. append ("a string")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "even_integers.py", line 7, in add
    raise TypeError("Only integers can be added")
TypeError: Only integers can be added

»> e.append(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "even_integers.py", line 9, in add
    raise ValueError("Only even numbers can be added")
ValueError: Only even numbers can be added
»> e. append (2)
```

注意：虽然这个类对于演示异常是有效的，但是在实际工作时并不好它还是可能通过列表索引或者分片，让其他值进到列表中的可以通过重写其他合适的方法来避免这个问题，其中一些方法以双下画线开头

当一个异常产生时发生了什么

当一个异常被抛出的时候.它会立即停止程序的执行。在这个异常之后任何应该被执行的代码都没有执行，并且除非处理这个异常，否则程序会伴随着一条错误信息退出。看看这个简单的函数：

```
def no-return ():
    print ("I am about to raise an exception")
    raise Exception("This is always raised")
    print ("This line will never execute''}
    return "I won1 1 be returned"
```

如果我们执行这个函数，我们看到第1个print调用被执行了，然后异常发生了。第2个print语句永远不会执行，return语句也永远不会执行：

```
»> no_return ()
I am about to raise an exception
Traceback (most recent call last):
  File "<stdin›", line 1, in <module>
  File "exception_quits.py", line 3, in no -return
    raise Exception("This is always raised")
Exception: This is always raised
```

进一步说，如果我们有一个函数调用了另一个抛出异常的函数，那么在第i个函数调用第2个函数那个点后酣的所有代码都不会被执行。异常抛出会停止这个函数调用栈内所有代码的执行，直到这个异常被处理或者强制解释器退出。为了演示，我们添加第2个函数来调用我们的第1个函数：

```
def call_exceptor():
    print("call_exceptor starts here...")
    nonreturn()
    print("an exception was raised...")
    print     so these lines don't run**)
```

当我们调用这个函数的时候，我们看到第丨条print语句就像nonreturn函数中的第I行代码一样执行了。但是一旦异常抛出，其他任何代码都不会执行：

```
»> call-exceptor (>
call_exceptor starts here...
```

```
I am about to raise an exception
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "method -calls-excepting.py", line 9, in call- exceptor
     no-return(>
  File ''method-calls-excepting .py" , line 3, in no-return
     raise Exception("This is always raised")
Exception: This is always raised
```

我们很快就会看到解释器其实并没有采用一个快捷方式立即退出；在任何一个方法里异常都会被处理。事实上，当异常最初抛出时，可以在任何层面上处理它们。如果我们从下往上看一下异常的输出（称之为错误追踪），我们看到两个方法都列了出来。在no_return方法里，异常最早被抛出。然后就在这时，我们看到在call_exceptor方法里，调用了那个讨厌的n_O_returri函数，由于调用这个函数，异常“冒出来了”。从那之后，它跑到上层主解释器，最后打印出了错误追踪信息。

异常处理

现在让我们看一下异常的尾巴。也就是如果我们遇到一个异常情况，我们的代码应该如何反应或者从异常中恢复？我们通过把任何可能引起I异常的代码（不管是异常代码本身，或者是调用有可能包含异常的函数或者方法）包在tryiexcept语句中来处理它们。最基本的语法如下：

```
    try:
        no_return()
    except:
       print (''I caught an exception")
  print ("executed after the exception*')
```

如果我们运行的这个简单脚本里使用了我们已经存在的n_O_-ret_U3:_n函数，并且我们知道这个函数总是会抛出一个异常，我们会得到这样的输出：

```
I am about to raise an exception

I caught an exception

executed after the exception
```

这个nonreturn函数很高兴地告诉我们它将会引发一个异常。但是我们愚弄了它并且捕捉到了这个异常。一旦捕捉到，我们可以自己清理它（在这种情况下，通过输出我们处理它的情况），然后继续执行我们的代码，并且没有干扰我们的函数运行。n○_return函数中的其余代码仍然没有执行，但是调用这个函数的代码得以恢复和继续执行。

注意try和except之间的缩进。try语句可以包含任何可能抛出异常的代码。然后except语句回到和try语句同样的缩进级别。任何处理异常的代码要在except语句后面缩进。然后正常的代码恢复到原来的缩进级别。

上面的代码有一个问题，那就是它将捕获任何类型的异常。如果我们写了一些能够引起TypeError和ZeroDivisionError的代码怎么办？我们可能希望捕捉ZeroDivisionError异常，而让TypeError传递到控制台。你能猜到这个语法吗？这里有一个相当愚蠢的函数做了这个功能：

```
def funny_division(anumber):
    try:
        return 100 / anumber
    except ZeroDivisionError:
        return "Silly wabbit, you can1t divide by zero!"
print(funny-division(0> >
print(funny_division(50.0))
print <funny_division("hello"))
```

使用print语句对这个函数测试显示了预期的行为：

```
Silly wabbit, you can't divide by zero!

Traceback (most recent call last):
  File ''catch-specific-exception .py" , line 9, in <module>
    print(funny - division("hello">)
  File "catch_specific-exception.py", line 3, in funny - division
    return 100 /anumber
TypeError: unsupported operand type(s> for /:  'int' and 'str'
```

输出的第1行显示，如果我们输人o,会被嘲笑。如果我们用有效的数字（注意它不是一个整数，但仍然是一个有效的除数）来阔用就会得到正确执行。如果我们输入一个字符串（你在想如何得到一个TypeError,不是吗？），它将会失败并返回一个异常。如果我们使用了一个空的except语句，没有指定一个ZeroDivisionError,当我们输人一

个字符串时，它将指责我们想除以零，这个行为并不适当。

使用相同的代码我们甚至可以捕捉两个或者两个以上不同的异常并且处理它们。这里有一个例子，抛出了 3个不同种类的异常。它通过同一个异常处理程序处理TypeError和ZeroDivisionError，但同时如果你输人数字13，它也会抛出一个ValueError异常。

```
def funny_division2(anumber):
    try:
        if anumber == 13:
            raise ValueError("13 is an unlucky number")
        return 100 / anumber
    except (ZeroDivisionError, TypeError):
        return "Enter a number other than zero'*

for val in (0, "hello", 50.0, 13):

    print ("Testing {}:'*. format (val) , end=" "}
    print(funny_division2 (val))
```

底部的for循环简单地循环了几个测试输人并且打印出结果。你是否想知道在打印语句里end参数是干什么的，它只是把默认打印结束换行变成了一个空格，这样下一行的输出就会紧跟着打印出来。下面是脚本运行的结果：

```
Testing 0: Enter a number other than zero
Testing hello: Enter a number otlier than zero
Testing 50.0: 2.0
Testing 13: Traceback (most recent call last):
  File "catch - multiple-exceptions.py", line 11, in <module>
    print(funny - division2(val>)
  File "catch - multiple-exceptions.py", line 4, in funny_division2
    raise ValueError("13 is an unlucky number")
ValueError: 13 is an unlucky number
```

数字0和字符串输入都被except语句捕捉并且打印出一条合适的错误消息。数字13的异常没有被捕获，因为它是一个ValueError异常，不包含在要处理的异常类型中。

这都是不错的，但是如果我们想捕获不同种类的异常并且对它们做出不同的操作呢？或者我们想对一个异常做一些操作，然后允许它继续吐给父函数，好像它从没有被发现一

样？我们不需要任何新的语法来处理这些情况。可以叠加except语句，并且只有一个会被执行。对于第2个问题，如果我们已经在一个异常处理中，一个不带任何参数的raise关键字，将会重新抛出最后一个异常，观察：

```
def funny_division3(anumber):
    try:
        if anumber == 13:
            raise ValueError("13 is an unlucky number")
        return 100 / anumber
    except ZeroDivisionError:
        return "Enter a number other than zero"
    except TypeError:
        return "Enter a numerical value"
    except ValueError:
        print("No, No, not 13!")
        raise raise
```

最后一行重新抛出了 ValueError异常，所以在输出No,No,not 13!之后，它会再抛出一个异常；我们仍然可以在控制台得到错误追踪。

如果我们把异常处理堆栈做成上面一样，即使有多个except语句满足条件，也只有第一个匹配的语句会执行。如何才能让多于一条语句匹配到呢？记住异常是一个对象，并且可以继承。在下一节我们将会看到，大多数异常扩展了 Exception类（本身来自BaseException类）。如果我们在捕捉TypeError之前捕捉Exception，那么只有Exception的处理代码会执行，因为TypeError是从Exception继承而来的。

当我们想特别处理一些异常，然后在一个更通用的情况下处理所有剩下的异常时，上面讲的可以派上用场。我们可以在捕捉任何特殊异常之后简单地捕捉Exception异常并且处理这种通用情况,,

有些时候，当我们捕捉到一个异常时，我们需要引用这个Exception对象本身。这通常发生在如果我们使用A定义参数定义自己的异常时，但是也可以和标准异常相关。大多数异常类在它们的构造阐数里都会接收一组参数，并且我们可能想在异常处理中访问这些属性。如果我们定义了自己的异常类，当捕获到这个异常时，我们甚至可以调用自定义的方法。可以使用as关键字语法来把捕获到的异常作为变量来访问：

```
try:
    raise ValueError("This is an argument")
```

```
except ValueError as e:
    print("The exception arguments were", e.args)
```

如果运行这段简单代码，它会打印出我们在初始化时传入ValueError的字符串参数。

在Python2.5和之前版本中，as关键字不会用于命名一个异常相反，会使用一个逗号，所以之前的例子会是像except ValueError, e:为了避免捕获多个种类异常时发生混清（如 except ValueError TypeError ），在 Python 3.0 中为此做出了改变，这个改变是从Python 2.6开始的。因此as关键字支持多数现在的版本，但是如果你正在使用一个老版本的解释器，记住这个语法的区别，因为它一定会抓住你.

我们已经看到了在处理异常上一些语法的变体，但是我们仍然不知道，不考虑是否有异常发生，代码都是如何执行的。我们di不能指定代码只有在异常不发生的时候才会执行。finally和else两个关键字可以填补这个空内。任何一个都不需要额外的参数。下面的例子随机地挑选一个异常并抛出它。然后运行了一些不是那么复杂的异常处理代码来说明新引进的语法：

```
import random
some_exceptions = [ValueError, TypeError, IndexError, None]

try :
    choice = random.choice(some_exceptions)
    print (''raising { } " . format (choice))
    if choice:
        raise choice("An error")
except ValueError:
    print("Caught a ValueError")
except TypeError:
    print("Caught a TypeError")
except Exception as e:
    print("Caught some other error: %s" %
(e.__class__.__name—))
else:
    print("This code called if there is no exception")
finally:
```

```
        print("This cleanup code is always called")
```

这个例子几乎展示了所有可能的异常处理情况，如果我们运行几次，根据random挑选的异常的不同，每次会得到不同的输出。下面是一些例子：

```
$ python finally -and -else.py
raising None
This code called if there is no exception
This cleanup code is always called

$ python finally -and—else.py
raising <class 'TypeError'>
Caught a TypeError
This cleanup code is always called

$ python finally_and -else.py
raising <class 'IndexError'>
Caught some other error: IndexError
This cleanup code is always called

$ python finally_and—else.py
raising <class 'ValueError'>
Caught a ValueError
This cleanup code is always called
```

注意不管什么情况，finally语句里的print语句是如何执行的，这都是非常有用的，比如，当代码结束运行，甚至异常发生的时候，我们需要清理一个打开的数据库连接，关闭一个打开的文件，或者通过网络发送一个关闭握手信息。这也能以有趣的方式应用，如果一个try •句里，我们从一个函数return时；finally仍然会在返回时执行。

同时注意没有异常发生时的输出，else和finally语句会执行。else语句看起来像多余的，显得这段代码应该只会在没有异常发生时执行，可以直接放到整个try...except代码块之后。然而，这段代码仍然会在没布异常发生和处理的时候执行ˆ不久，当我们讨论使用异常作为控制流时，我们会看到更多这方面的讲解。

except else和finally任何一个都可以在try代码块后面执行（虽然else本身是无效的）。如果你想包含它们中的不止一个，那么except语句必须放到第一，然后是else语句，finally语句在结尾。except语句的顺序通常是从最具体的到最通用的。

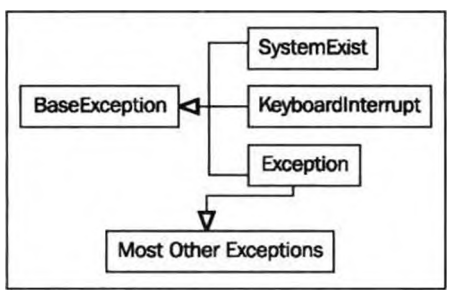
异常层级

我们已经遇到了许多最常见的内置异常类型，在你的H常Python开发过程中你将可能会遇到剩下的其他类型。正如我们上面所提到的，大部分异常都是Exception类的子类。但是对所有异常来说则是不对的。实际h Exception本身是从一个叫BaseException的类继承过来的（事实上，所有的异常必须扩展BaseException类或者它的一个子类)。存两个关键的异常 SystemExit 和 Keyboardlnterrupt,是盘接从 BaseException 继承过来的，而不是从Exception.、

不管程序是否正常退出，SystemExit异常都会被触发，通常是因为我们在代码的某个地方调用了 sys.exit函数（例如，用户选择了退出菜单项，在窗口上单击了关闭按钮，或者输人一个命令关闭服务器设计这个异常是为了能允许我们在程序最终退出之前清理代码，所以我们一般不需要显式地处理它（因为清理代码发生在finally语句里面，对吧？）。如果我们确实想处理它，我们通常会重新触发这个异常，因为抓住这个异常就会阻止程序退出当然，有些情况下我们可能想阻止程序退出，例如，如果有未被保存的改变并H.我们想提示用户，是否他们真的想退出程序。通常，如果我们处理SystemExit异常，那是因为我们想对它做一些特别的事情，或者是直接预测它。我们尤其不希望在一个要捕捉所有普通异常的通用汫句里不小心捕获到它。这就是为什么它直接继承于 BaseException

Keyboardlnterrupt异常通常用于命令行程序。当用户使用依赖于操作系统的纽合键（c/rZ+O显式地打断程序的执行时，这个异常会被抛出。针对用户要故意中断一个正在运行的程序，这是一个标准的方法，和SystemExit 一样,它应该几乎总是会终止程序..当然也和SystemExit 一样，它应该在finally代码块里做一些清理仟务！>

下面这个类图全面地展示了异常的层级：



我们使用except:语句而没有指定任何类型的异常，它将会捕捉所有

BaseException的子类，也就是说，它将会捕捉所有除了这两个“特殊”异常以外的其他异常。通常来说，我们不想捕捉它们，所以使用不带任何参数的except:语句是不明智的。如果你想捕捉除了我们刚刚讨论过的那两个异常以外的所有其他异常，那么你需要显式地捕捉Exception:就像我们之前讨论的一样.如果你试图去单独捕捉其他更具体的异常时，请确保这是最后一个except语句。

进一步说，如果你不想捕捉所有异常，我建议使用except BaseException:而不是一个原始的except:,这将明确地告诉你的代码的阅读者，你真正的意图是想处理上面两个特殊的异常。

定义自己的异常

通常，当我们想抛出一个异常，我们发现所有的内置异常没有一个能满足我们的需求。幸运的娃，定义一个我们自己的新的异常是非常简单的。类的名字通常被设计用于沟通出现了什么问题，而且我们可以在初始化函数中提供任意参数来添加额外的信息。

所有我们需要做的从Exception类继承。我们甚至不需要给这个类添加任何内容！当然，我们可以直接扩展BaseException类，但是这样它就不会被通用的except Exception语句捕捉到了。

闲话少说，下面是我们可能会在一个银行应用程序里用到的简单异常：

```
class InvalidWithdrawal(Exception):
    pass

raise InvalidWithdrawal("You don't have $50 in your account")
```

最后一行展示了如何抛出这个新定义的异常。我们可以给这个异常传递任意数M•的参数（通常是一个字符串消息，但是任何有用的对象都可以存储）。Excepti○n_init_方法被设计用于接收任何参数并且把它们作为一个元组存于一个叫args的属性里。这就使得定义异常变得更容易，因为不需要重写_init_方法。

当然，如果我们确实想定制初始化函数，也是允许的。这里有一个异常，它的初始化函数需要用户当前的余额以及用户想取出的数S。此外，它还增加了一个计算取款超出余额多少的方法：

```
class InvalidWithdrawal(Exception):
    def __init__(self, balance, amount):
        super().___init__("account doesn•t have ${}"•format<
```

```
            amount))
        self.amount = amount
        self.balance = balance

    def overage(self):
        return self.amount - self.balance

raise InvalidWithdrawal(25, 50)
```

最后的raise语句展示了如何构造异常。正如你看到的，我们可以对对象做的任何事情，都可以对异常来做。我们可以捕获一个异常并且作为一个T作对象传递给周围，尽管更常见的做法是包含这个工作对象的引用，作为这个异常的属性传递给周围。

下面我们看如果一个InvalidWithdrawal抛出了，该如何处理它：

```
try:
    raise InvalidWithdrawal(25, 50)
except InvalidWithdrawal as e:
    print("I*m sorry, but your withdrawal is "
          "more than your balance by "
          '*${}" .format (e .overage ()))
```

这里我们看到一个有效使用的as关键字。按照惯例，大部分Python程序员把异常命名为变量e，尽管如此，和往常一样，如果你春欢，你可以叫它ex、exception，或者aunt_sally〇

现在我们已经完全掌握广异常定义，包含初始化、属性和方法，并且当它被处理的时候，我们可以访问这个异常实例，在给异常传递信息上面，我们有绝对的权利。

定义我们自己的异常苻很多原因,:，给一个异常添加信息或者以某种方式写日志通常是有用的。但是，当目的是为了创建一个其他用户可以访问的框架、库，或者AP丨时，客户化的异常才真正变得有意义。在这种情况下，要小心确保你的代码所抛出的异常能让客户端程序员看得懂，容易处理，并且很清楚地描述了出了什么问题，这样他们才能修复它（如果是他们代码里的一处错误）或者处理它（如果是一个他们需要意识到的情况)。

异常不是例外

新手程序员趋向于认为异常唯一有用的情况就是作为“例外的情况”。然而，“例外情

况”的定义是模糊并且需要解释的。考虑下面两个函数：

```
def divide_with_exception(number, divisor):
    try:
        print (•'{} / {} = {}"• format (
            number, divisor, number / divisor * 1.0))
    except ZeroDivisionError:
        print("You can't divide by zero")

def divide_with_if(number, divisor):
    if divisor == 0:
        print("You can't divide by zero")
    else:
        print ("{} / {} = .format (
            number, divisor, number / divisor   1.0))
```

这两个函数的行为是一样的。如果**divisor**是零，会打印出一条错误信息，否则，除法的结果会打印并显示。显然，通过一条if语句的测试，我们就能避免一个**ZeroDivisionError**异常被抛出。同样地，我们可以通过显式地检金参数是否在列表内来避免**IndexError**异常，以及通过检查键值是否在一个字典内来避免**KeyError**异常。

但是我们不应该这么做。首先，我们可以写一条if语句来检查索引是否比列表参数要低，但是忘了检查负数值（记住，Python列表支持负数的索引，—1代表列表里的最后一个元素）。最终我们发现这个问题，并且发现必须找到所有我们检查代码的地方，去做修改。但是如果我们简单地捕捉**IndexError**异常并且处理它，我们的代码就可以工作了。

简而言之，对于例外的情况要使用异常，即使这些情况都只是一些很小的例外。

把这个观点反过来，我们可以看到，对于程序流控，异常语法也是有效的。就像一条if语句，异常可以用于决策制定、分支和消息传递。

想象一个为公司卖小部件和设备的库存应用程序。当客户购买商品时，这个商品可以是有库存的，这种情况商品会从库存中移除，然后返回剩下的商品数M，或者它可能已经缺货。现在，在一个库存应用程序中，缺货是一件非常正常的事情。这当然不是一个异常情况。但是如果缺货我们该返回什么呢？一个说“缺货”的字符串？一个负数？在这两种情况下，被调用的方法必须通过检查返回值是否是一个正数，或者其他东西来决定是否缺货。似乎有点混乱。相反，我们可以抛出一个**OutOfStockException**异常，并且使用**try**语句来引导程序的流控。有意义吗？

此外，我们要确保我们不能把相同的商品卖给两个不同的客户，或者卖给他们一件没有存货的商品。实现这个功能的一个方法就是锁定每一个商品的类型来确保每次只有一个人可以更新它。用户必须锁定这个商品，操作它（购买、添加商品、剩下商品的数量），然后解锁它。这里有一个非功能性的带着docstrings的Inventory对象，docstring描述了它的一些方法应该实现什么：

```
class Inventory:
    def lock(self, item_type):
        '*'Select the type of item that is going to
        be manipulated. This method will lock the
        item so nobody else can manipulate the
        inventory until it's returned. This prevents
        selling the same item to two different
        customers.'''
        pass

    def unlock(self, item_type):
        '*'Release the given type so that other
        customers can access it. '*'
        pass

    def purchase(self, item_type):
        '*'If the item is not locked, raise an
        exception. If the itemtype does not exist,
        raise an exception. If the item is currently
        out of stock, raise an exception. If the item
        Is available, subtract one item and return
        the number of items left.* *'
        pass
```

我们可以把这个对象原型交给一个开发者，让他们按照对象里说的做去实现这些方法，当需要购买商品的时候可以使用这些代码。我们将使用Python健壮的异常处理来考虑不同的分支，这些分支取决于我们所做的购买决策：

```
item_type = 'widget'
inv = Inventory()
inv.lock(item_type)
```

```
try:
    num-left = inv.purchase(item_type)
except InvalidltexnType:
    print("Sorry, we don•t sell {}"•format(item-type >
except OutOfStock:
    print ("Sorry, that item is out of stock. '*)
else:
    print("Purchase complete. There are "
          •'{} {}s left" . format (num__left, item_type))
finally:
    inv.unlock()
```

注意如何使用所有可能的异常处理语句来确保在正确的时间发生正确的行为。尽管OutOfStock异常不是一个很可怕的意外情况，我们可以适当地使用异常来处理它。也可以使用if...elif...else结构来编写，但是那样代码不容易维护或阅读，，

我们也可以使用异常来在不同的方法之间传递消息。比如，我们想通知客户预iH十么时间这款商品会再有存货，我们时以保证我们在构造OutOf Stock对象时，需要一个back_in-stock的参数。然后当我们处砰这个异常时，我们可以检查这个参数，然后提供给客户额外的信息，附加在对象上的信息可以很容易地在程序的两个不同部分传递。异常甚至可以提供一个方法来指导库存对象重新下单一个商品或者延期交货。

使用异常做程序流控可以用于一些方便的程序设计。这个讨论的重要性来自，异常并不是一个我们需要尽量避免的坏事。有一个异常发生并不是说，“你应该阻止这种情况的发生”。相反，它只不过是两部分可能直接相互调用的代码之间交互信息的有力方式。

案例学习

对于语法和定义，我们现在正在使用和处理的异常在细节上还处于相当低的水平。这个案例学习将帮助我们把之前章节的所有东西绑到一起，这样我们就能看到在更大的上下文的对象、继承以及模块中异常是如何使用的。

今天我们将设计一个简单的，集身份验证和授权一体的系统。整个系统将放到一个模块里，其他代码出于身份验证和授权的目的可以查洵这个模块对象.:，我们应该承认，从一开始，我们就不是安全专家，并且我们正在设计的系统可能充满了安全漏洞。然而，对于一个代码可以交互的基本的登录和权限系统，这将是足够的了。之后，如果其他代码需要

做得更安全，我们可以让一个安全专家或者密码学专家来审核或者修改我们的模块，而不用改变AIM。

身份验证是一个确保用户真实身份的过程。我们将效仿当今常见的使用用户名和私人秘钥组合的网络系统。其他的验证方法包括语音识别、指纹或者视网膜扫描仪，以及身份证。

另一方面，授权是关于决定是否一个给定的（身份验证过的）用户可以执行一个特定的动作。我们会创建一个基本的权限列表系统来存储一个特定的人允许做的每一个动作的列表。

此外，我们会添加一些管理功能，允许新用户添加到这个系统里。为了简便起见，一旦用户被创建，修改密码和权限的功能将被忽略，但是这些（非常必要的）功能一定可以在将来添加。

这里有一个简单的分析，现在让我们来进行设计。我们M然需要一个User类来存储用户名和加密后的密码。这个类也可以通过检查输人的密码是否有效来允许用户登录。我们可能不会需要一个叫Permission的类，因为这些只需要一个字典来存储字符串和用户列表的对应关系。我们应该需要一个集中的Authenticator类来处理用户管理以及登入/登出ˆ M后一片拼阁是一个Authorizes类，用于处理权限和检查用户是否可以执行一个动作。我们将会在auth模块里为这些类的每一个提供一个实例，这样其他模块就可以使用这个集中式机制来满足它们的身份验证和授权需求„当然，如果他们想要把这些类实例化成一些私有化实例，对于非集中的授权行为，它们也是可以这么做的。

随着进行,我们也会定义一些异常。我们将从一个特殊的AuthException基类开始，
这个类以username和可选的user对象作为参数;我们大多数自定义的异常都会继承它。

首先让我们来构建User类，它看起来似乎很简单$_3$可以使用用户名和密码来实例化一个新用户。为了减少被盗的可能性，存储的密码会被加密我们的目的是研究异常，而不是确保一个系统安全。我们已经警告过你了！我们还需要一个叫check_password的方法来测试是否提供的密码是正确的。下面是这个完整的类：

```
import hashlib

class User:
    def —init_(self, username, password):
        '''Create a new user object. The password
        will be encrypted before storing.* [1]'
        self.username = username
```

```
            self.password = self._encrypt_pw(password)
            self.is_logged_in = False

        def _encrypt_pw(self, password):
            '1 * Encrypt the password with the username and return
            the sha digest.*'*
            hash_string = (self.username + password)
            hash_string = hashestring. encode ("utf 8'*)
            return hashlib.sha256(hash_string).hexdigest()

        def check_password(self, password):
            1''Return True if the password is valid for this
            user, false otherwise.*''
            encrypted = self•—encrypt—pw(password)
            return encrypted == self.password
```

因为在_;1以<:_和che$_{c}$k_p$_{ass}$w$_{O}$rd两个方法里都需要有加密密码的代码，我们就把这部分代码拿出来形成了一个自d的方法。这样，如果有人意到它不安全需要改进，只需要在一个地方改就行这个类很容易扩展，让其包含像名字、联系信息和出生口期等强制或者可选的个人信息。

在我们写添加用户的代码之前（在尚未定义的Authenticator类中），我们应该研究一些用例。如果一切顺利，我们可以添加一个带有用户名和密码的用户；创建$_{User}$对象并把它插入到一个字典里。但是什么方式才能不正常呢？嗯，很明显，如果用户名已经在字典里存在了，那么我们就+想添加这个用户。否则，我们会把一个存在的用户数据覆盖掉，并a这个新的用户会有之前用户的权限，所以我们需要一个UsernameAlreadyExists异常。同样，出于安全考虑，如果密码太短，我们应该触发一个异常。这两种异常都会是我们之前提到的AuthException的扩展。所以，在写Authenticator之前，让我们先定义这3个异常类。

```
class AuthException(Exception):
    def __init__(self, username, user=None):
        super().—init—(username, user)
        self.username = username
        self.user = user
class UsernameAlreadyExists(AuthException):
    pass
```

```
class PasswordTooShort(AuthException):
    pass
```

AuthException需要一个用户名以及一个可选的用户参数。这第2个参数应该是带有用户名的User类的一个实例。我们定义的两个特定异常只需要在异常情况下通知调用类，所以我们不需要给它们添加额外的方法。

现在，让我们开始写Authenticator类。它可以简单地是一个用户名到用户对象的映射，所以我们在初始化函数里将从一个字典开始。添加新用户的方法，需要在创建一个新的User实例并把它添加到字典之前检查两个条件（密码长度和之前存在的用户）：

```
class Authenticator:
    def __init__(self):
        '''Construct an authenticator to manage
        users logging in and out.'[1]'
        self.users = {}

    def add_user(self, username, password):
        if username in self.users:
            raise UsernameAlreadyExists(username)
        if len(password) < 6:
            raise PasswordTooShort(username)
        self.users[username] = User(username, password)
```

当然，如果我们想的话，也可以扩展那个触发异常的密码验证方式，这个方式在其他一些情况很容易破解。

现在我们准备login方法。如果我们现在不考虑异常，基于登录是否成功，我们可能只需要这个方法能返回True或者False就行。但是我们现在考虑的就是异常，对于一个不是那么意外的情况，这正是使用异常的好地方。我们可以触发不同的异常，比如，如果用户名不存在或者密码错误。通过使用try/except/else语句，可以允许任何人优雅地处理尝试登录一个用户的情况。所以，首先我们先添加这些新的异常：

```
class InvalidUsername(AuthException):
    pass

class InvalidPassword(AuthException):
    pass
```

然后我们可以在Authenticator类里定义一个简单的login方法，这个方法会在必要的时候触发异常。如果没有异常，会标识user已经登录并返回：

```
    def login(self, username, password):
        try:
            user = self.users[username]
        except KeyError:
            raise InvalidUsername(username)

        if not user.check_password(password):
            raise InvalidPassword(username, user)

        user.is_logged_in = True
        return True
```

请注意如何处理KeyError。这个操作可以通过使用if username not in self.users:替代，但是我们选择了直接处理异常。我们结束并处理掉了第一个KeyError异常并且抛出了一个全新的我们自己定义的异常，这个异常更符合用户面向的API。

我们也可以添加一个方法来检查某个特定的用户名是否登录了。决定在这里是否使用异常是比较棘手的，如果用户名不存在我们应该抛出一个异常吗？如果用户没有登录我们应该抛出一个异常吗？

为了回答这些问题，我们需要考虑如何访问这个方法。大多数情况下，这个方法用来回答yes/no的问题，“我应该允许它们访问<某个东西>吗？”答案将可能是，“是的，用户名是有效的，并且已经登录了”，或者“不，用户名无效并且没有登录”。因此，一个布尔塑的返回值就足够了。如果只是为了使用，这里不需要使用异常。

```
    def is_logged_in(self, username):
        if username in self.users:
            return self.users [username] . is_logged_in
        return False
```

最后，我们可以把一个默认的认证器实例添加到我们的模块里，这样客户端代码通过使用 auth. authenticator 就nj"以访问它了：

```
authenticator = Authenticator()
```

这是在模块级别，任何类定义之外，所以认证器的变fi可以用auth. authenticator来访问。现在我们可以开始写Authorizes类了，这个类是把权限映射到用户。Authorizes不允许一个没有登录的用户访问一个授权权限，所以它们将需要一个特别身份验证的引用。我们也需要在初始化函数里初始化权限字典：

```
class Authorizor:
    def __init__(self, authenticator):
        self.authenticator = authenticator
        self . permissions == { }
```

现在我们可以写一个方法来添加新的权限并且建立用户和权限的关联。

```
    def add_permission (self, perm_name):
        1 *'Create a new permission that users
        can be added to'''
        try:
            perni-Set = self, permissions [perm 一 name]
        except KeyError:
            self .permissions [perzn_name] = set()
        else:
            raise PermissionError("Permission Exists")

    def permit   user(self, perm—name, username):
        1 *'Grant the given permission to the user'''
        try:
            perm_set = self.permissions[perm_name]
        except KeyError:
            raise PermissionError("Permission does not exist")
        else:
            if username not in self.authenticator.users:
                raise InvalidUsername(username)
            perm—set•add(username)
```

第1个方法允许我们创建一个新的权限，除非这个权限已经存在了，在这种情况下，会抛出一个异常。第2个方法允许我们给一个用户名添加一个权限，除非这个用户名或者这个权限不存在。

对于用户名我们使用了 set而没有使用list，因为这样即使你多次赋予一个用户权

限，集合的本性意味着这个用户只会在这个集合出现一次。集合像列表一样是序列，但是和列表不完全一样，集合是无序的，并且存储唯一值。不管我们往一个集合添加值多少次，这个值只会在这个集合里存储一次。

这两个方法里都会抛出一个PermissionError异常。这个新错误不需要用户名，所以我们直接让这个异常扩展Exception而不是我们客户化的AuthException:

```
class PermissionError(Exception):
    pass
```

最后，我们添加一个方法来检查某个用户是否具有特定的permission。为了赋予它们访问的权限，它们必须登录到认证器，并且要在一组具有访问权限的用户集合里。如果这两个条件都不满足，就会抛出一个异常：

```
    def check_permission(self, perm_name, username):
        if not self.authenticator.is_logged_in(username):
            raise NotLoggedInError(username)
        try:
            perm_set = self.permissions[perm_name]
        except KeyError:
            raise PermissionError("Permission does not exist")
        else:
            if username not in perm_set:
                raise NotPermittedError(username)
            else:
                return True
```

这里有两个新的异常；它们都需要用户名参数，所以我们把它们定义成AuthException 白勺子类：

```
class NotLoggedInError(AuthException):
    pass

class NotPermittedError(AuthException):
    pass
```

最后，我们可以以我们默认的身份来实例化一个“默认”的authorizes:

```
Authorizor = Authorizor(authenticator)
```

现在完成了一个基本的，但是完整的认证/授权系统。我们可以在Python提示符下测试这个系统，检查一个叫Joe的用户是否允许做油漆部门的任务：

```
»> in^>ort auth
»> auth.authenticator.add_user("joe", "joepassword")
»> auth. authorizor. add_jperxaission ("paint*')
»> auth.authorizor•check_permission("paint", "joe")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "auth.py", line 109, in check_permis3ion
    raise notloggedinerror(username)
auth.notloggedinerror: joe
»> auth.authenticator.is-logged-in(’• joe-)
False
»> auth.authenticator.login("joe", "joepassword")
True
»> auth. authorizor. check__permission ("paint" , "joe")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "auth.py", line 116, in check_j>ermission
    raise notpermittederror(username)
auth.notpermittederror: joe
»> auth.authorizor.check_permission("mix", "joe"
Traceback (most recent call last):
  File "auth.py", line 111, in check_permission
    perm_set = self.permissions[perm-    name]
Keyerror: 'mix

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "auth.pyH, line 113, in check^permission
    raise permisslonerror("permission does not exist")
auth.permissionerror: permission does not exist
»> auth. authorizor • permit-user ("mix" , "joe")
Traceback (most recent call last):
```

```
  File "auth.py", line 99, in permit - user
    perm—set = self.permissions[perm -name]
Keyerror: 'mix'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>"  line 1, in <module>
  File "auth.py", line 101, in permit   user
    raise permissionerror("permission does not exist")
auth.pennisslonerror: permission does not exist
»> auth. authorizor. permit user ("paint" , "joe")
»> auth.authorizor.check_permission("paint", "joe")
True
```

上面的输出故示了我们的所有代码以及绝大多数异常，但是想要真正理解我们所定义的AP丨，我们应该写一些异常处理代码并且真正去使用它。这里有一个基本的用户菜单接口，运行特定的用户去改变或者测试程序：

```
import auth

#创建一个测试用户并设置权限
auth.authenticator.add - user("joe", "joepassword")
auth.authorizor.add_permission("test program")
auth.authorizor.add_jpermission("change program")
auth. authorizor .permit—user (.'test program'',"joe")

class Editor:
    def -init-(self):
        self.username = None
        self.menu_map = {
                "login1*: self. login,
                "test": self.test,
                "change1*: self. change,
                "quit": self.quit
```

```
def login(self):
    logged—in = False
    while not logged_in:
        username = input("username:">
        password = input("password:")
        try:
            logged_in = auth.authenticator.login(
                   username, password)
            except auth • I rival idUser name :
                print("Sorry, that username does not exist")
            except auth.InvalidPassword:
               print<"Sorry, incorrect password")
            else:
                self.username = username

def is—permitted(self, permission):
    try:
        auth•authorizor•check—permission(
            permission, self.username)
    except auth.NotLoggedlnError as e:
        print("{} is not logged in".format(e.username))
        return False
    except auth.NotPermittedError as e:
        print (•'{} cannot {} M . format (
            e.username, permission))
        return False
     else:
         return True

def test(self):
    if self.is_permitted("test program"):
        print("Testing program now...")

def change(self):
    if self.is_permitted("change program"):
       print("Changing program now...")
```

```
        def quit(self):
            raise SystemExit()


        def menu(self):
            try:
                answer =
                while True:
                    print ("•, "
    Please enter a command:
    \tlogin\tLogin
    \ttest\tTest the program
    \tchange\tChange the program
    \tquit\tQuit
    """)
                    answer = input("enter a command: ").lower()
                    try:
                        func = self.menu-map[answer]
                    except KeyError:
                        print("{} is not a valid option".format(
                            answer))
                    else:
                        func()
            finally:
                print("Thank you for testing the auth module")


    Editor().menu()
```

这个相肖长的例子实际上非常简单。is_permitted方法可能是最有趣的；它是主要被test和change调用的内部方法，用来保证用户在继续做其他事情之前是允许访问的。当然，这两个方法是不完整的，但是在此我们不打算写一个编辑器，我们在通过测试一个身份验证和授权系统来说明使用异常和异常处理！

练习

如果你以前从没有处理过异常，你需要做的第一件事情就是去看一看你写过的Python代码，并且注意是否有些地方你应该做异常处理。你要如何处理它们？你需要全部处理它

们吗？有时候，让异常输出到控制台是最好的办法；有时候，你可以从错误中恢复并且允许程序继续执行；有时候，你只需要把错误重新格式化成用户可以理解的形式并且战示给他们。

比较常见的地方有文件丨/〇(有没有可能你的代码会去试图读一个不存在的文件？）、数学表达式（有没苻可能你在用零除一个数？）、列表索引（列表为空吗？）以及字典（键值是否存在？）。问问A己你是否该忽略这个问题，处理它的时候先检查这些值，或者采用异常处理。为了确保止 :确的代码在所有情况下都能执行，请注意那些你可能会使用finally 不″0 else 白勺也t〇

现在去写一些新的代码。想一想一个需要身份验证和授权的程序，并R试着去写一些代码，在里面使用我们在案例学习中构建的auth模块。如果它不够灵活，你可以随意修改它。试着以明智的方式去处理所有异常。如果对于一些需要身份验证的事情你觉得凼难，尝试把身份验证加到第2章里的笔记本的例子里。或者把授权添加到auth模块里——如果任何人都||丨以添加权限，这就不是一个非常冇用的模块!在允许添加或者修改权限之前，或许需要一个管理员的用户名和密码。

最后，试着去想一想你的代码里会抛出异常的地方。它可能会在你正在编写或者正在使用的代码中；或者写一个新项目作为练习。你可能会有很好的运气去设计一个供他人使用的小架构或者API;异常是你的代码和其他人代码之间非常好的沟通工具。作为API的一部分，请记住设计并且义拌化任何一个自己触发的异常，否则别人不知道如何或者是否该去处理它们!

总结

在这一章，我们学七了抛出、处理、定义以及操作异常的细节。在不需要调用函数显式地检查返回值的情况下，异常是一个非常强大的处理例外情况或者错误条件的方式。具体来说，我们讲了：

- 内置异常以及抛出异常。
- 一些处理特殊异常的方式。
- 定义新的异常。
- 在普通情况下使用异常。

下一章，在我们讨论阎向对象编程的原则和结构时，迄今为止我们学到的所有东西都应当会应用到Python应用程序当中。

5

第5章何时使用面向对象编程

在前面的章节里，我们已经介绍了多个面向对象编程的定义性特征。现在，我们知道了面向对象设计的原则和范例，以及在Python中面向对象编程的基本语法。

但在真正的应用开发中，应该在什么时候以怎样的形式应用这些原则和方法，仍然是闲扰我们的难题。在本章中，我们将会在更加有用的应用中对已经学过的知识进行进一步的讨论。在第7章，我们将介绍同样重要的内容，在什么时候不应该应用这些技术。通过本章你会学到：

- 怎样识别一个对象。
- 再论“数据”和“行为”。
- 通过property把数据包装在行为中。
- 通过行为限制数据的访问。
- DRY（ Don’t Repeat Yourself, 不要让自己重复）原则。
- 识别相同的代码。

把对象当作“对象”来对待

把对象当作“对象”来对待，这看起来似乎是显而易见的。在代码中，你通常会使用一个特殊的类，在每个问题域中产生单独的对象。在前面的章节中我们已经见过了很多这样的例子，一般来说，这个过程就是先确定问题中的对象，之后用代码模拟它们的数据和行为。

确定对象是面向对象分析与编程中最为重要的任务，但它并不像我们在一篇文章的短

段落中数名词的数量那样简单。请i己住，对象同时包含“数据”和“行为”。如果只对数据操作，使用列表、集合、字典或是Python中其他的数据结构（我们下章将会讲到）通常是更好的选择。如果只关注行为而不存储任何数据，那一个简单的函数则更为合适。

然而，一个对象是同时含有数据和行为的。对大部分Python程序员来说，除非（或是直到）遇到了一个特别明显的需求而需要定义一个类，否则都会直接使用Python内置的数据结构。这当然是件好事，我们没有理由在对组织代码没有任何帮助的情况下，只是为了增加一层额外的抽象就采用面向对象编程。尽管有时候，那些看似显而易见的需求其实并不那么明显。

Python程序员通常是从对几个变量中的数据进行排序开始学起的。随着排序程序一步步地扩展，我们会发现其实我们是在把同一组相关的变M传到不同的函数中去，这正是一个适合将变量和函数组合成类的好场景。假设我们需要设计一个程序在二维空间中对多边形建模，通常来说我们会以一个存储着各个顶点的列表来代表这个多边形。每个顶点以一个二维元组(x, y)建模，该元组描述顶点的位置坐标。这就是为多边形逮模需要的全部数据，我们将它们存储在一个两层嵌套的数据结构中（具体地讲就是一个元组的列表）：

```
square = [(1,1),        (1,2),    (2,2),    (2,1)]
```

现在，我们想要计算多边形的周长。很简单，我们只需要将相邻两点之间的距离相加即可。为了这样做，我们需要的就是一个能够计算两点间距离的函数。下面我们给出这两个函数：

```
import math

def distance(pi, p2):
      return math.sqrt((pi[0]-p2[0])**2 + (pi[1]-p2[1]}**2)

def perimeter(polygon):
      perimeter = 0
      points = polygon + [polygon[0]]
      for i in range(len(polygon)):
            perimeter += distance(points[i], points[i+1])
      return perimeter
```

作为一个掌握了面向对象技术的程序员，我们应该能清楚地意识到，这个时候如果有一个多边形类，就能够封装起点的列表（数据）和perimeter函数（行为)。此外，还有

一个如我们在第2章定义的点类，用来封装x、y坐标和distance方法。但是我们是不是真的需要这样做呢？

对于上面的代码，算对也不算对。我们已经学过了面向对象设计的原则，所以我们能很快地写出面向对象版本的代码：

```
import math

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, p2):
        return math.sqrt((self.x-p2.x)**2 + (self.y-p2.y)**2)

class Polygon:
    def __init__(self):
        self.vertices =[]

    def add_point(self, point):
        self.vertices.append((point))

    def perimeter(self):
        perimeter = 0
        points = self.vertices + [self.vertices[0]]
        for i in range(len(self.vertices)):
            perimeter += points[i].distance(points[i+1])
        return perimeter
```

我们看到，在高亮的部分，虽然add_point方法并不是严格必需的，但这部分代码也已经比之前的版本长了一倍还多。

现在，为了对两个版本代码的差异理解得更深入一些，让我们对两个版本使用的API做个比较。首先来看计算多边形周长的面向对象版本：

```
>>> square = Polygon ()
>>> square. add_point (Point (1,1))
>>> square . add_point (Point (1,2))
```

```
»> square.add__point (Point (2,2))
»> square . addjpoint (Point (2,1))
»> square. perimeter ()
```

是不是非常通俗易懂，让我们来和函数版本的代码做比较：

```
»> square = [(1,1),     (1,2),  (2,2),  (2,1)]
»> perimeter (square)
```

是的，面向对象版本的AP丨原来并不那么紧凑！但在另一方面，我认为面向对象版本的代码比起函数版本的代码要易读得多。在函数版本中，我们怎么才能知道一个元组的列表代表了什么？怎样才能记住这是哪种类型的对象（一个二维元组列表？太不直观了！），我们要把它传递给计算周长的函数？为了回答这些问题，我们不得不需要大量的外部文档才能解释函数的作用。

相比较而言，面向对象版本的代码自身就如同文档。我们只需要看一下方法列表和它们需要的参数，就可以知道这个对象的作用是什么，以及如何去调用它。当我们为函数版本的代码写好了全部文档时，很可能这已经比面向对象版本的代码还要长得多了。

此外，代码长度作为一个极其可怕的用来衡量代码复杂度的指标，一些程序员（庆幸的是他们大多数不是Python程序员）总是对长代码唯恐避之不及。他们在一行代码上费尽心思，这些“一行程序员”写出的代码，到第二天甚至连作者自己都读不懂。所以，相对于我们的代码是不是更短，我们应该关注它是不是更加地易读易用才对。

这里有个小练习，需要你停下来思考一下：你能想出一种办法使得面向对象的Polygon类像函数那样易于使用吗？

好的，所有我们要做的只是将Polygon类中的API变换一下，使它构造时能够接收多个点作为参数。让我们使它的初始化函数可以接收一个Point对象的列表。事实上，如果需要的话我们也可以只接收元组，然后自行把这些元组构造成Point对象：

```
def _init_____(self, points =[]):
    self.vertices =[]
        for point in points:
        if isinstance(point, tuple):
            point = Point(*point)
        self.vertices.append(point)
```

在这个例子中，通过简单地遍历列表，可以使列表中的元组转换为点。在这里我们假设，如果列表中的某个对象并不是一个元组，我们不做额外的处理，而是认为它已经是个Point对象，或者是一个未知的可以像Point对象一样1：作的鸭子类型对象。

正如我们看到的，识别一个对象是否应该被表示为自定义类并不总是那么容易。如果我们有个新函数，它可以接收多边形作为参数，比如area(polygO$_n$>或者point_in_polygon（polygon，x，y>，这时，面向对象版本代码的优势就体现出来_广$_0$同样地.假如我们需要为多边形添加一些其他的属性，例如color或者texture,将数据封装进类也会变得更有意义。

全部的区别取决于怎样设计。一般来说，越是复杂的数据集，越容易需要对这些数据进行特定操作的函数，这时使用具有属性和方法的类也就越有价值。

当我们决定采用这种设计方法时，还应该考虑类将会被怎样使用。如果我们只是在一个大型问题中试图计算某一个多边形的周长，一个函数可能是一次性处理中最快的开发方法，也是最简单的使用方式。但另一方面，如果我们的程序需要对多边形进行多种处理（计算周长、面积和其他多边形的交点等），可以肯定我们需要一个多功能的对象来满足这些需求。

还要额外注意的是，对象之间的交互。首先来看继承关系，继承关系在没有类的情况下不可能被简洁地模拟出来，所以一定要使用类来表现继承关系。再看其他几种我们在第1章就讨论过的关系：关联和组合。从技术上讲，组合关系仅使用一般的数据结构就能够被模拟。例如，我们可以使用一个包含元素是字典的列表来存储元组元索，但是通常创建一个对象会降低复杂程度，尤其是当存在与数据相关的行为时。

> 不要因为能够使用一个对象就急于使用这个对象，也绝不要在当你需要使用一个类的时候，疏于创建一个类。

使用property为类中的数据添加行为

本书通篇始终在关注行为和数据的分离，对于面向对象编程来说，这是非常重要的。但是，在Python中我们看到，这种区分有时候也是模糊的。Python这门语言非常善于对区别做模糊处理.它并不帮助我们打破思维的界限$_3$准确地说，Python想要告诉我们，其实我们的大脑才是界限，“思维本没有界限”。

在深入细节之前，先让我们讨论一些“坏”的面向对象理论。很多面向对象的编程语言（Java罪大恶极）都告诉我们绝不要直接访问属性。它们教导我们像下面这样去访问一个类中的属性：

```
class Color:
    def ____init____(self, rgb_value, name):
        self.rgb_value = rgb_value
        self.name = name

    def set_name(self, name):
        self._name = name

    def get_name(self):
        return self.__name
```

前缀有一个单下画线的变量表明它们是类所私有的（在其他语言中，私有变量实际上将被强制为只有对象本身可以访问），接着get和set方法提供了对每个变量的访问方式。这个类在实际使用中一般采用如下的方式：

```
>>> c = Color ("#ff0000" , "bright red")
>>> c.get_name ()
'bright red'
>>> c.set_name ("red")
>>> c.get_name ()
'red'
```

这并不像Python喜欢的直接访问方式那样具备可读性：

```
class Color:
    def __init__(self, rgb_value, name):
        self.rgb_value = rgb_value
        self.name = name

c = Color("#ff0000", "bright red")
print(c.name)
c.name = "red"
```

但为什么会有人推荐这种以方法为基础的语法呢？他们的理由是这样的，也许某一天可能会需要在这些变M被赋值或取值时添加额外功能的代码。举个例子来说，我们决定需要缓存一个值以及返回这个缓存的值，或者我们想验证输入值是否是合理的。在代码中，我们可以用改变set_name（）方法的实现来达到目的：

```
def set_name(self, name):
    if not name:
        raise Exception ("Invalid Name'*)
    self._name = name
```

在Java及其类似的语言中，如果我们在最开始的代码中采用直接访问属性的方式，而之后改成像上面介绍的基于方法的访问方式，我们会发现一个问题：采用直接访问属性方式的代码，现在必须通过调用方法才能访问原有的属性，如果他们不改变自己的访问方式，那么代码就被破坏了。在这类语言中有一则箴言，那就是一定不要将公共成员私有化。然而这在Python中就不那么起作用了，因为Python本来就没有私有成员的概念。

事实上，Python在该问题的处理上要好得多。我们 卩]″以使用Python中的property关键字，使一个方法看起来就像一个类属性一样。假如我们原本使用直接成员访问的方式去访问厲性，之后我们可以增加几个方法，在不改变访问接口的情况下，来对name这个变量进行取值和赋值。让我们看一下怎么做：

```
class Color:
    def  init_(self, rgb_value, name):
        self.rgb_value = rgb_value
        self .—name = name

    def _set_name (self, name):
        if not name:
            raise Exception("Invalid Name")
        self._name = name

    def —get name (self):
        return self._name

    name = property(—get—name, _set_name)
```

假设我们以最初那个基于非方法访问方式的，可以直接对name属性进行赋值的类开

始，那么之后可以把代码改成上面这个样子。先将name这个属性改为一个（半）私有的_nam$_e$属性，接着我们添加两个（半）私有方法对这个变量进行取值和赋值，并在赋值时同时进行验证。

最后，我们在代码底部使用property关键字进行声明。现在，我们的Color类就像被施了魔法一般，拥有了一个全新的name属性，替换掉前一个版本中的name属性。现在这个新的name厲性变为了一个property属性，需要通过调用我们刚刚添加的两个方法才能访问或是改变其值t而新版本的Color类仍能以与前一版本中相同的方式来使用，当然它还能支持在对name赋值时进行验证$_3$

```
»> c = Color ("#0000ff", "bright red")
»> print (c. name)
bright red
»> c.name = "red″
»> print (c.name)
red
»> c.name = ,,u
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "setting -name_property.py", line 8, in _set—name
    raise Exception("Invalid Name")
Exception: Invalid Name
```

如果我们采用这种先编写访问name属性的方法，再将它们构造成一个property对象的方式，那么我们之前编写的任何代码仍然能够工作。当然除了我们在一开始就禁止的那种传递一个空的property值的行为。好的，我们成功了！

不过请记住，即便name变为了 property属性，也不能保证100%的安全。如果人们真的想使它设为空字符串值，仍然可以通过直接访问_$_{na}$me属性的方式来达到目的。但是，如果他们真的直接访问了我们显式地标明为私有的变量，那么由此产生的一切问题就应当由他们自己去负责，而不需要我们处理。

property是怎样工作的

那么，property对象究竟是怎样工作的呢？仔细想一想，property函数实际上返回了一个对象，该对象通过我们指定的方法代理了全部对属性值访问或赋值的请求。property关键字实际上就是构造了这样一个对象。

property构造函数实际上还可以接收两个额外的参数:一个删除函数和一个property的文本字符串。在实践中很少使用删除函数，但是如果需要记录所删除的值，那么删除函数还是很有用处的。同时在我们满足某个条件的情况下，删除函数还可以否决删除操作。文本字符串是一个用来描述该property的字符串，它和我们在第2章中介绍的文本字符串没有区别。如果我们不提供文本字符串这个参数，那么该值将从property的第一个参数，也就是gettei•方法的文本字符串中复制过来。

这里给出一个有点笨的例子，只是为了让你明白在什么时候是哪个方法被调用了：

```
class Silly:
    def _get_silly(self):
        print("You are getting silly")
        return self ,__silly
    def _set_silly(self, value):
        print("You are making silly {}n.format(value))
        self.__silly = value
    def _del_silly(self):
        print("Whoah, you killed silly!")
        del self._silly

    silly = property(- get -silly, -set-silly,
            - del-silly, "This is a silly property")
```

如果我们真的使用了这个类，那么它将会打印出我们请求它打印出的字符串：

```
»> s = Silly ()
»> s.silly = "funny"
You are making silly funny
»> s. silly
You are getting silly
 'funny'
»> del s . silly
Whoah, you killed silly!
```

此外，如果我们查看Silly类的帮助文档（通过在解释器提示符下输入help（silly)),它将会把silly属性的自定义文本字符串展示出来：

```
Help on class Silly in module _main_____________:
class Silly(builtins.object)
```

```
    Data descriptors defined here:

    __diet__
      dictionary for instance variables (if defined)

    __weakref__
        list of weak references to the object (if defined)

I silly
I       This is a silly property
```

一切都再一次地像我们计划的那样正常工作了。在实际应用中，property通常只定义前两个参数：getter函数和setter函数。文本字符串被定义为一般的getter函数的文本字符串，并复制到property，而删除函数则为空，因为对象属性极少需要被删除。如果程序员真的需要删除一个没有指定删除函数的对象属性，程序就会抛出一个异常。所以，假如真的有正当的理由要删除property，那还是应该提供一个删除函数。

装饰器：创建property的另一种方法

（如果你从未使用过Python中的装饰器，你可能需要先跳过这部分，等我们在第8章讨论完装饰器模式后再回过头来读这部分。）

装饰器在Python 2.4中被首次引入。作为一个动态修改函数的方法，装饰器将被装饰函数作为参数传入到另一个函数，从而得到一个新函数并将它返回。在这里我们不会对装饰器做太深入的讲解，但是它的基本语法却是非常容易掌握的。如果你之前从没用过装饰器，也依然可以跟着我们继续。

使用装饰器非常简单，我们只需要为函数名添加一个@符号作为前缀，并把结果放置在被装饰函数的定义之前就可以了。property函数本身也可以使用装饰器语法来使一个get函数变成property函数的参数：

```
class Foo:
    @property
    def foo(self):
        return "bar"
```

上面的用法使property成为了一个装饰器，这相当于应用了 foo = property(foo)。不过从可读性的角度来看，二者最主要的区别在于我们在方法的顶部

就标记foo函数为一个property函数，而不再是在foo函数定义之后，那样可能会使我们很容易地就忽视了它。

再进一步，我们可以为这个新的property函数指定一个setter•函数，如下：

```
class Foo:
    0property
    def foo(self):
        return self. foo

    @foo.setter
        def foo(self, value):
            self•—foo = value
```

这个语法看起来稍有些奇怪。首先我们装饰了 foo方法，使它成为getter。接着我们用刚装饰过的foo方法的setter属性又装饰了一个新方法，而这个新方法的名字和刚装饰过的foo方法竟然是一样的！请一定记住，property函数返回的是一个对象，这个对象被自动设置拥有一个setter属性，而这个setter属性可以被设置成为一个装饰器去装饰其他的函数。对get方法和set方法使用同样的名字并不是必需的，但是这确实可以帮助我们把多个方法组合起来成为一个property。

我们当然也可以通过@foo.deleter来指定一个删除函数。但是我们却不能应用property装饰器来指定文本字符串，因此，对于文本字符串我们仍要依赖property从初始的getter方法中复制它的文本字符串。

下面就是我们将Silly类用property作为装饰器重写的版本：

```
class Silly:
    ©property
    def silly(self):
        "This is a silly property"
        print("You are getting silly")
        return self._silly

    0silly.setter
    def silly(self, value):
        print ("You are making silly { } *'. format (value))
        self._silly = value
```

```
@silly.deleter
def silly(self):
    print("Whoah, you killed silly!")
    del self._silly
```

这个类与之前的版本用起来是完全一样的，包括它的帮助文档。对于这两种语法，你可以任意选择你觉得更容易阅读和更简洁的那种使用。

何时该使用property属性

使用property关键字使得行为和数据之间的分界点变得模糊了，我们在调用时有时会因此感到闲惑，到底该使用哪个。我们之前看到的就是一个使用property属性的最简单的例子：我们在一个类中存储了一些数据，之后想要对这些数据添加一些行为。当然.决定是否使用property属性也还有一些其他需要考虑的因素。

从技术上讲，Python中数据、property属性、方法都是类的属性。实际上，尽管方法可以被调用，但我们并不需要区分它和其他两种属性的区别。我们在第7章中甚至会看到，创建一个可调用的普通对象也是可以的，函数和方法本身也可以是普通的对象。

实际上，方法只是一个可调用的属性，property属性也只是一个能帮助我们进行决策的自定义属性。方法应该只代表那些由对象去完成或执行的动作。当你调用一个方法时，哪怕只有一个参数，也应该做些什么。因为方法相当于动词。

接着留给我们的就是决定到底是用标准的数据属性还是property属性。一般来讲，我们通常会使用标准的数据属性。除非需要某些访问控制时，我们才会用property属性。无论在哪种情况下，你的属性都应该是个名词。属性和property属性之间唯一的区别，就是当property属性被检索、赋值或者删除的时候，我们可以自动调用一些自定义的动作。

让我们来看一个更实际的例子。假如有一个定制化行为的普遍需求，它要求对那些难以计弊或是查找起来花费过大的值（例如一个网络请求或是数据库查询）进行缓存。我们的目的是在本地存储这个值以避免重复调用那些花费过大的计算。

我们可以通过在property属性中使用自定义的getter来达到这个目的[3]当该值第一次被检索的时候，我们执行杏找或计算。接着就可以将这个值以对象中的私有属性的形式缓存在本地（或者存于专用的缓存软件中）。之后，当再次请求这个值时，我们就可以返回存储的数据。下面给出一个如何缓存网页的例子：

```
from urllib.request import urlopen
class WebPage:
```

```
    def —init-(self, url):
        self.url = url
        self, content = None

    @property
    def content(self):
        if not self• content:
            print("Retrieving New Page...")
            self.^content = urlopen(self.url).read()
        return self• content
```

我们可以测试这段代码，看看是不是页面只是被检索了一次：

```
»> in5>ort time
»> webpage = WebPage("http://ccphillips.net/")
»> now = time . time ()
»> contentl = webpage. content
Retrieving New Page...
»> time . time () - now
22.43316888809204
»> now = time. time ()
»> content2 = webpage. content
»> time. time () - now
1.9266459941864014
»> content2 = contentl
True
```

在糟糕的卫星通信条件下，第1次加载网页内容花费了近20秒时间。但是第2次，我在2秒内就得到了结果（这真的只是将文本写人解释器的时间）。

自定义的getter对于需要依据对象中其他成员进行计算的属性，也是非常有帮助的。例如，我们想要计算一个整数列表中各元素的平均值：

```
class AverageList(list):
    @property
    def average(self):
        return sum(self) / len(self)
```

这是一个非常简单的类，它继承自list,所以我们能够轻易地获得类列表的行为。通

过在类中加人一个property属性，很快我们的列表就可以得到一个平均值属性：

```
»> a = AverageList([l,2,3,4 >
»> a.average
2.5
```

当然，我们也可以用方法来实现它，可由于方法代表的是一个动作，所以我们应该把它称作calculate_average ○ ，但一个称为average的property则更为合适。肖然这两种方式都是易写易读的。

就像我们已经看到的那样，自定义的setter对于验证很有帮助，但它同时也可以用于将一个值代理存到另一个位置。比如，我们可以在WebPage类中增加一个对内容值的setter，通过这个setter实现不管何时其值被设定，它都可以自动记录到我们的Web服务器上，同时上传这个新页面。

管理对象

我们已经关注过了对象以及对象的属性和方法。现在，我们来看看怎样设计更高级别的对象：一种用来管理其他对象的对象。这种对象可以将一切都绑在一起。

这种类型的对象与我们目前为止所见过的其他对象的区别，就在于我们在例子中见过的对象往往代表的是具体的想法，而管理对象更像是办公室中的经理，他们不做那些实际“可见”的T_作，但是如果没有他们，部门之间就没有了沟通，没人知道自己该做些什么。类似地，管理类中的属性通常引用的是那些做“可见”工作的对象；管理类的行为就是在恰当的时间将任务委托给其他类，同时传递它们之间的消息。

例如，我们要写一个可以在ZIP压缩文件里实现查找和替换操作的程序。首先我们需要用对象来代表这个Z丨P文件和每个独立的文本文件（幸运的是，我们不必写这些类，这些都已经在Python的标准库里了）。管理对象将会负责的，就是确保下面3个步骤能够按顺序发生：

- 解压缩文件。
- 执行查找和替换动作。
- 压缩这些新文件。

这个类初始化时以一个.zip的文件名和查找、替换字符串作为参数。我们创建一个临时的目录存储这些解压缩后的文件，而文件夹本身可以保持不变。我们还要为内部添加一个实用的助手方法，用来帮助确认目录中每个文件都有独立的文件名。

```
import sys
import os
import shutil
import zipfile

class ZipReplace:
    def __init__(self, filename, search_string,
                 replace_string):
        self.filename = filename
        self.search_string = search_string
        self.replace_string = replace_string
        self.temp_directory = "unzipped-format(
            filename)

    def _full_filename(self, filename):
        return os.path.join(self.temp_directory, filename)
```

然后我们为这3个步骤中的每一个在整体上创建一个“管理者”方法，这个方法的职责就是将工作委托给其他方法。很明显，我们不需要创建对象，而是只用一个方法，甚至是一个脚本就能实现这3个步骤。为什么要把3个步骤分开呢？下面是这样做的一些好处：

- 可读性：每一个步骤的代码是一个自身独立的单元，这样非常便于阅读和理解■:方法的名字可以描述这个方法的内容，甚至不需要额外的文档去帮助理解代码的作用。
- 可扩展性：如果一个子类想要使用压缩的TAR文件而不是ZIP文件，那么就可以直接覆盖zip和unzip方法，而不用再复制一遍find_replace方法。
- 隔离性：一个外部类可以创建这个类的一个实例，这样它可以直接在一些文件夹中调用查找和替换的方法，而不必zip这些内容。

执行委托功能的方法在下面这段代码的开头；为了完整，我们将其余的方法也都包括了进来：

```
    def zip_find_replace (self):
        self.unzip_files()
        self.find_replace()
        self.zip_files()

    def unzip_files (self):
        os.mkdir(self.temp_directory)
```

```
        zip = zipfile.ZipFile(self.filename)
        try:
            zip.extractall(self.temp_directory)
        finally:
            zip.close()

    def find_replace(self):
        for filename in os.listdir(self.temp_directory):
            with open(self._full_filename(filename)) as fil
                contents = file.read()
            contents = contents.replace(
                    self.search_string, self.replace_string
            with open(
                self._full_filename(filename), "w") as file
                file.write(contents)

    def zip_files(self):
        file = zipfile.ZipFile(self.filename, 'w')
        for filename in os.listdir(self.temp_directory):
            file.write(
                self._full_filename(filename), filename)
        shutil.rmtree(self.temp_directory)

    if __name__ == "__main__":
        ZipReplace(*sys.argv[1:4]).zip_find_replace()
```

为简便起见，压缩和解压缩文件的代码我们没有详细讲解。我们当下的关注点是面向对象设计；如果你对zipfile模块的内部细节感兴趣，可以登录http://docs.python.org/library/zipfile.html或者在交互式解释器中输入import zipfile; help(zipfile)来参考标准库中的相关文档。记住，这个例子只会搜索ZIP文件中的顶层文件，如果在解乐后的内容中存在文件夹，那么这些文件夹及其内部的任何文件都是不会被扫描到的。

代码的最后两行让我们可以在命令行中通过传递ZIP文件名、查找字符串、替换字符串作为参数运行这个例子：

```
python zipsearch.py hello.zip hello hi
```

当然，这个对象不是必须从命令行中创建；它也可以在另外的模块（如执行批处理ZIP文件的流程）中被导人，或者通过GU丨接口或是其他的对ZIP文件进行处理的更高级别的管理对象（例如从FTP服务器上获取ZIP文件，或是将ZIP文件备份到外部磁盘）访问。

随着程序越发复杂，对象被模拟得越来越不像一个物理上的对象。property是另一种抽象对象，而方法则是可以改变这些抽象对象状态的动作c■但是无论一个对象有多么复杂，对象的本质仍是具体的property和定义好的行为两者的集合。

移除重复的代码

像ZipReplace这种具备管理风格的类，它的代码是非常通用的，可以应用在很多不同情况下。无论是组合关系还是继承关系，它都可以帮助我们使代码仅保留一份，即消除重复的代码。在我们学习如何这样做之前，先让我们讨论一些小理论，尤其是为什么重复的代码是一件坏事。

有很多个理由解释这个问题，但这些理由最终都归结于可读性和可维护性上。当我们要写一段和之前的代码片段相似的新代码段时，最简单的方法就是复制旧的代码，然后换掉需要替换的部分（不同的名称、逻辑、注释），接着让它能够在新的位置上T作。此外，如果我们要写一段类似于以前写过的代码，但在项目中又没有完全相同的代码时，最简单的方法就是写一段有相似行为的新代码，而不会想着怎样去提取出那些重复的功能[3]

但只要有人需要阅读和理解这些代码，他们就会遇到重复的代码块，从而陷人两难的境地：本可以直接通晓其意的代码却不得不需要花时间去理解。这段代码和另一段代码有什么不同？这几段代码怎么会是一样的呢？在什么条件下调用的是这段代码？在什么条件下会去调用另一段？你也许会辩解说只有你是唯一一个会读这段代码的人，但是如果你8个月都没有碰过这部分代码，那么你也会像一个新人一样难以理解它。当我们试图阅读两段相似的代码时，我们必须找出它们哪里是不同的，理解为什么它们有这些不同，这浪费了阅读者大量的时间。编写代码首要考虑的应该是它的可读性。

⑤/ 我曾试图理解一个人写的代码，那份代码中有对同一段三百多行写得很差的代码3处完全相同的副本。直到我意识到这3段“完全相同的”代码，实际上是对有着细微差别的税种进行计算的时候，我已经花去了一个月的时间。这些细微的差别中，有一些是故意的，但还有一些区域很明显就是某个人对其中一处函数的计算进行了更新，但却没有更新另外两处。代码中，这种细微差别和无法理解的Bug的数量真是难以计数。

阅读这样重复的代码是令人生厌的，而对这些代码进行维护则是更大的折磨。前面的故事表明，同时保持两份相似代码同步可能会是一场噩梦。无论何时我们要更新其中一份代码，都必须记住要对两份全部更新，并且我们还得记住每个部分的不同到底是什么. 以便当我们再编辑它们的时候可以对更改的部分加以修正。如果我们忘了同时更新两份代码，最终就会产生极其恼人的Bug, 这些Bug通常的表现就是“我已经修好它了，为什么它还会出现呢？”

当人们阅读和维护这些代码时，结果和我们一开始就把代码写成非重复性的方式相比，要想读懂并测试这些代码，那花费的时间简直就是个天文数字。更令人沮丧的莫过于我们自己就是做维护的人，我们通过复制-粘贴已有代码省下的时间全都浪费在我们第一次去维护它上了。阅读和维护代码比写出这些代码发生的次数要多得多，易于理解的代码应该是最为重要的。

这就是为什么程序员，尤其是Python程序员（比其他语言的程序员把简洁的代码看得更为重要），会遵循所谓的“不要让自己重复”原则或叫DRY原则。DRY代码是可维护的代码。我对初级程序员的建议就是不要使用他们编辑器里的复制粘贴功能。对于中级程序员，我也建议他们在使用Or/+C之前要=思。

但我们应该做些什么来代替代码重复呢？最简单的解决方案通常是将代码移到一个阐数中，无论代码中哪段存在不同，这个函数都可以接收参数来区分。这并非一个非常面向对象的解决方法，但是通常它已经足够了。例如，我们有两份代码用來解压ZIP文件到两个不同的目录，我们可以轻易地写出一个函数，这个函数接收一个参数以决定哪个是应该存放解压文件的目录。这可能使函数本身变得有些难以阅读，但是一个好的函数名字和文本字符串可以轻松地弥补它，任何代码调用这个函数也将会更容易阅读。

这确实足够理论了！这个故事的寓意就是：永远去努力重构你的代码，使它变得更加易读，而不是写出仅仅是容易写出来的烂代码。

实践一下

让我们来看一下两种重用现有代码的方式。我们在一个全是文本文件的ZIP文件中写完替换字符串的代码之后，乂想将一个ZIP文件中的全部图片都压缩到640x480。看起来我们可以使用像在ZipReplace中使用过的相似的范例来实现。显然第一处改动就是要复制这个文件并将find_replace方法改为scale_image或其他相似的方法。但是，这真的不够酷。如果有一天我们要去改变unzip和zip方法，使其能够支持打开TAR文件将怎么做？或者我们想要保证保存临时文件的目录名是唯一的将会怎么办？在这两种情况下，

我们必须修改两个不同的地方！

我们将通过展示一个基于继承的方案来开始这个问题。首先我们需要修改原始的ZipReplace类，使它成为一个能够处理通用ZIP文件的超类：

```
import os
import shutil
import zipfile

class ZipProcessor:
    def __init__(self, zipname):
        self.zipname = zipname
        self.temp_di rectory = "unzipped-{}"•format(
             zipname[:-4])

    def _full_filename(self, filename):
        return os.path.join(self.temp_directory, filename)

    def process_zip(self):
        self.unzip_files()
        self•process files(>
        self.zip_files()

    def unzip_files(self):
         os.mkdir(self.temp_directory)
         zip = zipfile.ZipFile(self.zipname)
         try:
             zip.extractall (self. temp__di rectory)
         finally:
             zip.close()

    def zip_files(self):
         file = zipfile.ZipFile(self.zipname, 'w *)
         for filename in os.listdir(self.temp_directory):
            file.write(self._full_filename(
                filename), filename)
         shutil.rmtree(self.temp_directory)
```

我们更改了 zipfile的filename property以避免和不同方法内部的局部filename变M相混淆。尽管实际上这并非一个对于设计的改变，但也有助于使代码更加可读。我们同时还去掉了一init_（search_string 和 replace_string）方法中针对ZipReplace的两个参数，然后我们将zip_f ind_replace方法重命名为process_zip，并使它调用一个名为process_files（尚未定义）的方法替代调用find_replac:e。这些名称的变化有助于表明我们的新类具备更为通用的性质。请注意，我们已经把指向ZipReplace的代码里的find_replace方法全部移除f，这里的代码现在没有任何业务逻辑。

这一新的ZipProcessor类并没有真的定义process_files方法，所以假如我们直接运行它，它将会抛出一个异常。正因为它实际上不是用来直接运行的，所以我们还需要移除在原始脚本底部的主函数调用。

现在，在我们继续我们的图片处理应用之前，让我们先利用这个父类把原有的zipsearch 修改好：

```
from zip_jprocessor in^>ort ZipProcessor
import sys
import os

class ZipReplace(ZipProcessor):
    def __init__(self, filename, search_string, replace_string):
        super () .init__(filename)
        self.search_string = search- string
        self.replace_string = replace_string

def process-files(self)
    '1 'perform a search and replace on all files in the temporary directory' ' *
    for filename in os.listdir(self.temp_directory):
        with open(self._full_filename(filename)) as file:
            contents = file.read()
        contents = contents.replace(
                self.search_string, self.replace_string)
        with open(
             self •- full—filename (filename) , 'fw") as file:
             file.write(contents)
```

```
if __name__ == *'_main_:
    ZipReplace (*sys. argv[l: 4]) .process zip ()
```

这段代码由于从父类继承了对ZIP文件的处理能力，和原来的版本比起来会稍短一点。我们首先导人了刚刚写出的基类，并使ZipReplace扩展该类，然后用super(>方法初始化它的父类。find_replace方法仍在此处，但我们把它重命名为process_files，使得父类可以调用它。由于这个新名字并不像旧名字那样有描述性，所以我们增加一个文本字符串来描述它的作用。

我们已经做了相当多的工作，现在，可以认为我们有了一个功能上和最开始写出的版本没什么区别的程序！但是通过我们的这些工作，现在我们能很容易地写出其他类来操作一个ZIP档案中的文件，例如我们的图片尺寸修改器。进一步，假如我们想改进压缩功能，我们可以只修改基类ZipProcessor, 从而使所有的类都获得这个改进。代码的维护从此变得更为高效。

看看多么简单，现在创建一个图片尺寸修改的类可以直接获得ZipProcessor的功能。（注意：这个类需要安装第三方库pygame。可以从如下网址下载到：http: / / www. pygame. org/ 。）

```
from zip_processor import ZipProcessor
import os
import sys
from pygame import image
from pygame. transform import scale

class ScaleZip(ZipProcessor):

    def process—files(self):
        '1'Scale each image in the directory to 640x480[11]'
        for filename in os.listdir(self.temp_directory):
            im = image. load (self .__full_filename (filename))
            scaled = scale(im, (640,480))
            image.save(scaled, self._full_filename(filename))

if __name__ == "_main_:
    ScaleZip(*sys.argv[1:4]).process_zip()
```

我们所做的早期T:作终于有了回报！看现在这个类是多么简洁。我们要做的一切就是

打开每个文件（假设它是一个图片，如果文件不能被打开，程序将毫无疑问地崩溃），裁剪它，然后把它储存回去。ZipProcessor负责压缩和解压缩的工作，而我们的代码则不需要任何额外的工作。

或者我们可以使用组合

现在，让我们试着用基于“组合”的解决方案来解决同一个问题。尽管我们想完全把范例从“继承”改为“组合”，其实我们只需要对ZipProcessor类做一个小的更改就可以了：

```
import os
import shutil
import zipfile

class ZipProcessor:
    def __init__(self, zipname, processor):
        self.zipname = zipname
        self.temp_directory = "unzipped-{}".format(
            zipname[:-4])
        self.processor = processor

    def _full_filename(self, filename):
        return os.path.join(self.temp_directory, filename)

    def process_zip(self):
        self.unzip_files()
        self.processor.process(self)
        self.zip_files()

    def unzip_files(self):
        os.mkdir(self.temp_directory)
        zip = zipfile.ZipFile(self.zipname)
        try:
            zip.extractall(self.temp_directory)
        finally:
            zip.close ()
```

```
    def zip_files(self):
        file = zipfile.ZipFile(self.zipname, 'w *)
        for filename in os.listdir(self.temp_directory):
            file.write(self.—full—filename(filename}, filename)
        shutil.rmtree《self•temp—directory)
```

我们所要做的全部就是使初始化函数允许接收一个processor对象作为参数。现在process_zip函数会调用这个processor对象中的方法，该方法接收一个ZipProcessor类本身的引用作为参数：：如此，我们就可以把ZipReplace类改成一个合适的processor对象，而不再需要使用继承广：

```
from zip_processor import ZipProcessor
import sys
import os

class ZipReplace:
    def __init__(self, search—string,
                 replace—string):
        self.search_string = search—string
        self.replace_string = replace_string

    def process(self, zipprocessor):
        '''perform a search and replace on all files in the
        temporary directory*'''
        for filename in os.listdir(
                zipprocessor.temp - directory):
            with open(
                zipprocessor._full_filename(filename)) as file:
                contents = file.read()
            contents = contents.replace(
                self•search—string, self.replace_string)
            with open(zipprocessor—filename(
                    filename), "w") as file:
                file•write(contents)
if __name__ == '•_main__":
    zipreplace = ZipReplace(*sys.argv[2:4])
    Zipprocessor(sys.argv[1], zipreplace).process—zip<)
```

实际上我们真的没做出太多的改变：类不再继承自ZipProcessor;当我们处理文件时，我们接收一个zipprocessor对象，它提供给我们一个函数用来计算_full_filename。在最下面的两行，每当我们从命令行运行时，首先会构建一个ZipReplace对象，然后该对象会传递到ZipProcessor构造®数中，这样两个对象便可以相互通信了。

这样的设计在功能上是个极好的分割。现在我们有了可以接收任何一个拥有实际处理过程process方法的对象作为参数的ZipProcessor,还有了可以被传递到任何一个想要去调用process函数的方法、闲数或是对象的ZipReplace,压缩过程的代码再也不需要由于继承关系被绑定在一起。举个例子来说，现在我们可以很轻松地把代码应用到本地或是网络的文件系统中，或者不同的压缩文件格式，比如RAR档案。

任何继承关系都可以通过建模替代为组合关系（把“是一个 is a)”变为“有一个父类 has a parent T),但这并不意味着总要这样做而且反过来的操作通常是不正确的，大部分组合关系都不能（正确地）被建模为继承关系。

案例学习

在本章的案例学习中，我们将试图更深人地探究以下问题：“我应该在何时选用一个对象而非一个内置类型？”我们将会为一个可以用于文本编辑器或是文字处理器的Document类进行建模，它应该包含什么样的对象、函数，或是property呢？

我们可以从一个str开始编写Document的内容，但字符串是不可变的。一个可变的对象即它是可以被改变的，但是str是不可变的。如果不创建一个新的字符串对象，我们不可能对str插人或是移除一个字符。而这会导致Python的垃圾回收器要在我们操作之后清理大fi遗留的str对象$_3$所以，我们需要使用字符列表来代替字符串，这样我们就能随意修改了另外 Document需要知道列表中指针的当前位置，同时还要存储文档的文件名。

现在，Document中应该有哪些方法呢？对于一个文本文件，我们可能有很多种操作，比如插人或删除字符，剪切、复制、粘贴以及文杓的保存和关闭。这看起来需要大量的数据和行为，所以把这些东西都放到我们的Document类中就有意义多了。

于是问题来了，这个类是否应该由一串如str文件名、int指针位置、一个字符list这样的基本Python对象来组成呢？还是它们中的一些或全部应该专门地被定义为符合它们fi身的对象？至于每一行和每一个字符，它们乂是不是需要也有自己的类呢？

我们下面将一一回答上面这些问题，但首先让我们先设计一个最简单的可能的Document类，来看看它能做什么。

```
class Document:
    def —init___(self):
        self.characters =[]
        self.cursor = 0
        self.filename =''

    def insert(self, character):
        self.characters.insert(self.cursor, character)
        self.cursor += 1

    def delete(self):
        del self.characters[self.cursor]

    def save(self):
        f = open(self.filename, 'w')
        f.write(¹'.join(self.characters))
        f.close()

    def forward(self):
        self.cursor +=1

    def back(self):
        self.cursor _= 1
```

这个简单的类允许我们在编辑一个基本文档时得到完全的控制权。让我们看一下它的执行结果：

```
»> doc = Document ()
»> doc. filename = "test document"
»> doc . insert (' h *)
»> doc. insert (' e')
»> doc. insert (¹1')
»> doc. insert (' 1')
»> doc. insert (' o¹)
»> '"'.join(doc.characters)
```

```
•hello'
»> doc. back ()
»> doc.delete ()
»> doc. insert (' p')
»> M'. join (doc. characters)
1hellp'
```

看起来它已经可以工作了。如果我们能把键盘上的字符和箭头键和这些方法关联起来，那么一个文档就能很好地被跟踪记录了。

但是如果我们想关联的不只是箭头键，还有//ome和 键 呢 ？ 我 们 就 需 要 为Document类再添加一些方法，来实现在字符串中对换行符（Python中，换行符或\n代表了一行的结束以及新的一行的开始）向前或向后的搜索，并能够跳到搜索到的换行符那里。但是，如果我们想实现每一个可能的移动操作（按词移动、按句移动、Page吵 Page Down、行尾、缩进或其他），那这个类的规模可能会变得巨大无比。更好的方法可能是把这些方法每个都放在一个单独的类中。我们要做的就是要将指针这个属性放到对象中，使这个对象能够知晓它的位置并能够对该位置进行操作。我们还可以将向前和向后的方法加到这个类中，并再添加一些方法来响应//owe和键：

```
class Cursor:
    def __init__(self, document):
        self.document = document
        self.position = 0

    def forward(self):
        self.position += 1

    def back(self):
        self.position -= 1

    def home(self):
        while self•document•characters[
                self.position-1] != '\n':
            self.position -= 1
            if self.position == 0:
                #在换行符前已经到了文件开头处
                 break
```

```
    def end(self):
        while self.position < len(self.document.characters
                )and self.document.characters[
                    self.position] != *\n':
            self.position += 1
```

这个类需要一个文档作为初始化的参数，这样所有的方法便可以访问文档字符列表中的内容了。这个类还为向前和向后移动提供了简单的方法，像上面说的，移动到行首 home ）和行尾 end ）位置。

$ / •试着去访问一个空文档的行首位置，那么它就会崩溃。为了便于阅读，这些事例代码都写得很短，所以也就意味着它们是不具防御能力的。你可以为这段代码添加错误检查，并把它当作一个练习，对你来说这可能是个提高自己异常处理技术的绝佳机会。

这段代码并不十分安全,，你可以轻易地移动到一个行尾之后的位置，还有如果你

Document类本身几乎不需要改变，除了要删除两个已经被移动到Cursor类中的方法：

```
class Document:
    def __init__(self):
        self.characters =[]
        self.cursor = Cursor(self)
        self.filename = 1'

    def insert(self, character):
        self.characters.insert(self.cursor.position,
                character)
    self.cursor.forward()

    def delete(self):
        del self.characters[self•cursor•position

    def save(self):
        f = open(self•filename, 'w')
        f.write(''.join(self.characters))
        f.close ()
```

我们简单地把访问旧指针整数的任何相关代码都更新为用新对象来访问，我们可以测试，行首（home)方法真的可以实现将指针移动到换行符。

```
»> d = Document ()
»> d. insert ('h')
»> d. insert (' e1)
»> d. insert (* 1')
»> d. insert ('1 *)
»> d. insert ('o')
»> d. insert ('\n')
»> d. insert ('w')
»> d. insert ('o')
»> d. insert ('r')
»> d. insert ('1')
»> d. insert (* d')
»> d. cursor. home ()
»> d.insert("*")
»> print ("" . join(d.characters))
hello
♦world
```

现在，由于我们大量使用了字符串的join函数（为了连接字符使我们可以得到真正的文构内容），我们可以在Document类中添加一个property属性来返回完整的字符準：

```
Qproperty
def string(self):
    return .join(self.characters)
```

这会使我们的测试变得简单些：

```
»> print (d. string)
hello
world
```

这个框架已经足够简单使它能扩展创建为一个完整的文本文档编辑器[3]现在，让我们使它对包含粗体、下画线、游体的富文本也能进行编辑。有两个办法能让我们处理它：第1种，我们在字符列表中插人“伪”字符来扮演指令的角色，例如“加粗字符直到遇到停止加粗符号”；第2种方法即为每个字符都增加额外的信息来表明它应该有怎样的格式。尽

管前一种方法可能更为常见，但我们还是要实现后者。为此，显然我们需要为字符设计一个类。这个类要有一个属性代表字符，同时还要有3个属性分别代表字符是否为粗体、斜体或有下画线。

嗯，等等！这个字符类难道没有任何方法吗？如果没有，或许我们应该使用一个包含了多个Python数据结构的数据结构来代替，一个元组或是命名的元组可能就足够了。有没有什么动作是我们想要对字符操作或是在字符上调用的呢？

好吧，显然我们可能想要对字符做更多的操作，比如删除或复制它们，但是这些事应该在Document的级别被处理，因为它们真的修改了字符列表。那么有没有什么事需要在单个字符上完成的呢？

事实上，现在我们要思考一下Character实际上是什么……它是什么？如果我们说Character是一个字符串，这种说法对吗？可能我们应该在这里使用继承关系，然后我们就可以利用str实例带来的众多好处。

哪些方法是我们要讨论的呢？这里有startswith、strip、find、lower等许多，这些方法中的大部分都被期望可以工作于包含多于一个字符的字符串。与之相对，如果Character是str的子类，我们可能最好还是要重写_init_方法，使它在接收多字符字符串时抛出一个异常。但是，由于我们不需额外花费就获得的所有这些方法都不会真的应用到我们的Character类中，最后的结果我们还是不应该使用继承关系。

这让我们又回到了第一个问题：Character真的应该被用作一个类吗？在Object类中有一个十分重要而又特殊的方法可以让我们加以利用来表示我们的字符。这个方法就是_3七l：_（两个下画线，就像这个方法在字符串的操作函数中常被用到，如print和str构造函数中用来转换任意类成为一个字符串。它的默认实现有些无聊，比如打印模块的名字或是类和它在内存中的位置。但是如果我们覆盖它，就可以让它打印出任何我们想要的。在我们的实现中，可以通过给字符串加上特殊字符的前缀来表示它们是否是粗体、斜体或者有下画线。因此，让我们创建一个类来表示字符，如下：

```
class Character:
    def ____init____(self, character,
                bold=False, italic=False, underline=False):
        assert len (character) == 1
        self.character = character
        self.bold = bold
        self.italic = italic
        self.underline = underline
```

```
    def  str_(self):
        bold = ••*., if self.bold else ¹ ’
        italic =          if self. italic else ’ ¹
        underline = if self.underline else ’’
        return bold + italic + underline + self.character
```

这个类允许我们创建字符，并且在str〈〉函数应用到它们时，可以给它们添加特殊字符作为前缀。这里也没有太多激动人心的东西。我们只需要对Document和Cursor类进行微小的修改就可以和这个类协同工作了。在Document类中，我们在insert方法的开始添加两行代码：

```
def insert(self, character):
    if not hasattr(character, 'character'):
        character = Character(character)
```

这段代码稍微有些奇怪。它的基本目的是检查Character或str类中字符是否被传入了。传人的字符被包装为一个Character类，所以列表中的全部对象都是Character对象。然而，完全有可能有人在使用我们的代码时，并不想使用Character也不想用字符串，而是使用鸭子类型。如果在鸭子类型类中的对象有一个字符属性，我们就假设它是一个“类Character”对象，如果没有，我们就认为它是一个“类str”对象，并将之包装为一个Character。这将帮助我们的程序利用鸭子类型和多态的优势，只要一个对象具有字符属性，它就可以被Document所用。这将是非常有用的。例如，我们想开发一个供程序员使用的具有语法高亮功能的编辑器，我们就需要character具备额外的数据，比如这个字符属于哪种令牌类型。

此外，我们还需要修改Document的字符串property属性，使它能够接收新的Character值。我们所要做的全部就是在我们连接这些字符之前，对每个字符都调用str〈〉函数：

```
0property
def string(self):
    return *"’•join((str(c) for c in self.characters))
```

这段代码使用了生成器语法，该语法我们将在下一章中讲到。在这里你可以将它理解为一个用于对全部对象按序执行特定动作的快捷操作。

最后，在home和end函数中，当我们查看字符是否是新的一行的时候，我们还需要检查Character. character, 而不是像之前那样简单地存储这个字符串字符。

```
    def home(self):
        while self.document.characters[
                self.position-1].character != '\n':
            self.position -= 1
            if self.position == 0:
                #在换行符前已经到了文件开头处
                break

    def end(self):
        while self.position < len(
                self.document.characters) and \
                self.document.characters[
                        self.position
                        ].character != '\n':
            self.position += 1
```

这样就完成了字符的格式化。让我们来测试一下：

```
»> d = Document ()
»> d. insert ('h')
»> d. insert ('e')
»> d. insert (Character (* 1' , bold=True))
»> d. insert (Character ('1' # bold=True))
»> d. insert ('o')
»> d. insert ('\n1 )
»> d. insert (Character ('wf , italic=True))
»> d. insert (Character (1 o1 , italic=True))
»> d. insert (Character ('r * , underline=True))
»> d.insert (* 1')
»> d. insert (f d')
»> print (d. string)
he*l*lo
/w/o rid
»> d.cursor.home ()
»> d.delete ()
»> d.insert ('W')
»> print (d. string)
he*l*lo
```

```
W/o – rid
»> d. characters [0] .underline = True
»> print (d. string)
– he*l*lo
W/o– rid
»〉
```

正如期望的那样，无论何时我们打印字符串，每个加粗的字符前都会有一个*,每个斜体的字符前都会有一个/,每个有下画线的字符前都会有一个_。我们的函数看起来都能正常r.作，同时我们也能对列表中的字符进行修改。这样，我们就有了一个可以被嵌人到用户界面，并且可以搭配键盘输入和屏幕输出的富文本文档对象来处理对象。当然，我们真正想要的是在屏•幕上显示出粗体、斜体和下两线字符，而不是我们的_3七1_方法，但是这个类对丁-我们所要求的基本测试已经足够了。

练习

我们看过了在一个面向对象的Python程序中对象、数据和方法以各种方式进行交互。像往常一样，你的第一个想法应该是如何将这些原则应用到自己的丁_作当中。你是否有闲置的脚本可以应用面向对象的管理方式来重写？浏览一下你的旧代码，找找那些不是动作的方法。如果它的名字不是动词，那么请尝试用property属性来重写它。

请思考一下你写过的代码，无论是用哪种语言。它打破了 DRY原则吗？有没有重复的代码？你是不是曾经复制粘贴过代码？有没有因为你理解不了原始的代码，于是写出了w个版本的相似代码？现在就回去看看你最近写的一些代码，看看你是不是可以使用继承或组合来重构耶些重复的代码。试着选择一个你还有兴趣维护的项目，并旦代码没有老旧到你都不想與碰它。这样才有助于在你改进它的时候保持兴趣！

现在，回头看看我们在这一章中碰到的一些例子。从使用property属性来缓存检索数据的网页这个例子开始在这个例子中，有一个明显的问题就是缓存是不会刷新的。通过在getter屈性中添加一个超时机制，可以做到只有当超时之前请求这个页面时才返回缓存页面，，你可以使用time模块（time. time （〉 – an_old_time可以返回从an_old_time以来经过的秒数）来确定缓存是否过期了。

再来看看基于组合和基于继承的两个版本的ZipProcessor。我们写了一个基于继承的ScaleZipper，但并没有将它融合到组合版本的ZipProcessor。试着写出一个组合版本的ScaleZipper，并比较这两个版本的代码。你觉得哪个版本更容易使用呢？哪个

更为简洁？哪个更易读？当然这些都是主观的问题，我们每个人的答案都不相同。然而有fl己的答案是非常重要的，如果你发现你喜欢继承胜过组合，你必须注意不要在口常编码中过度使用继承。如果你喜欢组合，确保你不会错过一个创造简洁的基于继承的解决方案的机会。

最后，清在我们创造的各个类的案例中添加一些错误处理。它们应该确保只存单个字符被输入，确保你不会试图移动光标到文件的结尾或开始之前，确保你不删除一个不存在的字符，并且确保你没有保存一个没有文件名的文件。尽可能地想到更多的边界情况，并且对它们做出解释。考虑不同的方式来处理它们，当用户试图跨越文件的末尾时，你应该抛出一个异常还是只是停留在最后一个字符？

在日常编码中，多留意复制和粘贴命令。在编码过程中每次使用它们的时候，想想改进程序的组织关系是不是个更好的办法，如果那能让你想要复制的代码只存在一个版本。

总结

在本章中，我们专注于识别对象，尤其是那些不会立即显现出来的对象，专注于管理和控制对象。特别是，我们讲到了 ：

•为什么对象应该有数据和行为。

. property属性是怎样使数据和行为之间的区别变得模糊的。

• DRY原则和重复代码的愚蠢。

•通过继承和组合来减少代码重复。

在下一章我们将讨论一些内置的Python数据结构和对象，专注于它们在面向对象中的属性以及它们可以怎样被扩展和适用。

6

第6章 Python数据结构

在之前的例子中，我们已经见过许多Python内置数据结构的使用方法了。你可能已经通过介绍性的书籍或者教程对它们有了一些了解在本章中，我们将讨论这些数据结构的面向对象功能，以及什么时候应该使用它们来代替常规的类，什么时候不该使用它们通过本章你会学到：

. 元组〇

• 字 典 。

•列表和集合。

.怎样及为何扩展内置对象。

空对象

让我们从最基本的Python内置数据结构开始，我们已经见过它很多次了，我们都会从创造的类中扩展出它来：object。从技术上讲，我们可以实例化一个对象而无需编写一个子类：

```
»›O = object ()
»> o.x = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError:    'object¹ object has no attribute 'x'
```

不幸的是，正如你所看到的，对于一个直接实例化的Object是不可能赋予其任何属性

的。这并不是因为Python开发者想要迫使我们编写自己的类，也不是因为任何其他如此阴险的原因.，他们只是想要节省内存，很多内存。如果Python允许一个对象有任意厲性，那就需要一定M的系统内存去跟踪每个对象都有哪些属性，用来存储属性的名称和值。即使没有属性需要存储，一定大小的内存也会分配给潜在的新属性。鉴于在一个典塑的Python程序中都会有几十、儿&乃至上千的对象（每个类都会扩展出对象），少fi的内存很快就会变得巨大。所以Python默认禁止object拥有任何属性，其他一些内置数据结构也一样。

在我们自己的类中可以使用“插櫂（slot）”来限制任意property属性，但插槽超出了本书的范闹，不过如果你想获得更多的信息，现在有了一个可以搜索的关键同在正常使用中，使用插槽并没有太多好处但是如果你写了将在整个系统中被复制上千次的对象，插槽可以帮助你节省内存，就像它对object做的那样，

不过还是让我们来简单地创建一个IN Q的空对象类，我们已经在最旱的例子中见到过它了：

```
class MyObject:
    pass
```

正如我们看到的，为这些类添加一个属性也是可能的：

```
»> m = MyObject ()
»> m.x = "hello"
»> m.x
'hello1
```

显然，如果我们想把厲性分组，可以就像这样将它们存储在一个空对象中。但我们通常不会这样做，w为柯一些其他的内置数据类型是专门用来存储数据的，很快我们就将看到它们。本书一再强调的，类和对象应该只在你想要同时指定数据和行为的时候被使用。这里编写一个空类的主要原因是，我们想要快速地确定一些东西，并且知道我们会在不久之后回来将行为添加到这个类匕.使一个类适配一些行为，要比用对象替换一个数据结构并改变所有对它的引用简单得多..从一开始就决定好数据只是数据还是一个伪装的对象是很重要的一旦这样的设计决定了，其他部分的设计通常也就能确定了。

元组和命名元组

元组是可以按序存储特定数量其他对象的对象。元组是不可变的，所以我们不能随意

对其添加、删除或替换对象。这看起来像是巨大的限制，但实际上，如果你需要修改一个元组，那么肯定是你错误地使用了数据结构（列表会更合适）。元组的主要好处就是它的不可变性，我们可以把它作为字典的键，或者用在一个对象需要一个散列值的地方。

元组是用来存储数据的，行为不能被存储在元组中。如果我们有需求要用一个行为来操纵一个元组，我们必须把这个元组传入到一个函数中（或在另一个对象的方法中）执行该动作。

元组通常存储各A不同的值。例如，我们不会把3个股票代码存在一个元组中，但我们可能会创建一个元织来存储股票代码、当前价格、当日最高价和最低价。一个元组的主要@的是把不同的数据聚合到一个容器中。实际上. – 个元组可以是我们前面提到的“没有数据的对象”的最简单的替代工具。

我们在创建一个元组时，可以爪逗号分隔每一个值。通常元组被包括在一个括号中，这会使它更容易阅读. 同时也能使它与一个表达式的其他部分区分开来。但是这并不总是必需的，以下两个写法是相同的（它们都为一家赢利的公司记录其股票名称、当前价格、最高价和最低价）：

```
»> stock = "GOOG", 613.30, 625.86, 610.50
»> stock2 = ("GOOG", 613.30, 625.86, 610.50)
```

如果我们把一个元组集成到其他的对象中，如函数调用、列表解析、生成器时，括号就是必需的。否则解释器就不可能知道这是一个元组还是另一个参数。例如，下面的函数接收一个元组和一个日期作为参数，并返回一个包含日期和股票最高值和最低值之间的中间值的元组：

```
import datetime
def middle(stock, date):
    symbol, current, high, low = stock
    return (((high + low) /2), date)

mid_value, date = middle(("GOOG", 613.30, 625.86, 610.50),
          datetime.date(2010, 1, 6))
```

在函数调用中，直接用逗号将各个值分隔开，并将整个元组包含在括号中可以直接创建一个元组。紧随其后.用一个逗号来将这个元组和第2个参数区分开。

这个例子还解释了元组的拆分。函数内部的第1行就将元组中的股票参数拆分为4个不同的变量。在拆分时，元组必须拆分为与其长度相等的变量，否则解释器将抛出一个异

常。在整个代码块的最后一行，我们还可以看到另一个元组拆分的例子。函数返回一个元组，但直接被拆分为两个值：mid_value和date。当然，这看起来有点奇怪，因为我们在最开始就把日期作为参数提供给了函数，但是在这里它却给了我们一个机会看到元组的拆分是怎样工作的。

```
»> stock = ”GOOG'., 613.30, 625.86, 610.50
»> high = stock [2]
»> high
625.86
```

我们还可以用分片符来提取更大元组中的数据：

```
»> stock[1:3]
(613.3, 625.86)
```

这些例子一方面说明了元组的用处，另一方面也说明了元组的一个主要缺点：可读性。怎样才能比阅读这段代码的人知道一个特定元组的第2位的值是什么？他们可以从我们分配给变ffl的名字来猜测，比如某些元组中的high,但是如果我们仅仅是在计算中访问元组的值而没有给它分配名字，那么就没有这样的指代了。他们只得通过通读代码来找到元组被卢明时的作用。直接访问元组的成员在某些情况下是好的，但不要养成这样的习惯$_3$这种所谓的“魔数”（数字似乎凭空在代码中出现而没有明显意义）是许多编码错误和长时间沮丧调试的原因。只有当你知道所有的值即将立刻变得有意义，并且通常访问它需要拆分的时候再来尝试使用元组。如果你直接访问某个成员或者使用分片，并且这个值的用途并不明显，那么至少请你添加一个注释来说明它是从哪里来的。

命名元组

那么，如果我们想要为某些值分组，但又知道我们需要经常单独访问它们的时候，应该怎么做呢？好吧，我们可以使用一个空对象，正如在前一节中所讨论的那样（但这样做意义有限，除非我们预期随后会添加行为），或者我们可以使用字典（如果我们不知道究竟有多少或哪个特定数据将被存储时最有用），这是我们将在下一节中讨论的。

然而，如果我们不需要为对象添加行为，并且我们也预先知道有哪些属性需要存储，这时我们可以使用命名元组。命名元组是含有属性的元组，它是没有行为的对象。命名元组是一个组织数据的极好方式，尤其是对只读数据。

构建一个命名元组需要比构建一个普通元组多一点工作。首先我们必须引入

namedtuple，因为它不在默认的命名空间中。之后我们通过起一个名字及描述它的属性来定义一个命名元组。这将返回一个类似于类的对象，我们可以通过给出所需的值来实例化它，无论多少次都可以：

```
from collections import namedtuple
Stock = namedtuple("Stock", "symbol current high low")
stock = Stock("GOOG", 613.30, high=625.86, low=610.50)
```

namedtuple构造函数需要两个参数，第1个参数是对该命名元组的标识符，第2个参数是一个用空格隔开的字符串用来指定命名元组含有的属性。其中第一个属性应该能够被列表化，接着紧跟一个空格，之后是第2个属性，再来另一个空格，如此反复。其结果是一个可以用来实例化其他对象的对象。这个新的对象可以像一个正常的类那样调用。它的构造函数必须含有正确数量的参数，这将使它们按照顺序来传递，或者以关键字参数的方式来传递参数，但所有属性必须全部被指定。我们创建多少个这个“类”的实例都是可以的，并且它们每个都可以有不同的值。

命名元组像普通元组那样可以打包或拆分，不过我们还可以访问其中某个属性，就好像它是一个类的属性那样：

```
»> s tock. high
625.86
»> symbol, current, high, low = stock
»> current
613.3
```

记住，构建命名元组需要两步。首先，使用collections.namedtuple来创建一个类，然后创建该类的实例。

命名元组对许多“仅含数据”的表示来说都是更好的选择，但它们也并不是适合所有的情况。就像元组和字符串那样，命名元组是不可变的，所以一旦被设定，我们便不能再来修改它的属性。例如，N从开始讨论这个话题后，股票的当前值就下降了，但我们并不能给它设置新的值：

```
»> stock.current = 609.27
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
AttributeError: can't set attribute
```

如果我们需要改变存储的数据，那么字典可能才是我们需要的。

字典

字典是非常有用的对象，它使我们能够直接将一个对象映射到另一个对象：一个具有属性的空对象就是一种字典，property的名称可以映射到property的值上，实际上这比它听起来更接近真理，因为在对象的内部，通常属性就是以一个字典来表示的，对象中的property和方法就是字典的值。即便是在一个模块中，其内部属性也储存于一个字典中。

在查询值上字典是非常高效的，给出一个具体的查询对象就可以直接映射出其值。当你需要基于另一个对象去查找一个对象时，最应该使用的就是字典。在字典中，被存储的对象称为值，被用作索引的对象称为键。我们已经在一些前面的例子中见到过字典的语法，但为了完整性，我们再重讲一次。可以使用dict 构造函数或{}快捷语法来构造一个字典。在实践中，几乎总是使用后一种方式。我们可以通过冒号在预填充字典时区分键和值，并使用逗号来分隔每组键值对。

例如，在股票的应用中，我们会经常想要通过股票代码来查找它的价格。我们可以创建一个以股票代码为键，以一个包含现价、最高价、最低价的元组为值的字典。如下：

```
stocks = {"GOOG": (613.30, 625.86, 610.50),
          "MSFT": (30.25, 30.70, 30.19)}
```

正如我们在之前的例子中见到的，我们可以通过在方括号内请求某个键，来查找该键在字典中的值。如果字典中不存在这个键，那么就抛出一个异常：

```
»> stocks ["GOOG"]
(613.3, 625.86, 6X0.5)
»> stocks ["RIM"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'RIM*
```

当然，我们可以捕获KeyError这个异常来处理它，但我们还有其他的办法。记住，即便我们使用字典的首要0的是存储其他对象，字典本身也仍是一个对象。因此，字典是有一些相关行为的。这些方法中最有用的一个就是get方法，它接收一个键作为第一个参

数，以及一个当键不存在时的可选默认值。

```
»> print (stocks .get ("RIM"))
None
»> stocks .get ("RIM", "NOT FOUND"}
'NOT FOUND'
```

如果想要控制得更多，我们可以使用setdefault方法。如果键在字典中，该方法的行为如同get方法：返回键的值。相反，如果键不在字典中，它不仅会返回我们在调用方法时给出的默认值（就像get那样），还会将该键设置为相同的值。另一种理解setdefault的方法可以认为当且仅当字典中之前不存在该键的值时，setdefault才会设置该键的值。然后返回字典中的这个值，无论是一个已经存在的还是新提供的默认值。

```
»> stocks.setdefault("GOOG", "INVALID")
(613.3, 625.86, 610.5)
»> stocks.setdefault("RIM",     (67.38, 68.48, 67.28))
(67.38, 68.48, 67.28)
»> stocks [ "RIM"]
(67.38, 68.48, 67.28)
```

GOOG股票已经在字典中了，因此，当我们试阎用setdefault将它设置为无效值时，它只是返回已经在字典里存在的值。而RIM不在字典中，所以setdefault返回默认值并在字典中为我们设置RIM的新值。之后我们检查这个新股票，是的，它已经在字典中了。

在字典中还有3个非常有用的方法：keys（）、values（）、items（）。前两个方法返回一个包含字典中所有键和所有值的迭代器。我们可以像列表一样使用它们，或是通过for循环来遍历它们，如果我们想处理所有的键或值。items（>方法可能是最有用的，它返回一个对字典中每个项目形成一个如(键，值)对这样的元组的迭代器。这个迭代器结合元组拆分，可以在for循环中使遍历键和值事半功倍。下面这个例子就是用来打印字典中的每个股票的当前值：

```
»> for stock, values in stocks. items ():
...    print("{} last value is {}".format(stock, values[0]))

GOOG last value is 613.3
RIM last value is 67.38
MSFT last value is 30.25
```

每个键/值的元组拆分为两个变量：stock和values（我们可以使用任何我们想用的变量名称，但这两个看起来最贴切），然后在一个格式化字符串中打印它们。

注意，股票并不会按照它们插入字典时的顺序打印出来。为了使键查找足够快，字典使用了高效的算法（该笕法称为哈希），因此在其内部是无序的。

如此，当字典被实例化之后，从中检索数据有很多种方法。我们可以使用方括号作为索引语法、get方法、setdefault方法或者遍历items方法等〇

最后，正如你可能已经知道的，我们可以使用与检索值相同的索引语法来在字典中设置值：

```
>» stocks ["GOOG"] = (597.63, 610.00, 596.28)
»> stocks ['GOCX5']
(597.63, 610.0, 596.28)
```

今天谷歌的价格较低，所以我更新了字典中的元组值。我们可以使用该索引语法为任意键设置值，无论该键是不是已经存在于字典之中。如果已在字典中，那么旧的值将被替换为新的；否则，一个新的键/值对将会被创建。

到目前为止，我们已经使用过字符串作为字典的键，但并不局限于字符串的键。使用字符串作为键很普遍，尤其是当我们将数据存储在字典中只为了将它们收集在一起（而不是使用一个有很多命名property的对象）。其实我们还可以使用元组、数字，或者我们自定义的对象作为字典键。除此之外，我们甚至可以在一个单一的字典中使用多种不同类型的键：

```
random -keys ={}
random_keys [ "astring" ] = ''somestring"
random— keys[5)="aninteger"
random - keys[25.2] = "floats work too"
random - keys[("abc", 123)] = "so do tuples"

class AnObject:
    def —init- (self, avalue):
        self.avalue = avalue

my_object = AnObject (14)
random -keys[my_object] = "We can even store objects"
my_obj ect.avalue = 12
try:
```

```
    random_keys [ [1,2,3] ] = "we can't store lists though,f
except:
    print("unable to store list\n")

for key, value in random }ceys.items(>:
    print (•'{} has value {} H . format (key, value))
```

这段代码展示了在一个字典中同时使用几个不同类型的键。它还给出了一种不能用作键的对象。我们已经广泛地使用过列表，在下一节我们将会看到更多使用列表的细节„由于列表可能在任何时候发生变化（例如添加或删除列表中的元素），所以列表不能被哈希成一个特定的值。可哈希的对象基本t都含有一个定义好的算法，可以将对象转换为一个整数值以便快速查找。这个哈希值实际上就会用来在字典中进行查找。例如，字符串映射为整数要基于字符串中的字符，而元组则结合了元组中各元素的哈希值，以某种规则被认为是相同的任意两个对象（如含有相同字符的字符串或具有相同值的元组）应该有相同的哈希值，并且对象的哈希值应该永不改变。然而，对列表来说，其内容可以发生改变，这将改变列表的哈希值（两个列表仅当它们的内容是一样的时才被认为是相同的 >。正因为如此，它们不能被用作字典的键。出于同样的原因，字典也不能用作另一个字典的键。

相比之下，字典的值对对象类型就没有限制了。例如，我们可以使用一个字符串类型的键映射到一个列表类型的值，或是用一个嵌套的字典作为另一个字典的值。

何时应该使用字典

词典是个多面手，它有许多种用法，，两种最主要的字典使用方法，一个是使字典中所有的键代表相似对象的不同实例。例如，我们的股票字典，这可以看作一个索引系统，我们使用股票符号作为值的索引，而值甚至可以是自定义的复杂对象，来包含买卖的决定或设置一个止损点，而不只是简单的元组。

第2种使用字典的设计是使字典的每个键代表一个单一对象的某些方面$_3$在这种情况下，我们可能会为每个对象使用一个单独的字典，且它们都有相似的（尽管通常不相同）键，，后一种情况可能与命名元组有些重叠，要注意.命名元组通常应该在我们确切知道哪些属性数据必须存储，并且所有数据必须一次性存储（当项目被构造时）的时候再使用。进一步来说，命名元组只有在我们知道这些数据不会发生改变才适合.:，其他情况下，例如我们需要在一个时间段内构建字典的键，或每个字典可能或多或少有不同键的集合，再或是我们需要频繁改变值时.字典是更好的选择。

使用 defaultdict

我们已经看到了如何使用setdefault来为一个不存在的键设置默认值，但是如果我们每次都需要在查找一个值时设置默认值，这可能就有点枯燥了。例如，我们要编写一段代码来统计一个字母在给定的句子中出现的次数，我们可以这样做：

```
def letter_frequency(sentence):
    frequencies = {}
    for letter in sentence:
        frequency = frequencies.setdefault(letter, 0)
        frequencies[letter] = frequency + 1
return frequencies
```

每次访问字典时，我们都需要检查这个键是不是已经存在对应的值，如果不存在则将其设置为零。当每次访问一个值为空的键时都需要像这样做，我们可以用另一个不同版本的字典，即 defaultdict：

```
from collections import defaultdict
def letter_frequency(sentence):
    frequencies = defaultdict(int)
    for letter in sentence:
        frequencies[letter] += 1
    return frequencies
```

这段代码看起来似乎不能T作。defaultdict类在它的构造函数中可以接收一个函数作为参数，当访问字典中一个不存在的键时，它就会以不含参数的形式调用该函数，创建一个默认值。

在这个例子中，所啁用的函数是int,也就是一个整数对象的构造函数。通常情况下，在代码中创建一个整数只需要键人一个整数数字。而如果使用int构造函数来创建一个整数，我们可以将想要创建的项目传给它（例如，将一连串的数字字符转换成一个整数）。但是如果我们调用int时不带任何参数，它就会简单地返回零。在这段代码中，如果字母在defaultdict中不存在，则在我们访问它时会返回零。之后我们为这个数字加1来表示我们发现了该字母的一个实例，当我们下次再找到一个时，就再次增加其值。

defaultdict类在创建容器时非常有用。如果我们想要创建一个字典来存储过去30天里的股价，我们可以以股票代码为键，在list中存储各天的价格。当我们第一次访问某个股票价格时，会希望创建一个空列表。我们可以简单地将list传入到defaultdict

构造函数中，它将会在每次访问一个空键时被调用。类似地，我们甚至还可以创建一个空字典，如果我们想把它与一个键结合起来。

当然，我们也可以编写自己的函数并把它传到defaultdict构造函数中。假设我们想创建一个defaultdict,其中的每个元素都包含一个元组，记录当时插人字典中的项目数f1和一个用来存储其他数据的空列表。没有人知道为什么我们要创建这样一个对象，但还是让我们来看看怎么实现它：

```
from collections import defaultdict
num_iterns = 0
def tuple—counter():
    global num_items
    num_iterns += 1
    return (num- items,[])

d = defaultdict(tuple—counter)
```

当我们运行这段代码时，可以在一行代码中同时做到访问空键并插入列表两个动作：

```
»> d = defaultdict (tuple—counter)
»> d['a² ] [1] .append("hello")
»> d['b'] [1] .append('world')
»> d
defaultdict(<function tuple—counter at 0x82f2c6c>,
{'a': (1, ['hello']), fb': (2, ['world'])})
```

当我们最后打印diet时，可以看到计数器真的在工作了。

▽一义 这个例子虽然简洁地展示了如何用自己的函数创建defaultdict,但实际上这并不是很好的代码.全局变量的使用意味着如果我们创建了 4个不同的defaultdict代码段，每个都使用一个tuple_counter,那么它将计算在所有字典中全部条目的数量，而不是为每一个字典记录不同的数。最好的办法是创建一个类，并将这个类中的一个方法传给defaultdict。

列表

列表是Python中最不面向对象的数据结构„尽管列表本身是对象，但Python中有很

多语法让列表的使用变得尽M毫不痛苦。不像其他那些面向对象的编程语言，Python中的列表简单好用•。我们既不需要引人它，也很少需要调用列表中的方法。我们不必显式地请求一个迭代器对象就可以遍历一个列表，同时我们也可以用m定义的语法构造一个列表(像字典那样〉。此外，列表解析和生成器表达式使得列表变得像瑞士军刀那般具有多种功能。

我们不会在语法的细节上深究，你已经通过网上的教程和本书前面部分的例子看过一些了。你不可能在没有学会怎样使用列表的情况下写出太长的Python代码。相反，我们将会介绍在什么时候应该使用列表，以及它作为对象的内在属性。如果你不知道如何创建列表或为列表追加元素，不知道怎样从一个列表中检索元素，以及不知道什么是“分片符”，你最好赶快去看Python的官方教程。它可以从下面的链接找到：

http://docs.python.org/py3k/tutorial/

列表，在Python中，通常用于当我们想要存储若干“相同”类型的对象的实例中，如字符串列表或数字列表。更常见的是，我们自定义对象的列表。在我们想要以某种顺序存储元索时更应该使用列表通常，这个顺序就是我们插入元素的顺序，但它们也可以按照某个标准进行排序。

像我们在前一章的案例学习中看到的，列表在我们需要修改内容时非常好用：可以在列表中任意位置插人或删除元素，或更新列表中的某个值。

像字典那样，Python的列表使用了极其高效&协调的内部数据结构，这使得我们可以史关注想要存储哪些元素，而不必关注怎样去存储a很多面向对象的编程语言为队列、栈、链表、阵列格式的列表提供了不同的数据结构。Python中确实为其中的一些类提供了特殊的实例，如果必须为超大数最的数据集提供最优访问的话。然而通常情况下，列表数据结构就iff以一次达成上面所有目的，程序员对如何访问它们有着完全的控制权。

在收集个别项目的不同属性时不要使用列表。举例来说，我们不想使一个存储厲性的列表具备特定的形式。元组、已命名元组、字典，甚至对象都比列表更适合达成这个目的。在某些语言中，它们可能可以创建一个每个元素都是不同类型的列表，例如，它可以以[’a’，1，’b\ 3]的形式来写我们的字母频率表。但这样，它们就不得不使用奇怪的循环来同时访问列表中的两个元素，或使用取模运算符来决定哪个位置应该被访问。

不要在Python中这样做。我们可以把相关联的项目用一个字典来分组，就像我们在前一个部分中做的那样（如果排列的顺序无关紧要），或是使用一个元组的列表。这里柯一个相，复杂的例子来演示我们如何使用列表来完成上面的有关频率的示例。它比使用字典的例子要复杂得多，并且说明了选择正确（或错误）的数据结构对我们代码的可读性有多么大的影响。

```
import string
CHARACTERS = list (string .ascii –letters) + [

def letter_frequency(sentence):
    frequencies = [(c, 0) for c in CHARACTERS]
    for letter in sentence:
        index = CHARACTERS.index(letter)
        frequencies[index] = (letter,frequencies[index][1]+1)
    return frequencies
```

这段代码首先给出r一个包含全部可能的字符的列表。string.ascii_letters属性提供了一个包含所有大小写字母的有序列表。我们将它转换为一个列表，然后使用列表连接（加号操作符使两个列表合并成一个）来添加另一个字符：空格。全部这些就是我们的频率列表中所有可用的字符（当我们试图添加一个不存在于这个列表中的字符时，代码将会跳出，不过一个异常处理程序就可以解决这个问题)。

函数第I行使用列表解析将CHARACTERS列表转变成一个元组的列表。列表解析是Python中一个非常重要的非面向对象的工具，我们将在下一章详细介绍它们。

然后，我们遍历语句的每个字符,首先找出每个字符在CHARACTERS列表中的索″jU这个索引与我们的频率列表中的索引是一样的，因为我们就是由第1个列表创建出第2个列表的。然后我们通过创建新元组丢弃旧元组来更新频率列表中的索引。但是，除去垃圾收集和内存浪费的问题之外，这仍旧是非常难以阅读的!

最终的代码当然是可以丁.作的，但和字典比起来它并不简洁。然而，这段代码比起先前的字典示例代码有两个优点。该列表将没有在语句中出现的字符的频率存储为零;此外，当我们使用这个列表时，它是有序的。下面的输出显示了区别。

```
»> letter–frequency(•'the quick brown fox jumps over the lazy dog'')
[(•a, ('b*,1), (fC r 1), (•d» ,1), Ce* r 3), Cf' rg,"
(, h, 2), ('i¹ , 1), (•j, , 1), d, 1), (T, 1>, ('m' 1>, ('n\ 1),
('0•, 4), ('p' /1)/ ('q', 1), Cr\ 2), Cs'f 1>, (.t, 2> , Cu', 2)t
rv, 1), ('W, 1), (, x, 1), (, y, 1), (, z, , 1), (, A., 0), ('B¹, 0),
(•C., 0)/ (■D, 0), (, E, 0), (fF- f 0), (•G, , 0), (■H, 0> , CI\ 0),
rj, , 0), (| K, 0), (, L., 0>, (, M., 0), (•N., 0)/ (, 0, , 0), ('P', 0),
(•cr, 0), (, R., 0), ('s, 0>, (, T, 0), ('U', 0), (•V, , 0), CW ,0),
(, x, 0>, CY* , 0), CZ\ 0), (. 8)]
```

字典版本的代码可以通过为字典所有可用字符预填充零值来提供这些优点，并且每当

我们需要它有序时，可以对返回的字典依键进行排序。

像字典一样，列表也是对象，也有许多方法可以被调用。最常用的一个方法是append(element>,它用于向列表添加元素。类似地，insert(index, element)可以将一个项目插人到列表中指定的位置 count (element}方法可以告诉我们一个元素在列表中出现了多少次，而index (),如我们在前面例子中看到的，它可以告诉我们一个项目在列表中的索引.reverse (>方法实现的功能如其名：将一个列表反转 sort (>方法也同样顾名思义，它用来对列表进行排序。但仍有一些相当复杂的面向对象的行为，我们将在下面介绍。

对列表排序

不需要任何参数，sort就可以做到我们希望做的事情。如果这是一个字符串列表，那么就按照它们的字母顺序进行排序。该操作是大小写敏感的，所以所有的大写字母都会排在小写字母的前面，也就是说Z会排在a前面。如果是一个数字列表，那么它就会按照数字顺序进行排序。如果给出的是一个元组的列表，那么会按照每个元组中的第一个元素进行排序。如果给出的是混合多种不可排序的项目，那么sort就会抛出TypeError异常。

如果我们想要把自定义对象存人列表，并且使它们支持排序，那么我们还要多做一点T作。个特殊的方法，它代表“小于’在我们的对象中，需要定义它来使我们的类的实例支持做比较。列表中的sort方法将会对每个对象调用该方法来决定它应该处于列表中的哪个位置.这个方法将会使我们的类与传人参数的类相比较，某方面小时返回True,反之返回False。下面这个简单的类实现了基于字符串或数字进行排序。

```
class WeirdSortee:
    def __init__(self, string, number, sort_num):
        self.string = string
        self.number = number
        self.sort_num = sort_num

    def __lt__(self, object):
        if self.sort_num:
            return self.number < object.number
        return self.string < object.string

    def __repr__(self):
```

```
        return"{}:{}".format(self.string, self.number)
```

_repr_方法使我们在打印列表时可以轻松地看到两个值。这里的_1七_实现了比较一个类（或任何具有string、number和sort_num属性的鸭子类型对象.缺失任意一个属性将会失败）的两个实例，下面的输出表明了这个类在排序时的动作。

```
»> a = WeirdSortee (' a' , 4, True)
»> b = WeirdSortee ('b' , 3, True)
»> c = WeirdSortee ('c', 2, True)
»> d = WeirdSortee ('d', 1, True)
»> 1 = [a,b,c,d]
»〉 1
[a:4, b:3, c:2, d:1]
»> l.sort()
»〉 1
[d:l, c:2, b:3, a:4]
»> for i in 1:
... i.sort__num = False

»> l.sort()
»〉 1
[a:4, b:3, c:2, d:l]
```

当我们第1次调用sort,它会依数字进行排序，因为进行比较的对象的$_{SO}rt__{nU}m$为True。第2次时，它会依字母进行排序。_lt_方法是使其支持排序的唯一一个我们需要实现的方法。然而从技术上讲，如果_11:_被实现了，通常与其相似的_gt_、__eq 、 ne 、_ge_及 le 方法也都应该实现，只有这样<、>、==、！ =、>=、<=操作符才能正确丁作。

sort方法也可以接收一个可选的key作为参数。该参数是一个用于将列表中每个对象转换成一个可以进行比较的对象的函数。当我们想要对元组值基于其第2个项目而不是第1个项目（默认以第1个项目对元组排序）进行排序时，这非常有用。

```
>» x =[<1,' c'> , (2,' a') , (3, 'b >]
»> x.sort()
»> x
[<1, ' c->, (2, 'a",〈3,' b n
»> x.sort (key=lambda i: i[1])
```

```
»> x
[(2, 'a'), (3, *b')f (1, 'O]
```

命令行中的lambda关键字创造了一个以元组为输人，使用顺序查找返回索引为1的项目（这是元组中的第2个项目）的函数。

另举一例，我们也使用key参数使排序不区分大小写a为了实现这一目的，我们只需简单地比较全部字符串的小写版本，所以我们将内置函数str. lower作为key函数传入：

```
»> 1 = [″hello″, ″HELP″, ″Helo″]
»> l.sort()
»> 1
[•HELP、'Helo、'hello* ]
»> 1. sort (key=str. lower)
»> 1
['hello', 'Helo* f 'HELP']
```

记住，即使lower是字符串对象的一个方法，它也仍然是个可以接收一个单一参数self的闲数。也就是说，str. lower (item)和item, lower (>是等价的。当我们将这个闲数作为key传入时，它将会以小写值代替默认的大小写敏感值进行比较。

采口

列表是个多功能的丁具，它适合做大多数应用程序中的容器对象。但是当我们想要确保列表中的对象是独一无二的时候，它就不是那么有效了。例如，一个歌曲库可能保存着一位艺术家的很多首歌曲，如果我们想要对整库进行排序，并创建一个包含全部艺术家的列表，我们就需要检查这个列表来确认我们添加的艺术家是不是在之前已经添加过。

这就需要集合了。集合的概念源自数学，在数学中集合用来表示一组无序的（通常是）唯一数字。我们可以将一个数字加人集合5次，但是它只会在集合中出现一次。

Python中，集合可以容纳任何可哈希的对象，不仅仅是数字。可哈希对象与可用作字典键的对象是相同的，所以列表和字典都不属于这一类。如同数学中的集合，它们只会为每个对象复制一份进行存储。所以，如果我们试图创建一个歌曲艺术家列表，我们可以创建一个存储名字字符串的集合，然后简单地把所有艺术家的名字加人进去就可以了。下面这个例子以一个（歌曲、艺术家）元组的列表为开始，然后创建一个艺术家集合：

```
song_library = [("Phantom Of The Opera", "Sarah Brightman"),
```

```
        (•'Knocking On Heaven' s Door", "Guns N' Roses"),
        ("Captain Nemo", "Sarah Brightman'1),
        ("Patterns In The Ivy", "Opeth"),
        ("November Rain", "Guns N* Roses**),
        ("Beautiful", "Sarah Brightman"),
        ("Mai' s Song", "Vixy and Tony")]

artists = set()
for song, artist in song_library:
    artists.add(artist)

print (artists)
```

并没有像列表和字典那样的内置语法让我们创建一个空集合，所以我们只能使用set（>构造函数来达到这个目的。不过，我们可以用字典的花括号语法来创建一个集合，只要集合中包含值。对字典来说，我们用冒号将一对值分隔开，如厂key、• value \ Wey2、•▽311ˆ2'}。对集合来说，我们使用逗号来分隔值，如{'乂31此'，'乂311ˆ2'}。通过add方法，其他的项目可以独立地加人到集合中。如果我们运行这个脚本，我们将会看到这个集合输出如下：

```
{'Sarah Brightman', "Guns N' Roses", 'Vixy and Tony',        *Opeth*}
```

如果你留意输出的结果，你会看到集合中的项目并没有按照它们加人集合的顺序打印出来。这是因为集合和字典一样是无序的。它们同样为了效率使用了基于哈希的底层数据结构。因为是无序的，所以集合也不能通过索引进行查找。使用集合的主要目的是将全部事物分为两组：“属于集合的事物”和“不属于集合的事物”。检验一个项目是否属于集合以及遍历集合中的项目都是很简单的，但如果我们想要对它们进行排序，我们将不得不将集合转换为一个列表。下面的输出显示了全部3个操作：

```
»> "Opeth" in artists
True
»> for artist in artists:
... print (•'{} plays good music" . format (artist))

Sarah Brightman plays good music
Guns N' Roses plays good music
Vixy and Tony play good music
```

```
Opeth plays good music
»> alphabetical = list (artists)
»> alphabetical. sort ()
»> alphabetical
["Guns Nf Roses", 'Opeth', 'Sarah Brightman', 'Vixy and Tony']
```

虽然集合的主要功能是提供唯一性，但这并不是它的主要目的。当两个或两个以上的集合需要结合时，集合是最有用的。集合类型的大部分方法都可以操作另一个集合，这使得我们可以高效地组合或比较两个或多个集合中的项目。如果你对数学中的集合并不熟悉的话，这些方法的名字对你来说可能会比较奇怪，因为它们来自数学中的术语。我们将从3个无论谁是调用方法的集合或被操作的集合都会返回相同结果的方法开始讲起。

union方法最为常见，也容易理解。它将第2个集合作为参数，并返回一个包含全部存在于两个集合中任意一个的元素的新集合。如果一个元素同时存在于两个原始集合，在新集合中当然它只会出现一次。并集操作就像逻辑运算or,也就是说，如果你不调用这个方法，、、I"操作符可以达成同样的目的。

相反，intersection方法接收第2个集合为参数，但返回的是仅包含同时存在于两个集合的那些元素。它就像逻辑元素and,也可以使用&操作符引用。

最后，symmetric_difference方法告诉我们哪些是剩下的元素，它是一个存储于或是存在于这个集合或是另一个集合，但不是同时存在于两个集合的对象的集合。下面的例子通过比较我的歌丨111库和我姐姐的歌曲库，演示了这些方法的作用：

```
my_artists = {"Sarah Brightman", "Guns N* Roses",
              "Opeth", "Vixy and Tony"}

auburns—artists = {"Nickelback", "Guns N' Roses",
              "Savage Garden"}

print("All: {}".format(my_artists.union(auburns_artists)))
print("Both: {}".format(auburns_artists.intersection(my_artists)))
print("Either but not both: {}".format(
      my_artists.symmetric_difference(auburns_artists)))
```

如果运行这段代码，我们可以看到这3个方法给出了打印语句中它们的执行结果：

```
All:  {'Sarah Brightman', "Guns N' Roses", 'Vixy and Tony',
'Savage Garden', 'Opeth'f 'Nickelback'}
```

```
Both: {"Guns N' Roses"}
Either but not both: {f Savage Garden', 'Opeth*, 'Nickelback',
1 Sarah Brightman', 'Vixy and Tony'}
```

无论是哪个集合调用另一个集合，这些方法都会返回相同的结果。我们既可以用my_artists . union (auburns_artists)，也可以用 auburns_artists . union (m_y_$artist_S$),它们都会得到相同的结果。也有一些方法，它们的结果取决于谁是调用者，谁是参数,>

这些方法包括互为逆操作的issubset和issuperset,它们都会返回一个布尔值。当调用方法的集合中的全部元素都存在于传入的参数集合时，issubset返回True。当传人的参数集合中的全部元素都存在于调用方法的集合时，issuperset返回True。所以，s . issubset (t)和t • issuperset (s >是完全一样的，.当t中包含s中的全部兀索时，它们都将返回True。

最后，difference方法返回全部只存在于调用方法的集合而不存在于传人参数的集合的兀素，它就好像symmetric_difference的一半。差集方法可以同样用操作符表示。下面的代码演示了这些方法是如何使用的：

```
my—artists = {"Sarah Brightman", "Guns N' Roses",
            "Opeth", "Vixy and Tony"}


bands = {"Guns N' Roses", "Opeth"}


print(umy_artists is to bands:"}
print("issuperset: {}n.format(my_artists.issuperset(bands)))
print ("issubset: { } '*. format (my_artists . issubset (bands)))
print ("difference: { } *'. format (my_artists . difference (bands)))
print("*"*20>
print("bands is to my-artists:")
print("issuperset: {}".format(bands.issuperset(my_artists)))
print("issubset: {}".format(bands.issubset(my_artists)))
print ("difference: {} *'. format (bands. difference (my_artists)))
```

这段代码只是简单地打印出了当从一个集合调用另一个集合时每个方法的执行结果，运行这段代码将得到如下输出：

```
my -artists is to bands:
```

```
issuperset: True
issubset: False
difference: {•Sarah Brightman•,      'Vixy and Tony'}
********************
bands is to my-artists:
issuperset: False
issubset: True
difference: set()
```

在第2个例子中的difference方法返回了一个空集合，这是因为没有项目存在于bands，但不存在于 my_artists。

union、intersection和difference方法可以接收多个集合作为参数，如我们所料，它们将返回一个在所有参数上执行所调用操作后的集合。

所以，集合中的方法清楚地表明，集合可以用来操作另一个集合，它们不仅仅只是个容器。如果我们有两处不同来源的数据，并想快速地以某种方法将它们结合起来，来查看数据是否重复、不同处何在，那么，我们可以使用集合操作来高效地比较它们。或者，如果我们收到的数据有可能含有已经处理过的重复数据时，我们可以使用集合来比较二者，并只处理新数据。

扩展内置数据类型

我们在第3章中简要讨论了如何通过继承扩展内置数据类型。现在，当我们再次讨论这个话题时，让我们来看看更细节的东西。

假如我们想对一个内置容器对象添加功能，有两个方法可供选择[3]其一，我们可以创建一个新的对象，让容器成为这个对象的一个属性（组合）；或者，我们可以成为内置对象的子类，添加或调整它的方法使之完成我们想要做的工作（继承）。

如果我们要做的只是用容器的一些功能来存储一些对象，那么组合通常是最好的选择。用这种方式可以很容易地做到将数据结构作为参数传递给其他方法，并且它们也能知道如何与其交互。如果我们想要改变容器原有的工作方法，那就需要继承了。例如，我们希望确保在一个list中的每个项目都是一个5字符的字符串，我们就需要扩展list，重写append()方法使其在收到非法输入时抛出一个异常。我们还需要重写 _setitem____(self, index, value)，这个是列表中每当我们用 x[index]=、、value”语法时调用的特殊方法

没错，我们看到的所有这些看起来特殊的非面向对象的语法，如访问列表、字典键、遍历容器，以及其他相似的工作，实际上都是“语法糖”，在底层它们都会映射到面向对象的范例上。我们可能会问，Python的设计者们为什么要这样做，尤其是在大家都已经达成了面向对象编程总是更好的这个共之后。这个问题很容易回答。在下面这个假想的例子中，对程序员来说哪个更易读，哪个需要的输人更少？

```
c = a+ b
c = a.add(b)

l[○] = 5
1. setitexn(0, 5)
d[key] = value
d.setitem(key, value)

for x in alist:
    #Mx进行某些操作
it = alist.iterator()
while it.has一 next():
    x = it. next ()
    #对31进行某些操作
```

高亮部分显示了面向对象版本的代码可能会是什么样子（在实践中，这几个方法实际上会以与对象关联的特殊的双下画线方法的形式存在）。Python程序员同意非面向对象的语法更易于读写，而非Python程序员会说这些语法令Python看起来并非面向对象，但这毫无意义。所有上述的Python语法都会在表面之下映射为一个面向对象的方法。这些方法有着特殊的名称（在其前后都有双下画线）来提醒我们这里会有种更好的语法。然而，我们现在有重写这些行为的手段了。例如，我们可以让一个特殊的整数在两两相加时总是返回0：

```
class Sillylnt(int):
    def __add__(self, num):
        return 0
```

这无疑是一个非常奇怪的举动，但是它恰当地说明了面向对象原则的原理。现在，当人们告诉我们Python不是真的面向对象的时候，我们可以据此力争了 ：正是因为它是面向对象的，所以才使得Python易于使用。让我们看看上面的类是如何工作的：

```
»>  a  =  Sillylnt(l)
»>  b  =  Sillylnt(2)
»> a + b
```

有关_add_方法的一个可怕的事实是，可以将它添加到任何一个我们写的类上，而a每当我们对一个类的实例使用“+”操作符，它就会被调用。这就是字符串、元组和列表连接如何工作的。

对于所有特殊方法这都是真的。如果我们想要使用x in myobj语法，我们可以重写_contains_。如果我们想要使用myobj [i]= value语法，我们可以实现—setitem____。如果我们想要使用something = myobj [i],我们就实现—getitem___。

在list类中，这样的特殊方法共有33个。我们可以用dir函数来查看它们：

```
»> dir (list)

[•__add__* , '__class__• , ¹__contains___' , '_delattr___' , 'delitem ' • •_
doc__• , ■_eq(_¹ ,  '__format_•  ,  '_ge__ᶠ ,  ¹_____ getattribute___*,•getitem⁻
⁻• ,  •__gt__' ,  •_____hash, , ¹ iadd ¹ •• imul_____' ,  '_____init___*,'iter ',
'__le__' ,  ' len___*,'___It ' '  mul__' ,  '_____ne_____* ,  *_____new____',
 •,  'reduce ex  ', '  repr '.• reversed  *,  *  rmul •,• setattr
* , * setitem ¹ , • sizeof_*, ' str_•, * subclasshook ' , ' append',
'count■, * extend•, 'index ' 'insert *, 'pop', 'remove¹, 'reverse', * sort•
```

更进一步，如果我们想要得到有关这些方法是如何工作的任何额外信息，我们可以使用help函数：

```
»> help (list.____add__)
Help on wrapper ⁻descriptor:

_add_(…}
x.__add <y) <=> x+y
```

列表中的加法操作符可以将两个列表连接。在本书中，我们没有笔墨来讨论全部这些可用的特殊函数，但是，你可以直接通过dir和help来浏览这些功能。官方的在线Python参考（http://docs. python. org/）也同样有大M的有用信息。应该特别注意的是，在集合模块中讨论的抽象基类。

回到我们前面提到的问题，什么时候我们该用组合，什么时候用继承：如果我们想要

以某种方式改变类中的任何方法，包括特殊方法，我们绝对需要使用继承。如果我们使用组合，那么我们可以写几个方法来实现验证或改变，并让调用者使用这些方法，但是我们并不能阻止它们直接去访问任何属性（没有私有成员，还记得吗？）。它们可以在我们的列表中插人一个没有5个字符的项目，这可能会使列表中的其他方法闲惑。

通常，如果需要扩展一个内置数据类型，可能意味着我们使用了错误的数据类型。虽然并非总是如此，但如果你突然想要扩展一个内置数据类型，一定仔细考虑清楚有没有其他不同的数据结构更为合适。

作为最后一个例子，让我们考虑如何创建一个能够记住键插人顺序的字典.一种方法（可能不是最好的方法）是在diet的派生子类中存储一个维护键的有序列表。之后，我们可以重写方法keys、values、_iter_和items来使一切恢复有序〇当然，我们还需要重写_36七1七6111_和361O^£3O11：，来更新我们的列表。为了保持列表和字典一致，可能还有一些在dir（diet)输出的其他方法需要重写（记住clear和_delitem_，用于跟踪项目被删除），但是在这个例子中我们不必担心它们。

因此我们将要扩展diet,添加一个键顺序的列表。确实足够简单，但实际上我们应该在哪创建这个列表呢？我们可以在_init_方法中添加，这可以很好地完成我们想要的丁.作，但是我们不能保证任何一个子类都调用这个初始化函数。还记得我们在第2章中讨论过的_new_*法吗？我说它通常只会在极特殊的情况下有用，现在就是其中一个特殊情况。我们知道_new_H会被调用一次，并且为新实例创建的列表对我们的类总是可用的。考虑到这一点，下面这个就是我们的排序字典的全部代码：

```
from collections import KeysView, ItemsView, ValuesView
class DictSorted(diet):
    def —new_(*args, **kwargs>:
        new_dict = diet.__new__(*args, **kwargs)
        new_dict.ordered-keys =[]
        return new_dict

    def —setitem—(self, key, value):
        '11 self[key] = value syntax'1'
        if key not in self.ordered-keys:
            self•ordered-keys•append(key)
        super().__setitem—(key, value)

    def setdefault(self, key, value):
```

```
        if key not in self•ordered—keys:
              self.ordered_keys.append(key)
         return super(>•setdefault(key, value)

    def keys(self):
        return KeysView(self)

    def values(self):
        return ValuesView(self)

    def items(self):
        return ItemsView(self)

    def ____iter____(self):
        '''for x in self syntax''1
        return self•ordered—keys•____________iter____()
```

new_*法简单地创建了一个新字典，然后把一个空列表放人该对象。我们不必重写:^:11:_,因为默认的实现就可以工作（实际上，这只在我们初始化一个空的DictSorted对象时才是对的，这是标准行为。如果想要支持diet构造函数的其他变种，如接收字典或元组列表，我们也需要同时修改_^^_来更新我们的orderec^keys >。设置项目的两个方法是非常相似的，它们都会更新键的列表，但只是对之前没有添加过的项目有效。我们并不希望在列表中出现重复项，但我们在这里并不能使用集合，因为集合是无序的！

keys、items和values方法会返回字典的某种视_图。集合库为字典提供了 3个只读的View对象，可以使用_iter_方法来遍历键，之后用$_{9}$^:1^11$_{1}$_(我们没有重写它）来获取值。所以我们只需要自定义我们的_iter_方法，就可以使这3个视图重新工作起来。你会觉得超类应该能够使用多态正确地创建这些视图，但是如果我们没有重写这3个方法，它们就不会返回正确的有序视图。

最后，真正特殊的方法。它确保如果我们遍历字典的键（使用f$_{O}$r...in语法），可以以正确的顺序返回值。可以简单地通过返回ordered_keys来实现它，ordered_keys的_iter_会返回我们对这个列表使用for...in返回的相同迭代器对象。因为Ordered_keys是所有可用键的列表（由于我们重写其他方法的方式），所以这个迭代器对象对于字典来说也是正确的。

让我们来看看对比标准的字典，这些方法是怎样工作的：

```
»> ds = DictSorted()
»> d = {}
»> ds [ ’ a ’  =1
»>   ds['bF]   =2
»> ds . setdefault (' c' , 3)
3
»>   d['a   =1
»>  d['b']  =  2
»> d. setdefault ('c' , 3)
3
»> for k, v in ds . iterns ():
...     print(k,v)

 a 1
b 2
c 3
 »> for k, v in d. items ():
...      print(k,v)


c 3
b 2
```

哦！我们的字典是有序的，而标准的字典不是。万岁！

/ 如果你想在生产环境中使用这个类，你还必须重写一些其他方法来确保对于所有例子来说键都是最新的。、但你不必这样做。因为这个类的功能在Python中已经提供了，使用collections模块中的OrderedDict对象就可以，，试试从collections中弓1入这个类，然后用help （OrderedDict）来查看它的详细信息。

案例学习

为了将我们上面介绍过的全部东西都放在一起，我们将编写一个简单的链接收集器，

它可以访问网站并收集在这个站点上从每个页面上找到的每个链接。在我们开始之前，还谣要准备一些测试数据，简单地写几个包含互相链接及到Internet的其他站点的链接的HTML文件，如下：

```
<html>
    <body>
        <a href="contact.htn\l">Contact us</a>
        <a href="blog.html">Blog</a>
        <a href="esme.html">My Dog</a>
        <a href=",,/hobbies .html">Some hobbies</a>
        <a href="/contact.html">Contact AGAIN</a>
        <a href="http: //www. archlinux. org/'f">Favorite OS</a>
    </body>
</html>
```

将其中一个文件命名为index. html，使它成为站点被访问时第一个出现的网页。确保其他的文件都是存在的，并使页面之间包含大量的链接以确保问题的复杂性。如果你没有自己做什么修改，本章的例子中包含一个名叫case_study_serve的字典（实在是最烂个人网站的一员！）

现在，通过进人包含所有文件的目录，并运行以下命令中来启动一个简易的Web服务器：

```
python3 -m http.server
```

这条命令会在8000端口上运行我们的服务器，你可以在浏览器上通过访问http://localhost: 8000/来查看这些页面。

> 我对任何认为能用更少的工作来构建起一个网站的说法都表示怀疑，绝不要被别人说：“用Python你不可能那么简单地做到。”

将站点基URL（本例中就是http://localhost:8000/)传递给我们的收集器，我们的目标就是创建一个列表保存这个站点中每个唯一的链接。我们需要考虑3种不同类型的URL（以http://开头的表示链接至外部站点，以/开头的表示绝对内部链接，以及表示相对链接的其他类型）。我们还需要注意页面上的链接可能在一个循环中，我们需要确保我们不会对同一个页面进行多次处理，否则它可能永远不会结束。为了满足上述需求，

看起来我们需要一些集合了。

在深人之前，让我们先从最基本的开始。我们需要什么样的代码使我们连接到一个页面并且解析该页面上的所有链接？

```
from urllib.request import urlopen
from urllib.parse import urlparse
import re
import sys
LINK—REGEX = re.compile(
        w<a [A>]*href=['\H] ([A r\M]+) [1 \M] [yx>]*>n)

class LinkCollector:
    def __init__(self, url):
        self.url = "http://" + urlparse(url).netloc

    def collect_links(self, path="/"〉:
        full__url = self. url + path
        page = str(urlopen(full_url).read())
        links = LINK_REGEX.findall(page)
        print(links)

if  name_ == "_main ’’ :
    LinkCollector(sys.argv[1]).collect_links()
```

这是一段简短的代码，让我们看看它实现了什么。通过由命令行传人的参数，它与服务器建立一个连接并下载页面，之后提取页面上的全部链接。法使用urlparse函数从URL中解析出主机名，所以即便我们传入http://localhost:8000/some/page.html，它也能对顶层的主机进行操作，即http://localhost:8000/。这是有意义的，因为我们只是想收集这一站点的全部链接。

cOll$_{eC}$t_link$_S$方法用来连接服务器并下载指定的页面，它使用正则表达式来找出页面上的全部链接。正则表达式是一款非常强大的字符串处理T.具。不过麻烦的是，它的学习曲线非常陡。如果你之前从来没有用过它，我强烈建议你从相关的书或网站t学习一下这个专题。如果你认为它们不值得了解，那么请你试试不用正则表达式来改写上述代码$_3$

上面的例子停在collect_links方法的中间，打印出了 links的值。这是测试程序中的一个非常普遍的做法：停下并输出值以确保值是我们所预期的。下面就是我们例子

中的输出：

```
[* contact.html', 'blog.html', 'esme.html', '/hobbies.html',
'/contact.html*, 'http://www.archlinux.org/']
```

那么现在，我们已经有了一个收集了第一个页面全部链接的集合。我们怎样对它进行操作呢？我们不能把所有链接都弹出来放人一个集合中做去重，因为链接的地址有可能是相对的，也有可能是绝对的。例如，contact.html和/contact.html所指向的就是同一个页面[3]所以，我们要做的第一件事应该是把所有链接都规范化为包含主机名和相对路径的完整路径。我们可以通过在对象中添加一个normali_Ze—Url方法来实现它：

```
def normalize_url(self, path, link):
    if link. startswith ('*http://'*):
        return link
    elif link.startswith (''/M):
        return self.url + link
    else:
        return self.url + path.rpartition('/'
                )[0] + '/* + link
```

这个方法能够将所有URL都转换为包含协议和主机名的完整URU现在，这两个连接页面就有了相同的值，所以我们可以把它们储存到一个集合中。我们还需要修改法来创建这个集合，修改cOll_ect_links来把所有的链接都放人到这个集合中。

然后，我们还要访问所有的非外部链接并对它们进行收集。但是等一下，如果这样做了，我们怎样才能避免对同一页面访问两次呢？看起来我们真的需要两个集合，一个用来存储收集到的链接，另一个存储访问过的链接。这表明选择集合来代表我们的数据是明智的，因为我们知道当需要操作多于一个数据时，集合是最有用的。让我们来做这些设置：

```
class LinkCollector:
    def -init-(self, url):
        self, url = "http ://+*' + urlparse (url) . netloc
        self.collected -links = set()
        self.visited -links = set(>

    def collect—links(self, path="/"):
        full__url = self .url + path
```

```
            self.visited_links.add(full_url)
            page = str(urlopen(full_url).read())
            links = LINK_REGEX.findall(page)
            links = {self.normalize_url(path, link
                ) for link in links}
            self.collected_links = links.union(
                    self.collected_links)
            unvisited_links = links.difference(
                    self.visited_links)
            print(links, self.visited_links,
                  self.collected_links, unvisited_links)
```

用于创建规范化链接列表的代码行使用了集合解析，这与列表解析没什么不同，除了它创建了一个集合来存储值。我们将会在下一章详细介绍这些新的概念。再一次，程序停下来并打印当前值，使我们可以验证我们的集合没有混乱，并且difference真的是我们为了收集unvisited_links所要调用的方法。然后我们就可以添加几行代码来遍历所有的未访问链接，并把它们加入到集合中：

```
        for link in unvisited_links:
            if link.startswith(self.url):
                self.collect_links(urlparse(link).path)
```

if语句确保了只从一个网站收集链接，我们并不想去访问和收集所有来自互联网页面的链接（除非我们是谷歌或者Internet Archive!）。如果在程序最下面修改主要代码用于输出全部收集到的链接，我们将会看到它似乎是收集到了我们想要的全部：

```
if __name__ == "__main__":
    collector = LinkCollector(sys.argv[1])
    collector.collect_links()
    for link in collector.collected_links:
        print(link)
```

这里显示了我们收集到的全部链接，并且每个只有一次，尽管在例子中很多页面都互相链接了多次：

```
$ python3 link_collector.py http://localhost:8000
http://localhost:8000/
http://en.Wikipedia.org/wiki/Cavalier_King_Charles_Spaniel
```

```
http://masterhelenwu.com
http://archlinux.me/dusty/
http://localhost:8000/blog.html
http://ccphillips.net/
http://localhost:8000/contact.html
http://localhost:8000/taichi.html
http://www.archlinux.org/
http://localhost:8000/esme.html
http://localhost:8000/hobbies.html
```

尽管收集到了至外部贞面的链接，但它并不会从任何一个我们链接的外部页面访问和收集。如果想要收集一个站点的全部链接，这的确是一个非常棒的小程序。但是，如果我想构逮一个站点地图，它并不能给出全部所需的信息，它能告诉我有哪些页面，但是却不能告诉我哪个页而链接到哪个页面。如果我们想要实现这个功能，我们将不得不再做出一些修改。

我们应该做的第一件事就是看看选用的数据结构。用于储存收集到链接的集合不再有效了，我们现在想知道某个链接是从哪个页面链接到的。那么，我们可以做的第一件事就是将集合转换成存储集合的字典，用以保存我们访问过的每个页面。字典的键代表与目前集合中完全相N的数据，而值将是这个页面h的全部链接的集合。下面就是这些改变：

```
from urllib.request import urlopen
from urllib.parse import urlparse
import re
import sys
LINK_REGEX = re.compile(
        "<a [^>] *href=['\"] ([^'\"]+) ['\"] [^>]*>")

class LinkCollector:
    def __init__(self, url):
        self.url = "http://%s" % urlparse(url).netloc
        self.collected_links = {}
        self.visited_links = set()

    def collect_links(self, path="/"):
        full_url = self.url + path
        self.visited_links.add(full_url)
```

```
        page = str(urlopen(full_url).read())
        links = LINK_REGEX.findall(page)
        links = {self. normalize__url (path, link
             ) f o r  link in links}
        self•collected-links[full- url =links
        for link in links:
             self.collected -links.setdefault(link, set())
        unvisited_links = links.difference(
                  self.visited_links)
        for link in unvisited_links:
             if link.startswith(self.url):
                       self.collect_links(urlparse(link).path)


def normalize_url(self, path, link):
        if link, startswith (·fhttp://″):
             return link
        elif link• startswith (•'/'•):
             return self.url + link
        else:
             return self.url + path.rpartition('/1
                       )[0] + '/' + link


        if _name_ == ″_main_″:
             collector = LinkCollector(sys.argv[1])
             collector.collect_links()
             for link, item in collector.collected -links•items():
                  print (″(→: { } ″ . format (link, item))
```

这是一个惊人的小改动，原来创建两个集合联合体的代码行被3行更新字典的代码所取代。这个小改动首先告诉了我们字典中这个页面收集到了哪些链接。其次，为每个还没有被添加到字典中而又应该在这个字典中的项目，使用setdefault创建了一个空的集合。作为结果，这个字典以所苻的链接作为键，每个内部链接键映射到一个链接的集合，每个外部链接键映射到一个空集合。

练习

学习如何选择正确的数据结构的最好方法就是先错上几回。找出一些你最近使用列表写的代码，或者用列表写些新的代码，然后试着用其他的数据结构来重写它。哪个数据结构更有意义？哪个没有意义？哪个是最简洁的代码？

试着用不同的数据结构对。你可以看看在上一章练习中完成的代码。有没有哪些有方法的对象可以用namedtuple或者diet去代替？都试试看。有没有字典能够转换成集合，因为你并不去访问它的值。你有没有为列表检查过重复项？用集合是不是能满足要求，亦或是多个集合？

如果你想要一些具体的例子作为练习，试着将链接收集器改成还能保存每个链接的标题。也许你可以用HTML来生成一个站点地图，列出一个站点上的全部页面，同时包含一个指向其他页面的链接列表，并用相同的链接标题为其命名。

你最近有没有写过任何容器对象，可不可以通过继承内置类型或重写某些“特殊”的双下_线方法来改进它？你可能需要做一些研究（使用dir和help，或者用Python库参考）来找出哪些方法需要被重写。你能确定使用继承是正确的吗？使用基于组合的方案会不会更有效？在做出决定前，请把两种方法都试试（如果可能的话），试着找出对于不同的情景哪个方法会更好u

如果你在开始本章之前就熟悉各种Python数据结构及它们的使用方法，可能你已经觉得厌倦了。但如果是这样的话，很有可能你使用的数据结构太多了。看看你以前写过的代码，用更多的自定义对象去重写它。仔细考虑这些替代方案并进行尝试，看看哪个才是最易读且易于维护的系统。

对你的代码和设计决定总要做仔细的评估。养成回顾旧代码的习惯，从你写了它，如果你对“好的设计”的理解有了变化，就应该记下来。美学是软件设计中一个庞大的组成部分，就如同_家一样，我们都必须找出最适合我们的风格。

总结

我们已经介绍了一些内置数据类型，并试着理解怎样为特定的应用做出合适的选择。有时候，最好的办法就是我们创建一个对象的新类，但通常情况下. 某个内置数据类型就能满足我们的需要。当它真的不能满足要求时，我们可以使用继承或者组合来使它们适应我们的需求。我们特别介绍了：

- 元组和命名元组。

- 字典和默认字典$_3$
- 列表和集合。
- 重写内置类型的特殊变fl。

在下一章中，我们将讨论如何将Python中的面向对象与非面向对象两方面结合起来。在这个过程中，我们将会发现比看上去多得多的面向对象设计！

7

第7章Python里面向对象的快捷方式

现在，让我们来看看Python中令人更多联想到结构式或函数式编程而不是面向对象编程的几个方面虽然面向对象编程目前在市面上最受欢迎，但是旧的范式仍然提供了有用的丨:具。实际上这些T具多数是语法糖，底层还是用面向对象来实现的。我们可以把它们看作是建立在（已经抽象的）面向对象的范式之上的更深层的抽象层。本章我们将会讨论：

- 一次调用就能处理一般工作的内置函数:
- 列表、集合和字典解析。
- 生成器。
- 方法重载的另一种选择。
- 函数就是对象

Python内置函数

Python中有很多函数可以在某些对象上执行一个任务或者计算一个结果，而无须成为一个类中的方法。这些函数的作用是对常用的计算进行抽象，进而适用于多种类型的类。这被称为鸭子类型 duck typing);这些函数可以接收具有某些属性的对象或者满足某个给定接口的方法，而且能够对这些对象执行通用的任务。

Len

最简单的例子就是函数len(>。这个函数用来计算一些容器对象中项目的个数，如字典或者列表。例如：

```
»> len([1,2,3,4])
4
```

为什么这些对象不设置一个表示长度的属性，而是去调用一个函数呢？从技术上讲，它们是有的。可以应用len()的多数对象都自带一个同样能够返回相同值的_len_<)方法，所以len (myobj)似乎就是调用myobj ._len_ (〉。

为什么我们不用这个方法而使用函数len()呢？显然，这些带有双下画线的特殊方法是不能直接调用的，必须要对此进行解释。Python的开发者不会草率地做出这样的设计决定。

更主要的原因是效率。当我们对一个对象调用_161!_时，该对象必须在自己的命名空间中查找该方法，并且，如果特殊的_getattribute_方法（每次一个对象的属性或方法被访问时都要调用这个函数）也在该对象中定义了，那么它也会被调用。甚至，用于特定方法的_getattribute_还有可能用来去做一些很讨厌的事情，比如拒绝我们访问一些特殊函数，就像_len_!而len函数则不会遇到这种情况，它就是实际去调用底层类中的—len_函数，所以 len (myobj 〉对应为 MyObj • _len_(myobj 〉。

另一个原因是可维护性。未来，Python开发者们可能会想修改len (),使它能够计算出没有_len_的对象的长度，例如通过对一个迭代器返回的项目数量计数得到长度值。这时，只需要修改一个函数，而不需要修改无数t_len_方法。

Reversed

reversed (〉函数的输人是任意一个序列，返回一份倒序后的序列副本。它通常用于for循环需要倒序循环时。

与len相同，反转函数reversed调用参数类中的_reversed_(〉函数。如果该方法不存在，reversed会通过调用_len___和_getitem_生成倒序的序列。如果我们需要定制或者优化这个过程，只需要重写_rev$_e$rsed_即可：

```
normal_list=[1,2,3,4,5]

class CustomSequence ():
    def __len__(self):
        return 5

    def __getitem (self, index):
        return Mx{0}M.format(index)
```

```
class FunkyBackwards(CustomSequence):
    def __reversed__(self):
        return "BACKWARDS!
for seq in normal_list, CustomSequence(), FunkyBackwards():
    print("\n{}: ".format(seq.__class__.__name__), end="")
    for item in reversed(seq):
        print (item, end=",
```

最后的for循环打印出了倒序后的正常列表和两个自定义序列的实例。输出结果显示倒序对3个序列都起了作用，但是当我们重新定义_1：#ˆ$_3$6〇1_后输出结果就大为不同了：

```
list: 5, 4, 3, 2, I,
CustomSequence: x4, x3, x2, xl, xO,
FunkyBackwards: B, A, C, K_f W, A, R, D, S, !,
```

> 注意：上述两个类都不是很好的序列，它们都没有定义恰当版本的_让61：_，所以对它们做正向的for循环将会陷入死循环。

Enumerate

有时，当我们用for循环对某个迭代对象遍历时，会想要访问当前被处理的项目的索引（即它在列表中的当前位置for循环不提供索引值，但是穷举函数enumerate可以给我们些更好的东西：它可以创建一个元组列表，每个元组的第1个对象就是索引，第2个是原始条目内容。

这是非常有用的，特别是当我们需要直接使用索引数值时。考虑一段简单的代码，在输出一个文件所有行时带着行号：

```
import sys
filename = sys.argv[1]

with open(filename) as file:
    for index, line in enumerate(file):
        print ("{0}: {1 }•'. format (index+1, line) f end
```

运行这段代码，并以它自己作为输人文件，我们可以看到它是如何T.作的：

```
1: import sys
2: filename = sys.argv[1]
3:
4: with open(filename) as file:
5: for index, line in enumerate(file):
6: print<"{0}: {1}"•format(index+1, line), end=1')
```

enumerate函数返回一个元组列表，我们的for循环将每个兀组分成两个值，print语句将它们一起进行格式化。由于enumerate函数像所有的序列一样，都是基于0开始的，所以耑要将每个索引值加〗成为行号。

Zip

Zip函数是Python收藏品中最不面向对象的一个函数。它将两个或以上的序列创建为一个新的元组序列。任意一个元组都包含每个列表中的一个元素

让我们通过解析文本文件的例子来简单解释这个函数。文本数据通常用制表位分隔的格式保存，文件的第1行作为“头”，其他每一行都是一条描述数据的唯一记录。一个以制表位为分隔符的简单通信录列表如下：

```
first       last        email
john        smith       jsmith@example.com
jane        doan        janed@example.com
david       neilson           dn@example.com
```

实现对这个文件的一个简单的解析器，可以通过使用zip来创建元组列表，将表头与数值映射起来。这些列表可以用来创建一个字典，一个在Python中用起来比文件简单得多的对象！

```
import sys
filename = sys.argv[1]
contacts =[]
with open(filename) as file:
    header = file.readline().strip().split(*\t')
    for line in file:
        line = line•strip().split(*\t*)
        contact map = zip(header, line)
        contacts.append(diet(contact-map)}
```

```
    for contact in contacts:
        print("email: {email} -- {last}, {first}"•format(
              contact))
```

这里到底发生了什么？首先我们打开由命令行提供的文件名对应的文件，读取它的第1行。然后去掉尾部的换行符，并且将剩余内容分割成一个有3个元素的列表。我们将• \t •传人strip方法中来表明该字符串应该被制表位分割。作为结果，头列表应该像[″first″, ″last″, ″email″】这样。

接着，对文件中剩下的行进行遍历（在头之后的部分）。我们将每行分割成3个元素。使用zip来为每一行创建一个元组序列。第一个序列是[(″first″, ″john″) , (″last″, ″smith″), (″email″, Mjsmith@example•com″)]〇

注意Zip做了什么。第1个列表保存了头，第2个保存了值。Zip函数为每一个P[J己项创建了一个包含头/值对的元组。

diet构造函数接收一个元组列表，然后将第]个元素作为键，第2个作为值来创逮一个字典，再将结果追加到列表中。

到这里，我们就可以自由地使用这个字典做所有与通信录相关的操作了。为了测试，我们简单地对通信录进行遍历來以不同的格式输出它们。像通常一样，格式化的行带有变量参数和关键字参数，**$_{CO}$nta$_{C}$t的使用能够A动将字典转换成一些关键字参数（我们会在本章结束前理解这个语法），下面是输出结果：

```
email: jsmith@example.com -- smith, john
email: janed@example. coin ——doan, jane
email: dn@example.com -- neilson, david
```

如果我们提供给Zip的是不同长度的列表，它会在最短的列表结束时停止。这个特性没有太多有用的应用，但是在这种情况下zip不会抛出异常。我们可以总是检查列表的长度. 并在需要的时候在短列表中加人空值。

Zip函数是它自身的反闲数^它可以将多个序列合成一个单一的元组序列。因为元组也是序列，所以我们可以对一个经过Zip的元组列表再次进行zip实现对它的“unzip”。哈，看看下面这个例子：

```
>» list—one = ['a¹, 'b' , * c']
»> list-two = [1, 2, 3]
»> zipped = zip (list-one, list—two)
»> zipped = list (zipped)
```

```
»> zipped
[('a', 1), Cb'f 2), (*。', 3)]
»> unzipped = zip (*zipped)
»> list (unzipped)
[(•a . . b . . c.>, (1, 2, 3)]
```

首先，我们将两个列表通过Zip转换成一个新的元组列表。之后，我们可以使用参数拆分将单独的序列作为参数传递给Zip函数。Zip将每一个元组的第1个值作为第1个序列，第2个值作为第2个序列，结果就是最开始的两个序列！

其他函数

另一个重要的函数是sorted(),它以一个迭代器作为输人，返回一个排过序的项目列表。它和列表中的sort ()方法很相似，区别在于它不仅适用于列表，也适用于所有的迭代器,,

像list .sort sorted 一样，它可以接收一个参数key来允许我们提供一个函数返回每个输人排序后的值同时，它还可以接收一个reverse参数。

另外3个函数可以在序列上应用的方法是min、max和sum。它们都以一个序列作为输人，然后返回最小值、最大值或序列中所有值的和。当然，只有当序列中所有值为数字时，sum才会正确运行。max和min方法与sorted和list. sort使用相同的比较机制，同时它们也允许我们定义一个相似的key函数。例如，以下代码使用enumerate max和min来返回一个列表中最大值和最小值的索引：

```
def min_max_indexes(seq):
    minimum = min(enumerate(seq), key=lambda s: s[1])
    maximum = max(enumerate(seq)/ key=lambda s: s[1])
    return minimum[0], maximum[0]
```

调用enumerate将序列转换为（index, time)元组。lambda函数作为key被传人是为了告诉函数去搜索每一个元组中的第2个项目（即原始项目）。然后变量minimum和maximum被设为enumerate返回的对应元组。return语句取得每个元组的第1个值（枚举的索引），然后将它们返回。以下交互会话显示了返回值的样子，即最小值和最大值的索引号：

```
»> alist = [5,0,1,4,6,3]
»> min_max_indexes (alist)
```

```
(1, 4)
>>> alist[1], alist[4]
(O, 6)
```

我们只接触了 Python内置函数中一小部分比较重要的函数。标准库中还有很多其他的方法，包括：

- all和any，它们接收一个迭代器，如果所有或任意一个项目被视为真（非空字符串、列表、非0数字、非None对象或文本True)时会返回True。
- eval exec和compile,它们在解释器中可将字符串当作代码执行。
- hasattr getattr setattr和delattr,它们允许对一个对象的属性通过其字符串名字来操作。
- 还有很多！每个函数都可以通过dir(_builtins_>列出的解释器帮助文档查看到。

解析

我们已经见过很多Python的for循环了。它允许我们遍历任何支持迭代协议的对象，间时可以依次对每个遍历到的元素进行特定的操作。

支持迭代协议仅仅意味着该对象拥有一个_iter_方法，该方法可以返回另外一个支持迭代协议的对象。支持迭代协议是对它拥有一t_next_方法的一种奇特的表达方式，这个方法可以返回序列的下一个对象，或者当所有对象都被返回时抛出Stoplteration 异常[3]

如你所见，虽然for语句看起来完全不面向对象，但实际上却是一些极其面向对象的设计的快捷方式。记住这一点会有助于我们讨论解析，因为解析看起来也完全不像是面向对象的T. 具，但它们也使用了与for循环相似的迭代协议[3]它们只是另一种快捷方式

列表解析

列表解析是Python最强大的工具之一，所以很多人认为它们很高级。其实不然。实际上，我已经檀自在之前的例子中丢出了解析的内容，而且现在假设你已经理解了。虽然确实有很多高级程序员们经常使用解析，但并不是因为它们很高级，恰恰相反是因为它们很平常，可以在程序中处理那些最平常的操作。

让我们来看一个这样的操作，即将一个某些项目的列表转换成另一个相关项目的列表。

特别说明的是，让我们假设，我们刚从一个文件里读出一个字符串列表，并想要把它转换成一个整数列表；我们知道列表中每一个项目都是一个整数，然后要对这些数字做一些操作（如计算平均值）。下面是一个简单的实现方法：

```
input_strings = [.I、, 5、’28、'131, ,

output_integers =[]
for num in input_strings:
    output_integers.append(int(num)>
```

看起来它T作得很好，只需要3行代码，如果你还不熟悉解析，也许不会觉得这看起来很丑陋！现在，让我看看使用列表解析来实现的相同代码：

```
input_strings = ['1', *5' ,■ 28 ' ,     •131’, , 3_]

output_integers = [int(num) for num in input_strings]
```

我们将代码缩减到丫一行，并且去掉了 append方法的调用。总之，它很容易就能讲明白正在做什么，虽然你可能不习惯解析的语法。

方括号表示的是在创建一个列表。在列表内部是一个for循环，对输人序列中的每一个项目进行了遍历。唯一让人疑惑的是，在左方括号和for循环开始之间发生了什么。但不管发生了什么，都是作用于输人列表中的每一个项目。这些产生疑问的项目是for循环中通过变量num引用的所以这就将每一个项目都转换成了 int。

这就是一个基本的列表解析。它们确实不高级吧！解析已经经过高度优化；当遍历列表中数量巨大的项目时，列表解析要比for循环快得多。如果说可读性是不能尽可能多地使用它们的有说服力的理由，那么速度一定是。

将一个某些项目的列表转换成另一个相关项H的列表并不是列表解析唯一可做的事情。我们可以通过在解析里添加一个if语句来实现选择并排除某些项目的目的。例如：

```
output_ints = [int(n) for n in input_strings if len(n) < 3]
```

这里将变M num简写成n，将结果变量记作output_ints，这样还可以只用一行代码完成。除了这个，这个例子与前一个例子的区别就是if len(n〉〈 3的部分。新加的代码能够将字符个数超过2的字符串排除掉。if语句是在int函数之前使用的，所以它检测了字符串的长度。由于我们输人的字符串本质上都是整数，所以实际上它排除的是所有超过99的数字。这就是列表解析的全部！我们用它们来映射输人值到输出值上，顺便过

滤掉任何不满足特定条件的值。

任何可迭代的对象都可以作为列表解析的输入；任何可以在for循环中打包的对象也都可以放在解析里。例如，文本文件是可迭代的；文件的迭代器每次调用_1^又七_都会返回文件的一行。那么我们早一点提到的通信录的例子（尝试zip函数）也可以使用列表解析实现：

```
import sys
filename = sys.argv[1]

with open(filename) as file:
    header = file.readline(>•strip().split(' \t ' >
    contacts =[
              diet (
                     zip(header, line.strip().split(¹\t'))
                   )for line in file
               ]
```

这一次，我加人了一些空格来增强可读性（列表解析不是必须一行写完)。这个例子和之前的版本完成了同样的事情：通过zip的头和文件中每个分隔好的行来创建一个字典的列表D

呃，什么？如果那些代码或者注释不清楚也不用担心，它确实有一些让人困惑。一个小小的列表解析在这里完成了一大堆工作.但代码既难懂又难读，而且还不好维护。这个例子说明列表解析并不总是最好的选择；很多程序员会认为之前for循环的版本比这个版本可读性更好。记住：我们提供的工具不应该被滥用！对工作总要选择正确的工具，而且工作就是要写出易于维护的代码。

集合和字典解析

解析并不限于列表。我们还可以借助于类似的花括号语法来创建集合和字典解析。先讲集合。一种创建集合的方法是把列表解析包装在set（>构造函数中，从而使其转换成一个集合。但是为什么在我们直接创建一个集合时，要将内存浪费在一个用完就丢弃的中间列表上呢？

这里是我们使用一个命名元组来为一个作者/书名/类型的三元组建模，然后检索出属于某一指定类型的所有作者的集合。

```
from collections import namedtuple
Book = namedtuple("Book", "author title genre")
books =[
          Book("Pratchett", "Nightwatch", "fantasy"),
          Book("Pratchett", "Thief Of Time", "fantasy"),
          Book《"Le Guin", "The Dispossessed", "seifi"),
          Book("Le Guin", "A Wizard Of Earthsea", "fantasy").
          Book ("Turner11, "The Thief", ·f fantasy"),
          Book<"Phillips", "Preston Diamond", "western"),
          Book《"Phillips", "Twice Upon A Time", "scifi"),
          )

fantasy_authors = {
          b.author for b in books if b.genre = 'fantasy'}
```

这个集合解析无疑比所需的设置短得多！如果我们使用列表解析，那么 Terry Pratchett 会出现两次。但对于集合，由于其会去掉重复元素的本质特点，所以最后我们得到的结果是：

```
»> fantasy-authors
{'Turner|, 'Pratchett*, 'Le Guin'}
```

我们可以引人一个冒号来创建字典解析。使用key:value对，将一个序列转换成字典。例如，当我们知道书名而需要快速查找作者或者类型时就会非常有用。我们可以使用字典解析将书名映射到书对象：

```
fantasy-titles = {
          b.title: b for b in books if b.genre == 'fantasy1}
```

现在，我们有了一个字典，并可以使用普通的语法对书进行查找„

综上，解析并不是高级的Python,而且也不是应该避免的“非面向对象”工具。它们只是从现有的序列创建列表、集合、字典的一种更精确、更优化的语法。

生成器表达式

有时候我们想要处理一个新序列，但不想在系统内存中放置一个新的列表、集合或者字典。如果我们只是对项目做一次遍历，而且也不关心是否要创建一个最终的对象容器，那么创建这个容器就是一种内存浪费。每当处理一个项目时，我们只需要在这个时刻将当

前的对象存储在内存中。但是如果我们创建一个容器，就必须在开始处理所有对象之前就将它们存储在这个容器中

以处理日志文件的程序为例。一个非常简单的日志文件可能包含以下格式的信息：

```
Jan  26,  2010  11:25:25      DEBUG          This is a debugging message.
Jan  26,  2010  11:25:36      INFO           This is an information method.
Jan  26,  2010  11:25:46      WARNING        This is a warning. It could be serious.
Jan  26,  2010  11:25:52      WARNING        Another warning sent.
Jan  26,  2010  11:25:59      INFO           Here * s some information.
Jan  26,  2010  11:26:13      DEBUG          Debug messages are only useful if you
want to   figure something out.
Jan  26,  2010  11:26:32      INFO           Information is usually harmless, but
helpful,
Jan  26,  2010  11:26:40      WARNING        Warnings should be heeded.
Jan  26,  2010  11:26:54      WARNING        Watch for warnings.
```

日志文件在web服务器、数据库或邮件服务器中都很常见，它有可能包含G级的数据fi。如果我们想处理这样的日志文件的每一行，肯定不会对这些行使用列表解析；因为它会生成一个包含文件中所有行的列表。根据操作系统的不同，这样内存很可能放不下，而且会达到计算机的极限。

如果我们对这个日志文件使用for循环，可以一次只处理一行而不需要将下一行读入内存。如果使用解析语法也能达到相同的效果是不是挺好？

生成器表达式因此应运而生。它们使用与解析相同的语法，但是不会创建一个最终的容器对象。要创建一个生成器表达式，需要将解析包装在（）而不是[]或{丨中。

下面的代码将一个日志文件解析成先前提到的格式，然后输出仅包含有WARNING行的新日志文件。

```
import sys

inname = sys.argv[1]
outname = sys.argv[2]

with open(inname) as infile:
    with open(outname, "w") as outfile:
    warnings = (1 for 1 in infile if 'WARNING' in 1)
    for 1 in warnings:
```

```
        outfile.write(l)
```

这个程序在命令行中需要两个文件名，使用生成器表达式来过滤所有的警告（在这个例子中，它使用了 **if**语法，并不对该行进行任何修改），然后将这些警告输出到另一个文件中。如果我们将它用于我们的样例文件，那么输出是这样的：

```
Jan 26, 2010 11:25:46    WARNING        This is a warning. It could be serious.
Jan 26, 2010 11:25:52    WARNING        Another warning sent.
Jan 26, 2010 11:26:40    WARNING        Warnings should be heeded.
Jan 26, 2010 11:26:54    WARNING        Watch for warnings.
```

当然，对于这样的一个短小的输人文件，即使使用列表解析也会很安全，但如果文件有百万行时，生成器表达式会在内存和速度方面都产生很大的影响。

生成器表达式在函数内部调用方面也很有用。例如，我们可以在一个生成器表达式而不是列表中调用**sum**、**min**或者**max**，因为这些函数一次只处理一个对象，我们只对结果感兴趣，对中间容器并不感兴趣。

通常，任何可能的时候都使用生成器表达式。如果我们实际上并不需要一个列表、集合或者字典，而只需要过滤或者转换序列中的项目，那么生成器表达式是最高效的。如果我们需要知道列表的长度，或者对结果进行排序、去重，或者创建一个字典，那就不得不使用解析语法了。

生成器

实际上，生成器表达式也是解析的一种；它们将更高级（这一次真的是更高级了！）的生成器语法压缩到一行。更强大的生成器语法看起来会比以往看到的一切都不面向对象，但是我们会再次发现，它其实是用一种简单的语法快捷方式来创建某种对象。

让我们继续使用日志文件的例子。如果我们想要从输出文件中删除掉**WARNING**列（由于这个文件只包含警告，所以它是冗余的），我们有可读性不同的几种选择。我们可以用生成器表达式来实现：

```
import sys
inname, outname = sys.argv[1:3]

with open(inname) as infile:
    with open(outname, "w") as outfile:
```

```
        warnings = (1.replace(¹\tWARNING',")
                    for 1 in infile if 'WARNING' in 1)
        for 1 in warnings:
            outfile.write(1)
```

这样做可读性非常好，尽管我并不想把表达式写得比这更复杂。我们也可以使用普通的**for**循环来实现：

```
import sys
inname, outname = sys.argv[1:3]

with open(inname) as infile:
    with open(outname, "w") as outfile:
        for 1 in infile:
            if 'WARNING' in 1:
                outfile.write(1.replace('\tWARNING',      ••))
```

这样做维护性好，但是这样少的代码却有这么多级的缩进，看起来也有点丑。现在，让我们考虑一种真正的面向对象的解决方法，不需要任何快捷方式：

```
import sys
inname, outname = sys.argv[1:3]

class WarningFilter:
    def __init__(self, insequence):
        self.insequence = insequence
    def __iter__(self):
        return self
    def __next__(self):
        1 = self.insequence.readline()
        while 1 and 'WARNING' not in 1:
            1 = self.insequence.readline()
        if not 1:
            raise Stoplteration
        return 1.replace('\tWARNING',     ■•)

with open(inname) as infile:
    with open (outname, f?w") as outf ile :
```

```
filter = WarningFilter(infile)
for l in filter:
    outfile.write(l)
```

这是毋府置疑的：代码很丑，而且可读性也差。这儿到底发生了什么？我们首先创建r一个对象，它将一个文件对象作为输入，并且提供了一t_next_方法使之可以像一个迭代器一样支持for循坏操作。该方法读取文件的行，如果不是WARNING则丢弃。当它遇到WARNING时，则将该行返回，然后for循环再次调用_next_获取下一行。当文件读取完毕后，会抛出StopIteration来告诉for循环我们结束了。与其他的示例代码相比，这看起来非常丑，但却很强大：因为现在我们有了一个类，可以对它做任何想做的事情。

在此背景下，我们最后来看一个生成器的例子。这个例子做的事情与上一个例子完全一样：创建一个对象，允许我们可以对输人进行遍历：

```
import sys
inname, outname = sys.argv[1:3]

def warnings_filter(insequence):
    for l in insequence:
        if 'WARNING' in l:
            yield l.replace('\tWARNING',

with open(inname) as infile:
    with open(outname, "w") as outfile:
        filter = warnings_filter(infile)
        for l in filter:
            outfile.write(l)
```

好了，这样做可读性还不错……至少它很短。但它究竟做了什么就看不出来了。而且，yield是什么？

先说最后一个问题：yield是生成器的关键字。当Python在一个函数中看到yield，它会取出这个函数并把它包装在一个对象中，就像前一个例子那样。可以认为yield语句和return语句是一样的，它退出一个函数并且返回一行。但与return不同的是，丐函数再次被调用的时候，会从h次结束的地方启动，也就是从yield语句后面的行开始。在这个例子驱,yield语句后面没有行了，所以会跳到for循环的下一个迭代处..由于yield

语句是在if语句里面，它只会获取包含WARNING的行。

尽管看起来这个函数只是在简单地对行进行遍历，但它确实创建了一个对象，一个生成器对象。

```
»> print (warnings filter ((l))
<generator object warnings filter at 0xb728c6bc>
```

我将一个空列表作为一个迭代器传递给该函数，这个函数所做的事情就是创建并返回一″生成器对象.：•这个对象有_iter_和_next_方法，很像我们在之前的例子中创建的那个对象。每当_next_被调用，生成器都会执行到直到它找到yield语句。然后它会返回yield的值，当_next_T一次被调用时，它会在上次停止的地方继续执行。

生成器的使用并不是那么高级，但如果你没有意识到这个函数在创建一个对象.那看起来就会很神奇。我们甚至能在一个单一函数内多次调用yield,它将仅仅选择最近一次的yield,然后再去找下一个。

生成器的内容比我们介绍的要多很多。当我们调用yield时，将一些值传递回生成器，这将使它们变成协程（coroutines）。从对象的技术上讲，协程鼓励我们用与之前讨论的面向对象原则不同的角度去思考，但协程超出了本书的范围。如果你有兴趣r解更多，就自行搜索一下吧。

方法重载的另一种选择

许多面向对象编程语言中的一个突出的特征就是一个叫作方法重载的「.具。简单来说，方法重载是指多个名字相同但可以接收不同参数集的方法3例如，在静态类型语言中，如果我们需要一个方法能够同时接收整数和字符串作为参数，方法重载就很有用。在非面向对象语言中，我们可能需要两个函数add_s和add_i来适应这种情况在静态类型的面向对象语言中,我们则需要两个叫作add的方法,一个接收字符串，另一个接收整数,:,

在Python中，我们只需要一个方法就可以接收任何类型对象。也许它需要对对象类型做一些测试（例如，如果是字符串，则将其转化为整数），但只需一个方法。

当然，当我们需要一个可以接收不同数量或者种类的参数的同名方法时，方法重载也很有用。例如，一个电子邮件消息方法也许有两个版本，其中一个接收发送方邮件地址作为参数，另一个方法则可以寻找一个默认地址来替代。Python不允许多个方法用同一个名字，但它确实能提供一种不同但同样灵活的接口。

在之前的例子中，我们已经看到过一些可行的传递参数给方法和函数的方式，但现在

我们将讨论所有的细节。最简单的是不接收参数的函数。其实我们并不需要例子，但为了完整还是给一个例子：

```
def no_args():
    pass
```

以下是它是如何被调用的：

```
no_args()
```

一个接收参数的函数会提供一个用逗号间隔的参数名称列表，只需要提供每一个参数的名称就够。

当调用一个函数时，必须按顺序指定那些位置上的参数，不能错过或跳过任何一个。在之前的例子中，这是我们指定参数最常规的方式：

```
def mandatory_args(x, y, z):
    pass
```

调用方式如下：

```
mandatory-args("a string", a_variable, 5)
```

任何类型对象都可以作为参数被传递：一个对象、一个容器、一种基本类塑，甚至函数和类。上面的代码展示了传递一个硬编码字符串、一个未知变量和一个数字给该函数。

默认参数

如果我们想让一个参数是可选的，我们并不需要再创建第2个不同参数集的方法，而只需在第1个方法中使用等号为其设一个默认值。如果调用代码没有提供该参数，这个参数就会将被赋予默认值。然而，调用代码时也可选择传递一个不同的值来覆盖默认值。通常，默认值适合设为None、空字符串或空列表。

以下为一个带有默认参数的函数定义：

```
def default_arguments (x, y, z, a="Some String*1, b=False):
    pass
```

前3个参数仍然是在调用代码中必须要传递的，剩下的两个参数则具有默认值。

我们可以通过多种方式调用该函数。我们可以按序提供所有参数，就好像所有参数为位置参数一样。

```
kwargs ("a string", variable, 8, •'", True)
```

或者我们仅仅按序提供必须输入的参数，其他关键字参数则被赋予默认值：

```
kwargs("a longer string", some_variable, 14)
```

在调用函数时，我们也可以用等号赋值语句，这样可以以不同的顺序来提供参数值，并且也可以跳过那些我们并不感兴趣的参数，使它保持默认值。例如，我们能跳过第1个关键字参数并提供第2个：

```
kwargs("a string", variable, 14, b=True)
```

更惊奇的是，我们甚至可以使用等号赋值语句来打乱位置参数的顺序，只要所有的参数都提供了值。

```
»> kwargs (y=l,z=2 ,x=3, a="hi")
3 1 2 hi False
```

有了这么多选择，似乎很难确定选择哪个，但如果你可以把位置参数当作一个有序列表，把关键字参数当作类似字典的东西，那么你将发现正确的布局看起来有条不紊。如果你需要调用者指定某个参数值，那就把这个参数设置成必需的；如果你有一个合理的默认值，那么把它作为关键字参数。选择以哪种方式调用该方法一般需要看方法本身，它取决于哪些参数需要被提供，而哪些参数可以被设为具有默认值。

关于关键字参数有一点还需要注意，我们提供的作为默认参数值的任何东西 都是在函数第一次解释时就计算完的，而不是在函数被调用时:，这意味着我们不能动态地生成默认值。例如，以下代码并不会像预期那样展现：

```
number = 5
def funky_function (niamber=number):
    print(number)

number=6
funky_function(8)
funky_function()
print(number)
```

如果我们执行这段代码，它将首先输出数字8,但是对后面的无参数调用，则会输出数字5。最后一行代码的输出可以证明我们已经将变量的值设为数字6,但函数被调用时,

还是打印出数字5;默认值是在函数被定义时计算的，而不是被调用时。

使用空容器也会很奇怪。例如，让调用代码提供一个在我们的函数中操作的列表，但列表是可选的，这种情况很常见。我们想使用一个空列表作为默认参数。但我们不能这样做，因为当代码第一次被构建时，它将只会创建一个列表：

```
»> def hello (b=[]):
        b.append('a')
...     print(b)

»> hello ()
[ 'a' ]
»> hello ()
['a', 'a']
```

哎呀，这和我们的预期大不一样。通常解决该问题的方法是将默认值设为None，然后在方法内部使用固定语句if argument is None: arg =[]。一定要注意这个！

可变参数列表

默认值本身并不能让我们获得方法重载全部的灵活性优点。使Python真正灵活的，是编写可以接收任意数M的位置参数或关键字参数的方法的能力，而不需要显式地命名这些参数。我们还能传递任意的列表和字典给这些函数。

例如，一个需要接收一个链接或链接的列表来下载网页的函数，它就可以使用这种可变参数或叫$_{varargs}$。与预期得到单个的链接列表作为参数不同的是，我们可以接收一组任意数量的参数，每一个参数都是一个不同的链接。为此，我们需要通过在函数定义时指定*操作符来操作：

```
def get__pages (*links):
    for link in links:
        #使用urllib下载该链接
        print(link)
```

links说明“我可以接收任意数量的参数，然后将它们放到一个叫作links的字符串列表中。”如果我们只提供一个参数，它将是只有一个元素的列表；如果不提供参数，它将是一个空列表。因此，所有以下函数的调用都是有效的：

```
get—pages()
get—pages(*http://www.archlinux.org•)
get_pages(* http://www.archlinux.org',
            * http://ccphillips.net/•)
```

我们还能接收任意的关键字参数。这些参数将作为一个字典传人到函数中，它们在函数声明中用两个星号指定（如**k_{Wargs}）。该丁_具常常在配置安装中使用。下面的类允许我们指定一组带有默认值的选项：

```
class Options:
    default_options = {
            •port•: 21,
            * host1: 'localhost *,
            f username *: None,
            1 password *: None,
            'debug': False,
            }
    def __init__(self, **kwargs):
        self .options = diet (Options.default__options)
        self.options.update(kwargs)

    def  getitem_(self, key):
        return self.options[key]
```

在这个类中所有存趣的事情都发生&_init_方法中。我们在类级别上有一个包含默认选项和值的字典。_init_方法做的第一件事就是将该字典做一份副本。我们不直接修改该字典，是为了以防要实例化两套独立的选项集。（记住，类级别上的变量会在类的实例之间共享。）然后，_11ˆ1:_使用新字典中的update方法将任何非默认值修改为以关键字参数提供的值t_getit$_{em}$_方法只是简单地允许我们通过使用索引的语法来使用这个新类。下面的会话是该类T：作的演示：

```
»> options = Options (username=" dusty " , pas sword: " dr ows sap'',
        debug=True)
»> options [' debug']
True
»> options [' port']
21
```

```
»> options [' username ']
'dusty'
```

我们可以使用字典索引的语法来访问我们的options实例，这个字典同时包括默认值和那些我们设置使用关键字参数的值。

关键字参数语法可能会很危险，因为它可能打破“显式优于隐式”的规则。在上述例子中，可以将任意的关键字参数传给Options初始化函数，来表示不存在于默认字典里的选项。由于各个类的目的不同，这未必是一件坏事，但是这会导致在使用这个类去发现哪些是有效的选项时变得异常困难。它还很容易输人令人困惑的错字（例如，输入了 “Debug”而不是“debug”），使得在只需要一个选项的地方加人了两个选项。

如果我们引导类的使用者只传递默认选项（我们甚至可以添加一些代码来强制执行这个规则），那么L面的例子也不是那么糟糕。这些选项都以文档的形式记录在类的定义中，因此可以很容易找到。

当我们需要接收任意参数传递给第2个函数时，关键字参数也是非常有用的，但我们不知道那些参数将会是什么。我们在第3章中创建支持多重继承时见到过这种用法。

当然，我们可以在一个函数调用时联合使用可变参数和可变关键字参数语法。同时，我们也能使川通常的位置参数和默认参数。接下来的例子会有点不自然，但却表明了这4种类型参数分别是怎样T.作的：

```
import shutil
import os.path
  def augmented - move(target-folder, *filenames,
            verbose=False, ♦♦specific):
      ''*Move all filenames into the target__folder, allowing
      specific treatment of certain files.*"


def print_verbose(message, filename):
      '''print the message only if verbose is enabled'''
      if verbose:
            print(message.format(filename))


for filename in filenames:
      target_path = os.path.join(target_folder, filename)
      if filename in specific:
            if specific[filename] == 'ignore':
```

```
                print-verbose Ignoring {0}", filename)
            elif specific[filename] == 'copy*:
                print_verbose ("Copying {0}' ',filename)
                shutil.copyfile(filename, target-path)
        else:
            print-Verbose ("Moving {o}•',filename)
            shutil.move(filename, target-path)
```

这个例子可以处理一个任意的文件列表。第一个参数是目标文件夹，默认的做法是将其余的所有非关键字参数文件都移动到该文件夹下。然后有一个唯一的关键字参数，verbose,告诉我们是否在每个文件处理过程中打印信息。最后，我们可以提供一个字典来包含对指定文件的操作；默认的操作是移动文件，但如果一个有效的字符串操作在关键字参数中被指定时，该操作将被忽略或替代.注意函数中参数的顺序，第一个指定位置参数，然后是*fil$_e$rmm$_{es}$列表，再然后是特定的唯一关键字参数，最后是一个持有其余关键字参数的**specif ic字典。

我们创建一个内部的辅助函数——print_verbose，它只在verbose参数被设置时才会打印信息。通过将功能封装在一个位置，此函数保持了代码的可读性。

通常情况下，该蜗数将会像下面这样被调用：

```
»> augmented_move("move_here", "one", "two")
```

该命令会将文件one和two移动到m$_{○}$ve_h$_{ere}$ H录下，假设它们都存在（该函数没有错误检查和异常处理，所以当文件或目标目录不存在时，它将会彻底地失败)。当移动发生时不会有任何输出，因为verbose默认为False。

如果想查看输出，我们可以这样调用：

```
»> augmen ted-move (''move -here", "three" f verbose=True)
Moving three
```

这次移动一个名为three的文件，并会告诉我们它正在做什么。注意在这个例子中，将verbose像位置参数那样指定是不可能的;我们必须传递这个关键字参数。否则 Python会认为这是*filenams列表中的另一个文件名。

如果想复制或忽略列表中的一些文件，而不是移动它们，我们可以传递额外的关键字参数：

```
»> augmented - move("move-here", "four", "five", "six",
         four:"copy", five="ignore")
```

这将会移动第6个文件，复制第4个文件，但不会显示任何输出，因为我们没有设定verbose。当然我们也可以那样做，并且关键字参数可以以任意顺序提供：

```
»> augmented_move (''move_here" , "seven" , "eight" , "nine",
        seven=''copy" , verbose=True, eight=" ignore ")
Copying seven
Ignoring eight
Moving nine
```

参数拆分

还有一种更漂亮的有关变量参数或者关键字参数的技巧。虽然我们已经在前面的例子中使用过它，但是还不晚，让我们现在再来解释一下。给定一个值的列表或者字典，我们可以将这些值像普通的位置或者关键字参数一样传给一个函数。看一下下面这些代码：

```
def show_args (arg1, arg2, arg3="THREE"):
      print(arg1, arg2, arg3)
some_args = range(3)
more_args = {
           "arg1": "ONE",
           "arg2": "TWO"}

print("Unpacking a sequence:", end="")
show_args(* some_args)
print("Unpacking a diet, end=" ">
show_args(* *more_args)
```

运行后的结果是：

```
Unpacking a sequence: 0 1 2
Unpacking a diet: ONE TWO THREE
```

该函数接收3个参数，其中有一个有默认值。但当我们有一个包含这3个参数的列表时，我们可以在函数调用内用*操作符来把列表拆分成3个参数。如果我们有的是一个参数的字典，那么就用**语法把它拆成关键字参数的集合。

当需要将从用户输入或者外部源（如一个网页、一个文本文件）收集的信息映射到一个函数或者方法调用时，这就会非常有用。

还记得前面的例子中，我们利用一个文本文件的头和行创建了一个包含通信录信息的

列表的字典吗？除了将字典添加到列表中，我们还可以使用关键字拆分来将参数传给特别构建的可以接收相同参数集的Contact对象的_init_方法。试试看你能不能将这个例子以适配成这种方法来完成任务。

函数也是对象

过分强调面向对象原则的编程语言，会对那些不是方法的函数感到越发不满。在这样的语言中，你被期望通过创建一个对象来对涉及的单个方法进行某种程度上的包装。很多情景下，我们需要传递一个很小的对象，而这个对象只是被调用执行某一个操作，在事件驱动的程序中经常会这样做，比如图像开发工具包或者异步服务器；在接下来的两章中我们会介绍使用了它的一些设计模式。

在Python中，我们不需要将这样的方法包装在对象中，因为函数本身就是对象！我们可以为函数设置属性（虽然这样的做法并不常见），之后，在被调用时将它们传过去。这些函数甚至还有几个特殊的属性可以直接访问。下面是另一个比较不自然的例子：

```
def my_function():
    print("The Function Was Called")
my_function.description = "A silly function"

def second_function():
    print("The second was called")
second_function.description = "A sillier function."

def another_function(function):
    print("The description:", end="")
    print(function.description)
    print("The name:", end="")
    print(function.__name__)
    print("The class:", end="")
    print(function.__class__)
    print("Now I'll call the function passed in")
    function()

another_function(my_function)
another_function(second_function)
```

如果运行这段代码，可以看到我们能够将两个不同函数传到第3个函数中，然后得到它们各自不同的输出：

```
The description: A silly function
The name: my_function
The class: <class 1 function'>
Now I * 11 call the function passed in
The Function Was Called
The description: A sillier function.
The name: second-function
The class: <class 'function'>
Now I'll call the function passed in
The second was called
```

我们还给函数设置了一个名为description的属性（诚然，这并不是很好的描述）。我们还可以看到这个函数的_name_属性，可以用来访问它的类，这证明这个函数确实是一个带有属性的对象。然后我们用调用语法（就是括号）来调用这个函数3

函数是顶级对象的原因是，经常用于将它们传递出去并在之后执行，比如当某些特定条件满足时。让我们来创建一个事件驱动的计时器来完成这样的工作：

```
import datetime
import time

class TimedEvent:
    def ____init____(self, endtime, callback):
        self.endtime = endtime
        self.callback = callback

    def ready(self):
        return self.endtime <= datetime.datetime.now()

class Timer:
    def ___ init____(self):
        self.events =[]

    def call-after(self, delay, callback):
        end time = datetime.datetime.now() + \
```

```
                    datetime.timedelta(seconds=delay)

            self.events.append(TimedEvent(end_time, callback))

    def run(self):
        while True:
            ready_events = (e for e in self.events if e.ready())
            for event in ready_events:
                event.callback(self)
                self.events.remove(event)
            time.sleep(0.5)
```

在生产环境中，这样的代码必然还要有另外的文档字符串作为文档。call_after方法至少应该提到delay是以秒为单位的，而callback函数还应该接收一个参数：进行调用的计时器。

这里我们有两个类。TimedEvent类并不真的是为了被其他类访问；它做的所有事情就是保存endtime和callback。在这里我们甚至可以使用tuple或者namedtuple，何是给对象一个行为来告诉我们事件是否已经准备运行会非常方便，所以我们选择使用类。

Timer类可以简单地保存一个即将发生的事件的列表。它有一个call_after方法用来添加一个新的事件。这个方法接收一个参数delay，表示执行回调函数前的秒数，而callback本身是在正确的时间所要执行的函数。回调函数应该接收一个参数。

run方法非常简单，它使用一个生成器表达式来筛选那些时间已经到了的事件，然后按顺序执行它们。计时器会无限循环，所以必须使用一个键盘操作来中断它（C/r/+C或者

在每次迭代之后休眠半秒，这样就不会导致系统挂掉。

这里需要注意的是那些接触回调函数的行。这个函数像任何其他对象一样被传递，而计时器从不知道也不关心函数的原始名字是什么或者在哪里定义的。当调用该函数的时间一到，计时器就会简单地将括号语法应用到存储的变量上。

下面是一组用来测试计时器的回调函数：

```
from timer import Timer
import datetime

def format_time(message, *args):
    now = datetime.datetime.now() .strftime('%I:%M:%S')
    print(message.format(*args, now=now))
```

```
def one(timer):
      format_time (*' {now} : Called One")

def two(timer):
      format_time("{now}: Called Two")

def three(timer):
      format_time("{now}: Called Three")

class Repeater:
      def _init_(self):
           self.count = 0
           def repeater(self, timer):
           format_time (f' {now}: repeat {0}", self.count)
           timer.call_after(5, self.repeater)

timer = Timer()
timer.call     after(1,     one)
timer.call     after(2,     one)
timer.call     after(2,     two)
timer.call     after(4,     two)
timer.call     after  (3,   three)
timer.call     after(6,     three)
repeater = Repeater()
timer.call_after(5, repeater.repeater)
format_time("{now}: Starting")
timer.run()
```

这个例子向我们展示了多个回调函数是如何与计时器进行互动的。第一个函数是format_time函数.它用字符串的format方法将当前时间添加到消息中，并且展示出正在T.作的可变参数。f○rmat_time方法使用可变参数语法，能够接收任意数量的位置参数，然后将它们作为位置参数转给字符串的format方法。在这之后我们创建了 3个简单的间调函数简单地输出当前时间，并输出一个简短的消息告诉我们哪一个回调函数被调用了。

Repeater类说明方法也可以作为回调函数，因为方法也是函数n它同时也展示出为什么timer参数对于回调函数是有用的：我们可以从当前正在运行的回调函数内添加一个

新的定时事件给计时器。

然后，我们简单地创建一个计时器，并将几个在不同时间后调用的事件添加给它。之后我们启动计时器，输出的结果显示事件按照我们预期的顺序被执行了：

```
02:53:35: Starting
02:53:36: Called One
02:53:37: Called One
02:53:37: Called Two
02:53:38: Called Three
02:53:39: Called Two
02:53:40: repeat 0
02:53:41: Called Three
02:53:45: repeat 1
02:53:50: repeat 2
02:53:55: repeat 3
02:54:00: repeat 4
```

使用函数作为属性

函数作为对象的一个很有意思的效果是它们可以被设置成其他对象可调用的属性=对一个已经实例化的对象添加或者改变一个函数是可能的：

```
class A:
      def print(self):
            print("my class is A")


def fake_print():
      print ("my class is not A'*)


a - A()
a.print()
a.print = fake_print
a.print()
```

这些代码创建了一个很简单的包含print方法的类，它不会告诉我们任何我们不知道的事情。然后，我们创建了一个新的函数告诉我们一些我们不相信的事情。

当我们在类A的一个实例上调用print时，它会像预期一样运行。如果我们设置

print方法指向一个新的函数，它会告诉我们一些不一样的东西：

```
my class is A
my class is not A
```

替换类中的方法而不是对象中的方法也是可能的，虽然在这种场景中我们不得不添加一个self参数到参数列表里，这将会改变该对象所有实例中的方法，即便是已经被实例化了的。

很明显，像这样替换方法是非常危险的，而且维护起来也很让人迷惑。看这段代码的人会看到一个方法被调用了，然后会去找原始类里面的方法。但是原始类中的方法并不是被调用的那个。真正搞清楚发生的事情将会变成一个非常棘手的调试过程。

尽管如此，它仍然有它的用处。通常，在运行时替换或者添加方法（也叫猴子补丁，monkey-patching）会用于自动化测试。如果要测试一个客户端服务器应用，在测试客户端时，我们可能不想真正连接到服务器上，因为那样可能会导致意外的资金转移或是把一些尴尬的邮件发给人们。相反，我们可以设置我们的测试代码，替换掉对象中的一些向服务器发请求的关键方法，使它们只是记录该方法被调用过。

猴子补丁还能够用来修补bug，或是给我们正在交互的不能按照我们需要工作的第三方代码添加功能。不过还是应该少用它，因为它几乎总是“混乱地侵人”，虽然有时它是将现有库适配成满足我们需求的唯一方法。

可调用对象

既然函数只是可以对调用语法进行响应的对象，我们就会好奇是否可以写出可被调用的但又不是真正函数的对象。答案是：当然可以！

任何对象都可以变得可调用，只要简单地给它一个可以接收所需参数的_〇311_方法。让我们把汁时器例子中的Repeater类，通过让它可调用，使它变得更容易使用一些：

```
class Repeater:
    def __init__(self):
        self.count = 0
    def __call__(self, timer):
        format_time("{now}: repeat {0}", self.count)
        self.count +=1
        timer.call_after(5, self)
```

```
timer = Timer()
timer.call_after(5, Repeater())
format_time("{now}: Starting")
timer.run()
```

这个例子与之前的类没有太大区别；我们所做的只是把函数repeater的名字改成__call__，从而使对象本身变成可调用的。注意，当我们调用call_after时，我们将Repeater作为参数进行传递。那两个括号创建了一个新类的实例，但它们并不是明确地调用类。调用是一会后在计时器内部发生的，如果我们想要在新实例化的对象上执行_call_方法，我们要用到一个奇怪的语法：Repeater()()。第1对括号构建了对象，第2对括号则执行T_call_方法。

案例学习

为了把本章讲的这些原理结合在一起，让我们来创建一个邮件列表管理器，这个管理器会记录电子邮件地址并把它归到已经命名的分组下面。当需要发送消息时，我们可以选择一个组，然后将消息发送给这个组的所有邮件地址。

现在，在开始这个T.程之前，我们应该有一种安全的方式进行测试，避免把邮件发送给一群真正的人们。幸运的是，Python在这里可以帮助我们，就像测试HTTP服务器一样，它有一个内置的简单邮件传输协议（SMTP)服务器，我们可以发指令给这个服务器然后捕获我们发出的消息而不是真正发给他们。我们可以使用下面的命令来运行这个服务器：

```
python -m smtpd -n -c DebuggingServer localhost:1025
```

在命令行中运行这个命令会在本地的1025端口启动一个SMTP服务器。但我们已经指定它使用DebuggingServer类（来自内置的SMTP模块），这样就不会将邮件发给预期收件人，而只会在收到它们时简单地打印在终端屏幕上。非常巧妙，是不是？

现在，在开始写我们的邮件地址列表前，让我们写一些真正发邮件的代码。当然 Python标准库支持这个，但接口有点复杂，所以我们写一个新的函数来利落地包装一下它：

```
import smtplib
from email.mime.text import MIMEText

def send_email(subject, message, from_addr, *to_addrs,
        host="localhost", port=1025, **headers):
```

```
        email = MIMEText(message)
        email[1 Subject1] = subject
        email[* From'] = from-addr
        for header, value in headers.items():
              email[header】=value

sender = smtplib.SMTP(host, port)
for addr in to_addrs:
        del email[* To•]
        email['To1] = addr
        sender.sendmail(from_addr, addr, email.as_string())
sender•quit()
```

我们并没有彻底地讨论这个方法里的全部代码；标准库的文档可以告诉你高效使用smtplib和email模块所需要的所有信息。

我们在函数调用中同时使用了可变参数和关键字参数语法；所有未知的参数都映射为附加的发送地址；附加的关键字参数则映射为邮件标题。

传给函数中的标题，包含了可以附加到方法中的各个副标题。这样的标题可能会包含Reply-To、Return-Path 和 X-pretty-much-anything。你发现这里的问题了吗？

Python中的有效标识符不能包含字符u通常情况下这个符号表示减法。所以，调用含有Reply-To = myeemail.com的函数是不可能的:，也许我们太想用关键字参数了，因为它是本章新学到的一个工具？

我们不得不把参数转换成一个普通的字典，由于字典中任何字符串都可以作为键使用，所以这样就可以工作丫。默认情况下，我们希望这个字典是空的，但我们不能将默认参数设置为一个空字典。不行，所以我们只好设置默认的参数为None,然后在方法开始的地方设置字典：

```
def send_email(subject, message, from_addr, *to-addrs,
            host = ' *localhost", port=1025, headers=None):

        headers = {} if headers is None else headers
```

如果我们把调试SMTP服务器运行在一个终端上，就可以在Python的解析器上测试这段代码了：

```
»> send -email<"A model subject", "The message contents",
ⁿfrom@example. com" , "tol@exan5>le • com" , "to2@exan5>le.com")
```

然后如果我们查看调试SMTP服务器上的输出结果，会得到下面的内容:

```
---------- MESSAGE FOLLOWS ----------
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: A model subject
From: from@example.com
To: tol@example.com
X-Peer: 127.0.0.1

The message contents
---------- END MESSAGE ----------
---------- MESSAGE FOLLOWS ----------
Content-Type: text/plain; charset=' ' us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: A model subject
From: from@example.com
To: to2@example.com
X—Peer: 127.0.0.1

The message contents
---------- END MESSAGE ----------
```

太棒了，它已经把我们的邮件“发送”给了两个正确的邮件地址，而且包含标题和消息内容。

既然我们已经可以发消息了，就让我们开始写这个电子邮件组管理系统。我们需要一个对象可以以某种方式将邮件地址与它所在的组进行映射,,由于这是一个多对多的关系（任何一个邮件地址都可以在多个组中，而任意一个组都与多个邮件地址关联），我们学过的数据结构都不是很理想，我们可以尝试建立一个组名的字典，与关联的邮件地址列表相映射，但是这样邮件地址会重复；我们也可以尝试建立一个邮件地址的字典，与组名相映射，但这样会导致组名的重复。这两种方式都不是最优的。虽然直觉告诉我们将邮件地址分组的方案可能会更直接，但还是让我们先尝试一下后一种方法。

由于字典中的值总会是唯一的邮件地址的集合，我们可以把它们存于一个set。我们可以使用Clefaultdict来保证对于每个键来说，永远都有一个可用的set：

```
from collections import defaultdict
class MailingList:
      *'*Manage groups of e-mail addresses for sending e-mails.!''
      def _____init_____(self):
            self.email— map = defaultdict(set)

                  def add—to_group(self, email, group):
             self.email_map[email].add(group)
```

现在，让我们添加一个方法，允许我们将一个或多个组中的全部邮件地址收集起来。我们可以使用集合解析来很容易地完成它：

```
def emails_in_groups(self, *groups):
      groups = set(groups)
      return {e for <e, g) in self.email— map•items()
               if g & groups}
```

好了，这个集合解析需要解释，对吗？首先看我们是对什么进行迭代的：self. emailjap. itemsO。当然，那个方法对字典中的每一个项目都返回了一个键值对元组。这些值是代表组的字符串集合。我们将它们分割成两个变M,针对电子邮件（E-mail ）和组 group ）简称为e和g。由于最终目标的输出是一个邮件地址的集合，所以我们只为每一个项目返回键（即邮件地址）。

唯一剩下的不合理的是那个条件语句。这个语句简单地将groups集合与那些已经关联了邮件的组取交集。如果结果是非空的，邮件地址就会添加进来；否则.就会将它丢弃掉。g & groups语法是g.intersection(groups)的快捷方式；set类通过实现特殊的方法_and_调用intersection来完成这项工作。

现在，利用这些构建好的模块，我们可以简单地将一个方法加人到MailingList类中，来实现发送邮件给特定的组：

```
def send—mailing(self, subject, message, from_addr,
            ★groups, **kwargs):
      emails = self.emails_in_groups(*groups)
      send—email(subject, message, from—addr,
                  ★emails, *^kwargs)
```

这个函数强调可变参数，作为输人，它将一个组列表作为变量参数，而其他可选的关键字参数则作为字典，它根本不关心这些关键字参数，而只是简单地把这些参数传递给我们之前定义的send_email喊数然后获取该特定组的邮件列表，然后把它们作为可变参数传给 send_email〇

测试这个程序要保证SMTP调试服务器在一个命令提示符中运行，然后在第2个命令提示符中加载这段代码：

```
»> python -i mailing—list.py
```

用下面的语句创建一个MailingList对象：

```
»> m = MailingList ()
```

然后用下面的行创违一些假的邮件地址和组：

```
»> m. add—to—group ("friendl@example. com" , "friends")
»> m. add—to—group ("friend2@example • com" , "friends")
»> m. add_to_group ("familyl@exan5>le. com" , "family")
»> m. add—to—group ("prol@example • com" , "professional")
```

最后，我们使用一个命令给指定组发送邮件：

```
»> m. send_mailing ("A Party",
"Friends aind family only: a party" , "me6exan^>le. com" , "friends",
"family" , headers={ ''Reply-To" : "me2@example. com" })
```

给指定组中的邮件地址发送的邮件会在SMTP服务器的控制台中显示。

练习

如果你在H常编码中不常m到解析，你要做的第一件事就是在已经存在的代码中搜索，找到一些for循环,，看看是否其中的任何一个能够被轻易地转化成一个生成器表达式，或是一个列表、集合或字典解析。

测试表明列表解析比for循环快很多，这可以通过内置的timeit模块来实现。通过查看timeit. timeit函数的帮助文档，可以了解到如何使用它。大体上要写两个相同功能的函数，一个使用列表解析，另一个使用for循环。将每个闲数传人timeit．timeit，然后比较结果。如果你喜欢冒险，也可以比较生成器和生成器表达式。使用timeit测试

代码会让人上瘾，所以记住，代码并不需要是超快的，除非它有巨大的执行次数，比如操作一个巨大的输人列表或日志文件。

尝试使用组作为字典的键，邮件地址列表作为值，来编写案例学习中的代码。你很可能会为所做的改变之小而感到惊讶。如果你感兴趣，试着进行返工来接收姓氏和名字以及邮件地址。然后允许自定义邮件消息（使用str. format ）来在每个消息中包含这些姓氏或名字。

除了 send_mailing方法本身，MailingList对象也非常通用。想想需要做点什么就能使它完成任何对邮件地址的通用操作，而不只是发送邮件。提示：回调函数会非常有用。

试试生成器函数。从需要多个值的基本的迭代器开始（数学序列是典型的例子，如果你想不出更好的，那么斐波那契数列也是经常使用的）。尝试一些高级的生成器用法，比如输入多个列表，然后合并产生的值。生成器还可以用在文件上，你能写出一个简单的生成器显示两个文件中相同的行吗？

总结

本章涉及了几个差别很大的话题，每个话题都代表了 Python中既重要又受欢迎的非面向对象功能。这正是因为我们可以使用面向对象的原则，但并不意味着我们应该使用！

无论如何，我们还看到了所谓的“Python方式”，其实就是提供了传统的面向对象语法的快捷方式。了解了这些工具背后的面向对象原则，我们就能够有效地在(*1己的类中使用它们。

我们已经介绍了：

- 内建函数。
- 解析和生成器。
- 函数参数、可变参数和关键字参数。
- 回调函数和可调用对象。

在下一章中，我们将学习设计模式，为面向对象程序员用来创建可维护的应用而构建模块。像本章我们所看到的一样，Python提供了很多我们可以直接使用的流行的设计模式语法。

8

第8章 设计模式1

前面我们已经讨论r面向对象编程的基本构建模块。现在，让我们来看看由这些模块可以构建哪些二级结构。这些更高层次的结构被称为设计模式，能够帮助组织复杂的系统。在接下来的两章，我们将讨论：

- 什么是设计模式，
- 许多具体的模式。
- 在Python中每个模式的标准实现。
- 代替某些模式的Python语法。

设计模式

当工程师和建筑师决定建造一座桥、一座塔或一栋楼时，他们会遵循一定的原则，确保其结构的完整性。现在已经有许多可行的桥梁设计的方案（例如悬索桥或悬臂桥），工程师如果不使用这些标准设计中的任何一种，也没有一种杰出的新设计，那么他/她设计的桥梁就很可能会倒塌

设计模式试阁将这种相同的对正确设计的结构的一般定义应用到软件工程当中。使用不同的设计模式来解决各种普遍性问题。设计模式的创造者首先将开发人员在不同场景下面临的普遍问题辨识出来。接着他们依据面向对象的设计原则，提出可能会被视为解决问题的理想方案的建议。

我们已经对一个最常见的设计模式有了相当多的经验，它就是迭代器（iterator）。迭代器模式意味着提供了一个规范地循环访问序列中项目的公共接口。从序列中将实际循环操

作分离允许在不干扰执行循环的代码和被循环对象的前提下改变循环操作代码。例如.两个不同的迭代器能从不同的方向或者按照某种排序来执行循环。更进一步，它可以改变对象的内部结构但仍允许其在循环过程中仅使用单一的迭代器接口。

在典型的设计模式范例中，迭代器是一个包含**next**()方法和**done**方法的对象；当序列中已无项目时，后者会返回**True**。在没有内置迭代器支持的编程语言中，迭代器像如下一样进行循环：

```
while not iterator.done():
    item = iterator.next()
# 进行一整处理
```

当然，在Python中，next方法被名为_next_的内置方法取而代之，此外，我们拥有大**S**更加可读的for item in iterator式语法去访问这些项目。和使用done方法不同，Python在结束迭代时会抛出异常类Stoplteration。虽然应用同样的模式，并且依然是基于一种设汁模式的解决方案，但Python为我们提供了一个更具可读性的方法来应用和访问这些模式。

了解一种设计模式并选择在我们的软件中使用它却并不能保证我们可以创造一个“正确”的解决方案。在1907年，魁北克大桥（截至这一天，世界上最长的悬臂桥）在尚未建成之前就倒塌了，因为设计它的工程师严重低估了构造它的钢的重量。同样，在软件开发中，我们可能会错误地选择或应用一种设计模式，并导致依此创建的软件在正常操作或超过原始设计限制的压力下“崩溃”。

任何一种设计模式都会提出一组以一种特定方式相互作用的对象来解决某个普遍性问题，程序员的工作就是，当面临一些特定问题时，识别出其中蕴含的这些普遍性问题，并将通用的设计适配至对此问题的解决方案中。

本章中，我们将回顾几种常见的模式，以及它们在Python中是如何实现的.。Python经常提供一些可替代的语法来简化这些模式的实现。我们将同时讨论“传统”的模式设计方法和这些模式的Python版本。

装饰器模式

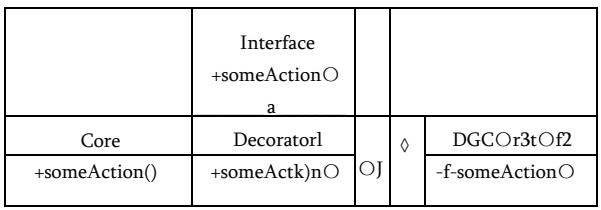
装饰器模式允许我们将一个提供核心功能的对象和其他可以改变这个功能的对象“包袈”在一起。使用装饰对象的任何对象与装饰前后该对象的交互遵循完全相同的方式（也就是说，装饰对象的接口与核心对象相同）。

装饰器模式主要有两种用途：

•增强一个组件向另一个组件发送数据时的响应能力。

•支持多种可选行为。

第2种用途可以适当地代替多重继承。我们可以创建一个核心对象，接着围绕这个核心对象创建一个装饰对象。既然装饰对象和核心对象有相同的接口，我们甚至可以将这个新对象封装在其他的装饰器里。下图是它在UML中的表现形式：



在这里，核心，以及所有的装饰器都实现了特定的接口。装饰器通过组合维护这个接口另一个实例的引用当被调用时，装饰器会在调用它的封装接口之前或之后做一些额外的处理。封装的对象可能是另一个装饰或核心功能。多个装饰器可以互相封装，但处于所有这些装饰器“中心”的对象将会提供核心功能。

装饰器实例

让我们来看一个网络编程中的实例我们将会使用TCP套接字。方法socket. send（〉接收一个字符串的输人字节并输出到另一端的接收套接字。有很多库可以接收套接字并访问函数来发送数据流。让我们来创建一个这样的对象；这是一个等待客户端连接.然后提不用户输入一个字符串作为响应的交互式shell:

```
import socket


def respond(client):
      response = input("Enter a value:")
      client.send(bytes(response,      'utf8'))
      client.close()


server = socket•socket(socket.AF—INET, socket•SOCK—STREAM)
server.bind((!localhost',2401))
server.listen(1)
try:
```

```
        while True:
            client, addr = server.accept()
            respond(client)
finally:
        server.close()
```

respond函数接收一个套接字参数并提示输人数据作为一个回复，接着发送它。之后，我们构建一个服务器套接字，告诉它监听本地计算机上的2401端口（我随机选择了这个端口）。当客户端进行连接时，它会调用respond函数来交互地请求数据并进行适当的响应。需要注意的重要的一点是，respond函数只关心套接字接口的两个方法：send和close。为了测试这一点，我们可以编写一个简单的客户端连接到相同的端口并在退出前将响应输出：

```
import socket


client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(('localhost', 2401))
print("Received: {0}".format(client.recv(1024)))
client.close()
```

为了使用这些程序：

1. 在一个终端打开服务器。
2. 打开第2个终端窗口并且运行客户端。
3. 提示输入一个值：在服务器提示窗口，输人一个值并按回车键。
4. 客户端会接收到你输人的内容，将它打印到控制台，然后退出。再次运行客户端，服务器端会提示输人第2个值。

现在再看一遍我们的服务器代码，我们看到两个部分。respond函数发送数据到一个套接字对象，其余脚本负责创建套接字对象。我们将创建一对装饰器来定制套接字的行为而不用扩展或修改套接字本身。

让我们从“丨ogging”装饰器开始。这个对象只是简单地将任何正在发送给服务器控制台的数据在它们被发送到客户端之前输出：

```
class LogSocket:
        def _init_(self, socket):
        self.socket = socket
```

```
    def send(self, data):
        print("Sending {0} to {1}".format(
            data, self.socket.getpeername()[0]))
        self.socket.send(data)

    def close(self):
        self.socket.close()
```

这个类装饰了一个套接字对象并向客户端套接字呈现send和close接口。一个更好的装饰器也能够实现（并可能定制）所有的套接字方法。它同样应该正确地实现send函数的所有参数（实际上是接收一个可选标记的参数），但是，就让我们这个例子简单些吧！当send函数在此对象中被调用时，它就会和原始套接字一样在发送之前将内容记录并输出到屏幕上。

我们只需要在原始代码的基础上修改一行就可以使用这个装饰器。和使用套接字调用respond函数不同，我们通过一个装饰过的套接字来调用它：

```
respond(LogSocket(client))
```

尽管这很简单.我们还是要扪心自问为什么不简单地扩展套接字类和重写send方法。我们可以在记录数据后调用super（）. send函数进行实际发送T.作。这种设计是有效的。

当在装饰器和继承之间面临选择时，我们应该只在需要根据一些条件对对象进行动态修改时使用装饰器。例如，如果服务器目前处于调试模式，我们就只能启用日志装饰器了。当我们有多个可选行为时，装饰器也比多重继承更加优越。举例来说，我们可以编写另一个装饰器，在send函数被调用时用gZ ip压缩数据：

```
import gzip
from io import BytesIO

class GzipSocket:
    def _____init_____(self, socket):
        self.socket = socket

    def send(self/ data):
        buf = BytesIOO
        zipfile = gzip.GzipFile (f ileobj=buf, mode="wf,)
        zipfile.write (data)
        zipfile.close()
```

```
            self.socket.send(buf.getvalue())
        def close(self):
            self.socket.close()
```

在这个版本中，send方法在发送进来的数据给客户端之前将其压缩。编写一个客户端提取被压缩的内容是可行的，但这里我们没有多余篇幅来详细阐述这个例子。

现在我们已经拥有了两个装饰器，我们可以编写代码使他们在响应过程中进行动态切换…这个例子并不完整，但它阐述了我们在混合和匹配装饰器时可能需要遵循的逻辑：

```
client, addr = server.accept()
if log_send:
    client = LoggingSocket(client)
if client.getpeername()[0] in compress_hosts:
    client = GzipSocket(client)
respond(client)
```

这段代码检查名为log_send的假想配置变量。如果它被启用，它会将套接字封装在LoggingSocket装饰器中。同样，它会检查所连接的客户端地址是否存在于一个可接收压缩内容的已知地址列表中。如果是这样，它会将客户端封装在叫作GzipSocket的装饰器中。不启用装饰器、启用装饰器中的一个或是启用全部两个装饰器，取决于配賢以及正在连接的客户端。试着使用多重继承来编写它，看看你会变得多么困惑！

Python中的装饰器模式

装饰器模式在Python中是很有用的，但我们还有其他的选择。例如，我们可以使用在第7章中讨论过的猴子补丁来达到类似的效果。使用单继承，在一个大型的方法函数中完成“可选”计算可以被视为一个选项，而多重继承也不应该仅仅因为它不适合前面看到的某些特定的例子就被取消！

对于Python而言，在函数中使用这种模式是很常见的。正如我们在前一章中所看到的函数也是对象。事实上这种情况如此普遍以至于Python提供了一种特殊的语法使得装饰器可以很容易地应用到函数中去。

例如，我们可以从更一般的角度来看看logging这个例子。与仅仅记录套接字中发送的调用命令不同，我们发现记录特定函数或方法的调用都是很有用的。下面的示例实现了可以完成这个功能的装饰器：

```
import time
```

```
def log_calls(func):
    def wrapper(*args, **kwargs):
        now = time.time()
        print("Calling {0} with {1} and {2}".format(
            func.__name__, args, kwargs))
        return_value = func(*args, **kwargs)
        print("Executed {0} in {1}ms".format(
            func.__name__, time.time() - now))
        return return_value
    return wrapper

def test1(a,b, c):
    print("\ttest1 called")

def test2(a,b):
    print("\ttest2 called")

def test3(a,b):
    print("\ttest3 called")
    time.sleep(1)

test1 = log_calls(test1)
test2 = log_calls(test2)
test3 = log_calls(test3)

test1(1,2,3)
test2(A,b=5)
test3(6,7)
```

这个装饰器函数非常类似于前面提到的例子；在那种情况下，装饰器接收了一个套接字式的对象并且创建一个类套接字式的对象。这一次，我们的装饰器接收了一个函数对象，并返回一个新的函数对象。这段代码由3个独立部分组成：

- 创建一个函数log_calls,接收一个函数作为参数。
- 在该函数（内部）定义一个名为wrapper的新函数，用来在调用原始函数之前进行一些额外的计算。
- 返回一个替换原始函数的新函数。

3个函数样例展示了装饰器的使用。第3个甚至调用了一个睡眠函数来展示计时测试。我们将每个函数传人装饰器并返回一个新函数。我们将原有函数的函数名分配给新闲数，从而有效地使用装饰过的阐数替换原有函数。

这个语法允许我们动态地创建装饰过的函数对象，这和我们在套接字实例中做过的一样；如果我们没有做函数名的替换，我们甚至可以让装饰过和未装饰过的版本都得以保留以应对不同的场景。

_些装饰器往往用来对不同的函数做永久性的修改。在这种情况下，Python支持一种特殊的语法，可以使函数在定义时应用装饰器。我们曾经在使用属性装饰器时第一次见到它。与定义方法之后再应用装饰器函数不同，我们可以使用(^decorator语法一次性完成所有工作：

```
@log -calIs
def testl(a,b,c>:
    print("\ttestl called")
```

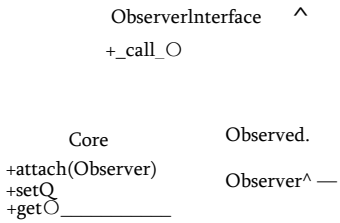
这个语法的主要优点在于，我们可以很容易地看到函数在定义的同时也被装饰了。如果装饰器稍后才被应用于函数，他人在检查代码时可能会忽略函数已经被更改了。想要回答像“为什么我程序中的logging函数会对控制台进行调用？ ”这样的问题会变得更加困难！显然，这种语法只能应用于我们自己定义的函数。如果我们需要装饰我们没写过的函数.我们就不得不使用以前的那些语法。

除了我们在这里见到的一些，还存在更多的装饰器语法。我们在这里没有篇幅讨论这些高级主题，因此请查看Python参考手册或其他教程来获取更多信息。装饰器可以被创建为可调用的对象，而不仅仅是返回新函数的函数。类也可以被装饰，在这种情况下，装饰器会返回一个新的类而不是新的函数。最后，装饰器能够接收参数来基于每个应用程序进行定制。

观察者模式

观察者模式在状态监测和事件处理等场景中是非常有用的。这种模式确保一个核心对象可以由一组未知并可能正在扩展的“观察者”对象来监控。一旦核心对象的某个值发生变化，它便通过调用update（>函数让所有观察者对象知道情况发生了变化。各个观察者在核心对象发生变化时，有可能会负责处理不同的任务；核心对象不知道也不关心这些任务是什么，通常观察者也同样不知道、不关心其他的观察者正在做什么。下图是它在UML

中的表现形式：



观察者实例

观察者模式在一个冗余备份系统中是非常有用的。我们可以编写一个核心对象来维持一些特定的值，然后用一个或多个观察者来创建该对象的一系列副本。例如，这些副本可能存储在数据库、远程主机或者一个本地文件中。让我们来实现这个核心对象的一些属性：

```
class Inventory:
    def _____init_____(self):
        self.observers =[]
        self•—product = None
        self._quantity = 0

    def attach(self, observer):
        self.observers.append(observer)

    @property
    def product(self):
        return self._product
    @product.setter

    def product(self, value):
        self•—product = value
        self•—update— observers()

    @property
    def quantity(self):
        return self.^quantity
```

```
Oquantity.setter
def quantity(self, value):
      self._quantity = value
      self•—update—observers()

def – update– observers(self):
    for observer in self.observers:
        observer()
```

这个对象有两个属性，对其执行赋值，便调方法。该方法所做的全部工作就是对所有可用的观察者进行遍历，好让它们知道发生了一些变化。在这里，我们直接调用观察者对象，而这个对象必须实现_O311_函数来处理变化。这在许多面向对象的编程语言中是不可能的，但是在Python中，这是一种让我们的代码更具可读性的捷径。

现在让我们实现一个简单的观察者对象，它只是将一些状态打印到控制台。

```
class ConsoleObserver:
                    def —init—(self, inventory):
                        self.inventory = inventory

    def ___call___(self):
        print(self.inventory.product)
        print(self.inventory.quantity)
```

这里没什么特别激动的事情；在初始化函数中设置被观察的对象，以及当观察者被调用时，我们会做的“一些事”。我们可以在一个交互式控制台中测试观察者：

```
»> i = Inventory ()
»> c = ConsoleObserver (i)
»> i.attach(c)
»> i.product = "Widget"
Widget

»> i.quantity = 5
Widget
5
```

在将观察者附加到库存对象后，每当我们改变这两个被观察属性时，观察者就会被调

用并执行动作。我们甚至可以添加两个不同的观察者实例:

```
»>  i = Inventory()
»>  cl = ConsoleObserver(i)
»>  c2 = ConsoleObserver(i)
>»  i.attach(cl)
»>  i.attach(c2)
>»  i.product = "Gadget*'
Gadget

Gadget
0
```

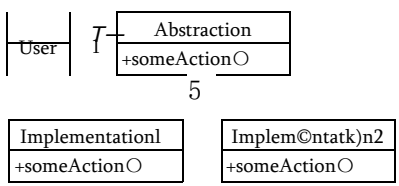
这一次当我们改变产品时，会出现两组输出，每个观察者各一个。这里的关键理念是，我们可以很容鉍地添加两个完全不同类型的观察者，从而将数据同时备份至文件、数据库或互联网应用中。

观察奔模式将正在被观察的代码和执行观察的代码分离。如果我们不使用这种模式，我们就必须要在每个属性中添加代码来处理可能出现的情况，例如登录到控制台，更新一个数据库或文件等。每个任务的代码都会和被观察的对象混合在一起ˆ想要维护它们将会是一个噩梦，在H后添加新的监控功能也会变得非常痛苦。

策略模式

策略模式是一种常见的面向对象编程的抽象模式。针对同一问题，这个模式实现了不同对象采用不同解决方案.客户端代码可以在运行时动态选择其中最恰当的实现。

一般来说，不同的算法各有可取之处；一种可能会比另一种速度快但会占用更多的内存，而当多CPU或分布式系统存在时，我们或许能发现更合适的第3种算法。下阁是策略模式在UML中的表现形式：



连接到策略模式的用户代码仅仅需要知道和它打交道的抽象接口。对同一任务的实际实现可能会选择完全不同的方法，但是无论采用何种方法，接口都是完全相同的。

策略实例

策略模式的一个典咽例子就是排序；这些年来，大M的算法被发明出来用于对一个对象的集合进行排序；快速排序、归并桦序和堆排序都是具有不同特性的快速排序算法，算法本身的使用价值，则取决于输人内容的大小、类型以及系统的需要。

如果我们的客户端代码需要对一个集合进行排序，我们可以将它传递给一个拥有sort（>方法的对象。这个对象可以是一个QuickSorter或MergeSorter对象；两种情况的结果都会相同：一个有序的列表。用来排序的策略是从调用代码中抽象而来的，这使其可以模块化并且可替换

当然，在Python中，我们通常仅仅调用sorted函数或list.sort函数，并且相信它们能以一个接近最优的方式来进行排序。所以我们需要看一个更好的实例，不是吗？

让我们考虑一下桌面墙纸管理软件。首先，策略模式可以用于从硬盘中加载不同格式的图像（JPEG、G丨F、PNG、T丨FF),然后显示它们。这里已经有一些库可以帮我们处理透明度问题，因此，让我们冉多想一些。当阁像被M示到桌面背景时，它能够以不同的方式适应屏幕的大小。例如，假设图像比屏幕小，那它可以平铺到整个屏幕上、居中，或是缩放至合适的大小。同样这里还存在其他更复杂的策略，例如扩展到最大的高度或宽度，将它与固态、半透明或渐变的背景色结合，以及其他的操作。我们稍后或许会添加这些策略，现在让我们从最基本的开始。

我们的策略对象接收两个输人；将要显示的阎像以及一个包含屏幕宽度和高度的元组。它们将根据屏幕的大小使用指定的策略处理图像，并返回一个新的图像。

```
from pygame import image
from pygame.transform import scale
from pygame import Surface


class TiledStrategy:
      def make_background(self, img_file, desktop_size):
            in_img = image.load(img_file)
            out_img = Surface(desktop_size)
            for x in range((out_img.get_width(
                  )// in_img.get_width()) + 1):
```

```
                    for y in range((out_img.get-height(
                          )// in_img.get_height()) + 1):
                          out_img.blit(in_img, (in_img.get_width() * x,
                                in__img. get_height () * y))
              return out-img


class CenteredStrategy:
      def make_background(self, img_file, desktop_size):
            in_img = image•load(img-file)
            out_img = Surface(desktop-size)
            out^img.fill((0,0,0))
            left = (out_img.get_width() - in_img.get_width()) / 2
            top = (out_img.get_height() - in_img.get_height()) / 2
            out_img.blit(in_img, (left, top))
            return out_img


class ScaledStrategy:
      def make_background(self, img_file, desktop_size):
            in_img = image.load(img_file)
            return scale(in_img, desktop_size)
```

这里我们有3种策略，每种都使用pygame来执行他们的任务。每个策略都包含一个make_background方法来接收同样的一组参数。一旦被选定，就会调用适当的策略来创建一个大小合适的桌面背景。TiledStrategy根据输人图像的高度和宽度进行循环并将其重复叠加到新图像的适当位置。CenteredStrategy计算出需要留出多少空间给4个边缘来使图像居中。ScaledStrategy只是简单地将阁像调整成输出需要的尺寸（忽略长宽比)。

考虑一下如何不用策略模式就能实现这些选项之间的切换。我们需要将所有的代码放进一个巨大的方法中，并且使用很尴尬的if语句去进行选择。每次当我们想添加一个新的策略时，我们就不得不使这个函数变得更加笨重。

Python中的策略模式

上面谈论的策略模式的规范实现在大多数面向对象函数库中被非常普遍地运用，但在Python编程中却很少见。你知道为什么吗？

没错：这些类的对象仅仅提供了一个函数。我们可以轻松地调用 call函数使对

象可以被直接调用。由于这些对象没有和任何数据关联，我们实际上可以创建一组顶级函数并传递它们来替代。

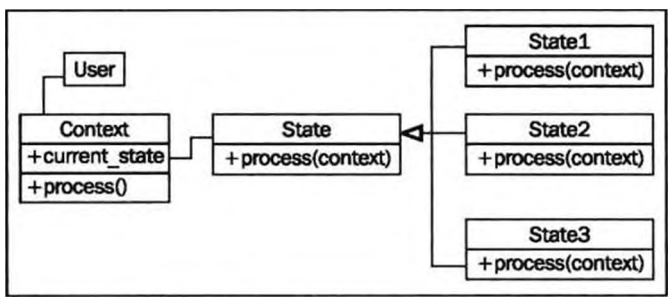
设计模式理论的反对者将因此说，“因为Python有一级函数，所以策略模式是不必要的”。事实上，Python的一级函数允许我们用更直接的方式来实现这一策略模式。熟识这个模式的存在仍然可以帮助我们为我们的程序选择正确的设计；我们只是使用了不同的简单语法来实现它。当我们需要允许客户端代码或终端用户在多个具有相同接口的实现方式之间进行选择时，策略模式或某个可以实现它的顶级函数就应当被使用。

状态模式

状态模式和策略模式在结构上有些相似，但是它的意图和目的却非常不同。状态模式的目的是实现状态转换系统：对象显而易见地处于一个特定状态，同时某些活动可能驱动它转变到另一个不同的状态。

为了实现这个目的，我们需要一个管理者类或上下文类为状态转换提供一个接口。在内部，这个类包含一个指向当前状态的指针；每个状态都知道可以转换为哪些其他状态并如何根据调用的操作转换到这个状态。

所以我们有两种类型的类，一个上下文类和多个状态类。上下文类维护当前状态，并且转发操作到状态类。状态类对其他调用上下文的对象而言通常是隐藏的。它的行为就像一个黑盒子，在内部执行状态管理。下图是状态模式在UML中的表现形式：



状态实例

为了能够将状态模式说明白，让我们构建一个XML解析丁. 具。上下文类自#就是解析器。它将一个字符串作为输人，并将T具处于初始解析的状态。各种解析状态都将吃掉

字符来寻找一个特定的值，当这个值被找到时，它就会变成另一种状态。我们的目的是为每个标签和它的内容创建一个节点对象树。为了便于管理，我们只解析XML的一个子集；标签和标签名称。我们不处理标签中的属性。它将解析标签的文本内容，但不会试图解析文本中存在标签的“混合”内容。下面是一个能够被解析的“简化的XML”文件实例：

```
<book>
      <author>Dusty Phillips</author>
      <publisher>Packt Publishing</publisher>
      <title>Python 3 Object Oriented Progranuning</title>
      <content>
            <chapter>
                  <number>1</number>
                  <title>Object Oriented Design</title>
            </chapter>
            <chapter>
                  <number>2</number>
                  <title>Objects In Python</title>
             </chapter>
      </content>
</book>
```

在关注状态和解析器之前，我们先考虑一下这个程序的输出。我们知道我们想要得到一个Node对象树，但Node是什么样子的呢？显然，这需要了解正在解析的标签的名称，既然它是一棵树，它就应该有一个指向父节点的指针和一个有序排列的子节点列表。某些节点可能会包含文本值，但不是全部都有。让我们先来看看Node类：

```
class Node:
      def ____init____(self, tag_name, parent=None):
            self.parent = parent
        * self•tag-name = tag—name
            self.children =[]
            self.text=""

      def —str—(self):
            if self.text:
                  return self. tag_name + ": ' • + self. text
            else:
```

```
        return self•tag—name
```

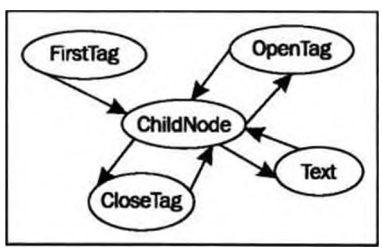
这个类在初始化时就已经为属性设置了默认值。当我们完成后，_str_函数帮助我们实现树结构的可视化。

现在查看一下示例文档，我们需要考虑解析器可以处在什么样的状态。显然，它将从尚未有节点被处理的状态开始。我们将需要一个状态表示正在处理开始标签和结束标签。此外，当我们位于一个包含文本内容的标签中时，我们必须把它当成一种单独的状态来进行处理。

切换状态的时候可能会遇到些麻烦：我们怎么知道下一个节点是一个开始标签、一个结束标签还是一个文本节点呢？我们可以在每个状态中放一段逻辑代码来得出这个结果，但实际上，创建一个新的状态会更有意义，它的唯一目的就是弄清楚下面我们将切换到哪一个状态ˆ如果我们把它称为过渡状态子节点，我们将最终得到以下状态：

- 起始标签（FirstTag ）。
- 子节点（ChildNode ）。
- 开始标签（OpenTag ）。
- 结束标签（CloseTag ）。
- 文本 （ Text)。

起始标签的状态将切换至子节点，由子节点负责决定切换到接下来3个状态中的哪一个；当这些状态结束时，它们将切换回子节点。下面的状态转换关系图显示了可用的状态变化：



状态类负责取出“剩下的字符串”，尽可能多地处理他们知道如何处理的，然后告诉解析器来处理剩下的部分。让我们先构建一个Parser类：

```
class Parser:
      def _____init_____(self, parse—string):
            self•parse_string = parse—string
            self.root = None
```

```
        self.current node = None

        self.state = FirstTag()

    def process(self, remaining_string):
        remaining = self.state.process(remaining_string, self)
        if remaining:
            self.process(remaining)
    def start(self):
        self.process(self.parse_string)
```

这个类的初始化函数已经设置了各状态将会访问的几个变M，$par_{Se_S}trin_{g}$正是我们试图解析的文本。在XML结构中，root节点是“顶”节点。current_node是我们正在添加子节点的节点。

process函数是这个解析器的重要特性，它可以接收剩余的字符串并将其传递到当前状态。解析器本身（self参数）也将被传递到该状态的处理函数以便状态类可以操作它。状态类在完成处理后返回剩余的未解析字符串。接着解析器会递归地调用process方法处理剩余的字符串来创建树的其他部分。

现在，让我们看看FirstTag状态类：

```
class FirstTag:
    def process(self, remaining_string, parser):
        i_start—tag = remaining_string.find(1<')
        i_end_tag = remaining_string.find('>1)
        tag—name = remaining__string[i_start_tag+l:i_end_tag]
        root = Node(tag_name)
        parser.root = parser,current_node = root
        parser.state = ChildNode()
        return remaining -string[i_end—tag+l:]
```

这个状态类找到第一个标签中打开和关闭尖括号的索引（1_代表索引）。你可能认为这种状态是不必要的，因为XML要求开始标签之前不能有文本。然而，这样可能会需要消耗空格，这就是为什么我们需要寻找开始尖括号，而不是假设它是文档中的第一个字符。需要注意的是，这段代码假设输人文件是有效的。一些适当的实现可以检测无效输人，并试图做一些恢复或是显示一个可描述的错误消息。

该方法提取标签的名称并将其分配到解析器的根节点。该方法还会指定它作为

Current-node，因为我们以后会对其添加子节点。

然后接下来的部分很重要；这个方法将解析器对象上的当前状态变成为ChildNode状态。然后它返回字符串的其余部分（在开始标签之后）并允许它被处理。

这样，看似很复杂的ChildNode状态变得只需要一个简单的条件：

```
class ChildNode:
      def process(self, remaining_string, parser):
            stripped = remaining_string.strip()
            if stripped. startswith 〈"</' ' 〉:
                  parser .state = CloseTag0
            elif stripped, startswith (··<ᴹ):
                  parser, state = OpenTag0
            else:
                  parser. state = TextNode0
      return stripped
```

我们可以调用strip ()闲数来删除字符串中的空格。然后解析器会决定下一个是开始标签、关闭标签还是文本字符串。取决于可能会发生什么，它会将解析器设置为一个特定的状态，然后让它解析字符串的其余部分。

除了添加新创建的节点到先前current_node对象的children中并且将它设置成新的current_node，OpenTag状态与FirstTag状态相似。在继续处理之前，它将状态返回到ChildNode状态：

```
class OpenTag:
      def process(self, remaining_string, parser):
            i-start-tag = remaining_string.find(ˡ<ˡ)
            i_end_tag = remaining_string.find('>')
            tag_name = remaining_string[i_start_tag+l:i_end_tag]
            node = Node(tag_name, parser.current_node)
            parser.current_node.children.append(node)
            parser•current-node = node
            parser.state = ChildNode()
            return remaining_string[i_end_tag+i: ]
```

CloseTag完成的事基本上和它相反。它将解析器中的current__node设置回父节点，使得外部标签添加更多的子节点：

```
class CloseTag:
```

```
    def process(self, remaining—string, parser):
        i—start—tag = remaining_string.find('< *)
        i_end_tag = remaining_string.find(* >')
        assert remaining—string [ i_start—tag+1 ] == "/' •
        tag_name = remaining_string[i—start_tag+2:i_end—tag]
        assert tag_name == parser.current_node.tag_name
        parser.current_node = parser.current_node.parent
        parser, state = ChildNode0
        return remaining__string [ i_end__tag+1: ] . strip ()
```

这两个assert语句帮助我们确保解析字符串是一致的。方法末尾的if语句仅仅确保完成时结束处理,、如果这个节点的父节点为None,意味着我们正处于根节点中。

最后，TextNode状态非常简单地提取下一个结束标签之前的文本并将其设置为当前节点的一个属性值：

```
class TextNode:
    def process(self, remaining—string, parser):
        i_start_tag = remaining—string•find(*<•)
        text = remaining_string[:i_start_tag]
        parser • current—node. text: = text
        parser, state = ChildNode0
        return remaining_string[i_start_tag:]
```

现在我们只需要对我们创建的解析器对象设置初始状态。初始状态是FirstTag对象，所以我们需要将下面的语句添加到方法中：

```
        self.state = FirstTag()
```

为了测试这个类，让我们添加一个主脚本，从命令行中打开文件然后对其进行解析并打印出节点：

```
if _name— == ' •—main_•' :
    import sys
    with open(sys.argv[1]) as file:
        contents = file.read(>
        p = Parser(contents)
        p. start()

        nodes = [p.root]
```

```
while nodes:
    node = nodes.pop(0)
    print(node)
    nodes = node.children + nodes
```

脚本所做的全部工作就是打开文件，加载内容并且解析。然后按顺序打印每个节点和它的子节点。最初添加到节点类的_3七1_方法负责格式化打印的节点。如果我们运行这个脚本解析先前的实例，它会输出如下的树：

```
book
author: Dusty Phillips
publisher: Packt Publishing
title: Python 3 Object Oriented Programming
content
chapter
number: 1
title: Object Oriented Design
chapter
number: 2
title: Objects In Python
```

将它和原先的XML文档相比较可以看出解析器工作正常。

状态和策略模式的对比

状态模式看起来非常类似于策略模式，事实上它们两个的UML图是完全相同的。而且实现方式也是相同的；我们甚至可以将状态写成一级函数而不是像策略中建议的那样将它封装进对象当中。

虽然两种模式具有相同的结构，其目的却是非常不同的。策略模式用于在运行时选择一种算法；一般来说，只有其中一个算法将被会选择用作特定的用途。另一方面，状态模式被设计成允许在不同状态之间进行动态切换，正如同某些过程的发展那样。在代码中，最主要的区别在于，策略模式通常对其他策略对象没有意识。而在状态模式中，状态或上下文需要知道它们将会切换到什么样的状态。

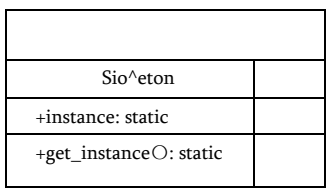
单件模式

单件模式是一种最具争议的模式；许多人指责它是一个“反模式”，是一种应该避免而不是推广的模式。在Python中，如果有人使用了单件模式，几乎可以肯定他们在某些方面出错了，可能因为他们之前使用的是一种更严格的编程语言。

但我们为什么要讨论这个问题呢?首先单件模式是一种最著名的设计模式。其次，它在极度面向对象的语言中非常有用，而且是面向对象编程的一个重要组成部分。最后，单件模式背后的思想是很有用的，即使我们在Python中需要用完全不同的方式来实现这一理念。

单件模式背后的基本理念是允许一些对象只存在一个实例。一般来说，这个对象是一种类似于我们在第5章中i、j•论过的管理类。这样的对象往往需要被各种各样的其他对象所引用，然后在它们的各种方法和构造函数中传递管理对象的引用，这使得它们的代码很难被阅读。

相反，当使用单件时，不同的对象可以从管理对象类中请求它的单一对象实例，所以我们不需要传递对它的引用。虽然UML图并不能完全描述它，但为了章节的完整性我们在这里列出来了：



在大多数的编程环境中，单件通过使构造函数私有化（所以没有人可以创建额外的实例）,然后提供一个静态方法来获得单一实例。这个方法在第一次调用时将创建一个新实例,然后无论何时被调用都会返回同一个实例。

单件的实现方式

Python没有私有构造函数，但是正由于这个原因，它有个更好的东西。我们可以使用_neW_类方法来保证只有一个实例被创建出来。

```
class OneOnly:
    _singleton = None
    def  new—(els, *args, **kwargs):
        if not els._singleton:
            els•—singleton = super(OneOnly, els
```

```
                ).__new__(els, *args, **kwargs)
        return els.—singleton
```

当_1ˆ«_方法被调川时，它一般会构造该类的一个新实例。当我们重写它时，我们首先检查这个单件的实例是否已经被创建出来了。如果没有，我们调用super函数来创建它。因此，每当我们调用OneOnly的构造函数时，总是可以得到完全相同的实例：

```
»> 01 = OneOnlyO
»> 02 = OneOnlyO
»» O1 = O2
True
»» O1
<  main__.OneOnly object at 0xb71c008c>
»» O2
<  main__.OneOnly object at 0xb71c008c>
```

这两个对象相等并且具有相同的内存地址；意味着它们是同一对象。这种特殊的实现方式并不是很容易理解，因为单件对象是否已被创建出来并不很明显。当我们调用构造函数时，我们期望得到这个对象的一个新实例；而在这种情况下，我们是无法获得的。或许，当我们真的认为需要一个单件时，该类的一段优秀的文档字符串可以缓解这个问题。

但我们并不需要它。Python程序员不赞成强制用户将代码变为特定的模式。我们可能认为只需要一个类的实例，但其他程序员可能会有不同的想法。单件可能会干扰分布式计算、并行编程和自动化测试。在所有这些情况下，如果一个特定对象的实例有很多个或者是可选的，这将会非常有用，即使“正常”的操作可能永远都不会需要这些东西。

模块变量能够模仿单件

在Python中，通常我们可以使用模块级变量来充分模仿单件模式。对单件而言，如果人们可以在任意时刻对这些变量重新赋值，这当然将是很不“安全”的。但就像我们在第2章中讨论过的私有变量，在Python中这是可以被接受的[3]如果一个人有一个令人信服的理由来改变这些变ft，我们为什么要去阻止他们呢？我们同样不能阻止人们实例化对象的多个实例，再次说明，如果他们有一个充足的理由这么做，那为什么要干预呢？

理想情况下，我们应该创建一个机制可以让他们访问“默认单件”，当需要时也允许他们创建其他实例。虽然从技术层面而言，它根本就不是一个单件，它只是使用了具有Python风格的机制使它的行为像个中-件。

为了使用模块级变M代替单件，我们可以简单地在定义类之后将其实例化。我们能够通过使用单件来改进我们的状态模式。与其每次改变状态时都要创建一个新对象，我们可以创建一个总是可以被访问的模块级变董：

```
class FirstTag:
      def process(self, remaining_string, parser):
            i_start_tag = remaining_string.find('<')
            i_end_tag = remaining_string.find('>')
            tag_name = remaining_string[i_start_tag+1:i_end_tag]
            root = Node(tag_name)
            parser, root = parser. current__node = root
            parser.state = child_node
            return remaining_string[i_end_tag+1:]


class ChildNode:
      def process(self, remaining_string, parser):
            stripped = remaining_string.strip()
            if stripped.startswith ("</"):
                  parser.state = close_tag
            elif stripped.startswith("<"):
                  parser.state = open_tag
            else:
                  parser.state = text_node
            return stripped


class OpenTag:
      def process(self, remaining_string, parser):
            i_start_tag = remaining_string.find('<')
            i_end_tag = remaining_string.find('>')
            tag_name = remaining_string[i_start_tag+1:i_end_tag]
            node = Node(tag_name, parser.current_node)
            parser.current_node.children.append(node)
            parser.current_node = node
            parser.state = chiId_node
            return remaining_string[i_end_tag+1:]


class TextNode:
```

```
        def process(self, remaining_string, parser):
            i_start_tag = remaining_string.find('<')
            text = remaining_string[:i_start_tag]
            parser.current_node.text = text
            parser.state = child_node
            return remaining_string[i_start_tag:]

class CloseTag:
        def process(self, remaining_string, parser):
            i_start_tag = remaining_string.find('</')
            i_end_tag = remaining_string.find('>')
            assert remaining_string[i_start_tag+1] ==
            tag_name = remaining_string[i_start_tag+2:i_end_tag]
            assert tag_name == parser.current_node.tag_name
            parser.current_node = parser.current_node.parent
            parser.state = child_node
            return remaining_string[i_end_tag+1:].strip()

first_tag = FirstTag()
child_node = ChildNode()
text_node = TextNode()
open_tag = OpenTag()
close_tag = CloseTag()
```

这里我们所做的就是创建各种可以被重用的状态类实例。需要注意的是，我们为何能够在类的内部，甚至是在这些变M被定义之前访问这些模块变量？这是因为类内部的代码在被方法调用之前是不会被执行的，并且到那时，整个模块都已经被定义了。

这个例子和之前的区别在于，在这个实例中我们没有创建一群最终会被像垃圾一样回收掉的新实例，我们只是对每个状态重用了同一个状态对象。即使有多个解析器正在运行，也只需要使用这些状态类。当然，如果有人想要创建自己的实例，他们可以这么做。所以它并不是一个真正的单件，但是习惯上会强烈建议按照单件范例来使用。如果某人不希望遵循这些范例，那么他们就必须要面对可能造成的有益或有害的后果。

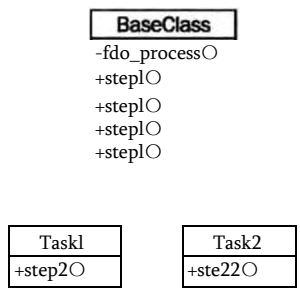
当最初我们创建基于状态的解析器时，你可能想知道为什么我们没有像之前我们做过的一样，通过将解析器对象传递给每一个状态类的_：11^1：_方法来取代将解析器对象传递给处理方法。状态当然可以引用为self.parser。这种对状态模式的实现是完全有效

的，但是它将无法引入单件模式。如果状态对象保留了对解析器的引用，那么它们就不能同时用于引用其他解析器，

需要记住的是，单件和状态是两种具有不同目的的模式D单件模式对于实现状态模式非常有用这一事实并不能证明，在某种程度上，这两种模式之间是存在关联的。

模板模式

模板模式有助于去除重复代码；这符合我们在第5章中讨论过的“不要重复自己”的原则，模板模式是为这种情况设计的，我们有不同的任务需要完成，但这些任务中的一些（不是全部）步骤相同。共同的步骤在基类中被执行，而不同的步骤会在子类中被重写以提供自定义的行为。除了使用基类共享算法相似的部分，在一些方面，它就如同一个通用的策略模式。下面是模板模式在UML中的表现形式：



模板实例

让我们创建一个汽车销售报告单作为例子。我们能在SQLite数据库表中存储销售记录。SQLite是一个基于文件的简单数据库引擎，它允许我们使用SQU吾法来存储记录 Python 3将SQLite包含在其标准库中，所以这里不需要进行设置。

我们需要执行两个常见的任务：

. 选择所有销售的新车并以逗号分隔的格式输出到屏幕上。

•输出一个以逗号分隔的包含所有销售人员和他们销售总额的列表，并且将其保存到一个可以导人到电子表格的文件当中。

这看起来是两个很不相同的任务，但它们有一些共同的特征。在这两个任务中，我们都需要执行如下步骤：

1. 连接数据库。
2. 构造一个查询新车或销售总额的查询函数。
3. 发出查询请求。
4. 将结果格式化为一个以逗号分隔的字符串。
5. 输出数据到一个文件或电子邮件中

这两个任务查询函数的构建和输出步骤是不同的，但是它们的其他步骤是相同的。我们可以使用模板模式将相同步骤放在一个基类中，而将不同步骤放在两个子类中。

在开始之前，让我们用几行SQL代码来创建一个数据库，把一些示例数据放进去：

```
import sqlite3

conn = sqlite3.connect("sales.db")

conn.execute ("CREATE TABLE Sales (salesperson text, ••
              "amt currency, year integer, model text, new boolean)")
conn .execute (*'INSERT INTO Sales values"
              •'  (^im', 16000, 2010, »Honda Fit、·true·)">
conn.execute("INSERT INTO Sales values"
              "(' Tim*, 9000, 2006, * Ford Focus*, * false *)")
conn. execute ('*INSERT INTO Sales values'*
              ■'  (1 Gayle ', 8000, 2004,          * Dodge Neon 1, •false*)"}
conn•execute("INSERT INTO Sales values"
              "('Gayle', 28000, 2009,               * Ford Mustang *,        * true')")
conn.execute("INSERT INTO Sales values"
              "('Gayle ' , 50000, 2010, 'Lincoln Navigator',                          ' true *) *')
conn.execute (** INSERT INTO Sales values"
              " (' D o n、20000, 2008, •Toyota Prius', * false1) ")
conn.commit()
conn.close ()
```

希望你即使不懂SQL也可以看出这里发生了什么;我们已经创建了一个表来保存数据，并使用了 6个插人语句来添加销售记录。数据存储在一个名为sales. db的文件中。现在我们就有一个可以开发模板模式的数据样本了。

既然我们已经列出了模板所必须执行的步骤，我们就从定义包含这些步骤的基类开始。每一个步骤都有自己的方法（使它很容易有选择地重写任何一步），我们有一个更加普遍的

方法叫作依次调用步骤。它不包含任何内容，看起来就像下面的样子：

```
class QueryTemplate:
    def connect(self):
        pass
def construct—query(self):
    pass
def do_query(self):
    pass
def format_results(self):
    pass
def output_results(self):
    pass

def process - format (self):
    self.connect()
    self.construct_query()
    self.do_query()
    self•format—results(>
    self,output_results()
```

process_format方法是被外部客户端调用的最主要的方法。它保证每个步骤都按照顺序执行，但它无法保证这一步骤是在这个类还是在某个子类中执行。对于我们的例子来说，我们的两个类之间有3个方法是相同的：

```
import sqlite3

class QueryTemplate:
    def connect(self):
        self.conn = sqlite3.connect("sales.db")

    def construct—query(self>:
        raise NotlmplementedError()

    def do—query(self):
        results = self. conn. execute(self. query)
        self.results = results. fetchall()
```

```
    def format__results (self):
        output = [ ]
        for row in self.results:
            row =[str(i) for i in row]
            output.append(", ".join(row))
        self.formatted - results = "Xn".join(output)

    def output-results(self):
        raise Notln^leznentedError ()
```

为帮助子类的实现，另外两个没有指定内容的方法会提示NotlmplementedError。在Python中，这是一个指定抽象接口的常见方式。这种方法可以不包含任何实现（使用pass ），甚至可以完全不指定。然而，提不NotlmplementedError可以帮助程序员理解，这个方法是需要被子类重写的；当我们忘记实现这些方法时，一个完全空白或是根本不存在的方法会使得我们在试图实现他们的时候很难进行鉴别和排错。

现在我们有了一个模板类可以来处理这些无聊的细节，但它也是足够灵活的，能够允许各种各样的查询方式来对它进行执行和格式化。最棒的部分在于，如果我们想要将我们的SQLite数据库引擎改变成另一种数据库引擎（比如py-postgresql ），我们只需要在这里，在这个模板类中完成它就可以了，而不需要接触两个（甚至两百个）我们曾经写过的子类。

现在让我们来看一个具体的类：

```
import datetime
class NewVehiclesQuery(QueryTemplate):
    def construct - query(self):
        self.query = "select * from Sales where new=1 true

    def output - results(self):
        print(self•formatted—results)

class UserGrossQuery(QueryTemplate):
    def construct -  query(self):
        self.query = ("select salesperson, sum(amt)
        •'  from Sales group by salesperson")

    def output - results(self):
```

```
filename = "gross_sales_{0}".format(
            datetime .date . today () .strftime ('%Y%m%d')
            )
with open(filename, 'w') as outfile:
        outfile.write(self.formatted_results)
```

这两个类实际上都是相当短的，思考一下他们所做的i作：连接到一个数据库，执行一个查询，格式化结果并输出。这个超类可以负责处理重复性的工作，但也可以很容易地让我们K分这些任务之M的不同步骤。更进一步，我们也可以很容易地更改基类提供的步骤。例如，如果我们想要输出一个格式有别于用逗号分隔的字符串（例如，将一个HTML报告上传到一个网站上），我们依然可以重写format results。

练习

在撰写本章的过程中，我发现，想要举出应当使用设计模式的好例子非常闲难但极富教育意义。正如我在前几章中所建议的那样，与其一一检查如何在现有的项目或是陈旧的项0中应用设计模式，不如思考哪些不同的情况下会用到这些模式。跳出条条框框，在你已有的经验之外进行思考。如果你现在的项目是在银行业务范围内，考虑一下你应该如何作零售或销售程序中应用这些设计模式。如果你通常编写的是网页应用程序，考虑一下使州设计模式来编写一个编译器。从迭代器模式开始，我们一直在通过这本书学习迭代器，包括特殊的理解方式和生成ift法。考虑一下哪些地方你可能想从头开始执行迭代器模式；在哪些对象上你想执行_iter_或_next_阴数？

再看看装饰器模式，想出一些好例子来应用它。要注意关注模式本身，而不是我们讨论过的Python语法；这比实际的模式更加具有普遍性。而且，特殊的装饰器ift法可能也会让你想要在现有的项fl中找到位置来应用它。

有哪些不错的地方可以使用观察者模式？为什么要使用它？你不仅要考虑如何应用模式，而且还要考虑如何不使用观察者模式完成相同的任务？选择使用它你会有怎样的得失?

考虑一下策略模式和状态模式之间的区别。从实现上讲，它们看上去是非常相似的，怛却有着不同的使用目的。你是否能想出这些模式可以互换的场景？是否能够合理地使用策略模式重新设计一个基于状态的系统，或反过来？这些设计实际上会有多少不同？

模板模式是一种使用继承来减少重复代码的最典型的应用，你可能之前已经用过这种方式. 只是不知道它的名字。试着想出至少半打可以应用它的不同场景。如果你能做到这一点，你就会在今后日常编码中到处应用它。

总结

在本章，我们学习到设计模式是一些有益的抽象，能够为常见的编程问题提供“最佳实践”式的解决方案。我们知道，在Python中，由于其动态特性和内置的语法，使得设计模式与它们在其他语言中的演绎看起来很不一样。我们通过实例、UML图以及Python和面向对象语言的静态类之间LX」别的讨论，学习丫：

- 什么是设计模式。
- 迭代器模式。
- 装饰器模式。
- 观察者模式。
- 策略模式和状态模式。
- 模板模式。

下一章，我们将讨论几个更加有用的设计模式以及他们在Python中的应用。

9

第9章 设计模式2

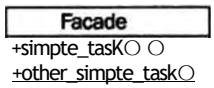
本章是上一章的延伸，将介绍更多的设计模式。同样，我们将涵盖规范示例以及在Python中常见的替代实现方法。我们将会讨论：

. 适配器模式。
- 外观模式。
- 惰性初始化和享元模式。
- 命令模式。
- 柚象工厂模式。
- 组合模式。

适配器模式

和在第8章中介绍的绝大多数模式不同，适配器模式被设计用于与已有代码进行交互.我们不会设计一套全新的对象来实现适配器模式。适配器用于允许两个已存在的对象在一起工作，即使他们的接口不兼容。就像允许USB键盘插人PS/2端口一样，适配器对象位于两个不同的接口之间，对两个接U进行即时转换。适配器对象的唯一工作就是执行转换，转换可能涉及很多任务，比如将参数转换为另一种格式，重新排列参数的顺序，调用名称不同的方法或提供默认参数。

在结构上，适配器模式类似于一个简化的装饰器模式。装饰器模式通常提供它们所替代的相同接口，而适配器模式在两个不同的接口之间进行匹配。下图是适配器模式在UML中的表现形式：



BIG SYSTEM 1	
|Complex Cor	lponent ⊦
jAnottier Com	P○nent j--, ^

这里，接口 1期望调用一个称为make_action(some，argumen(5)的方法。我们已经有个很完美的接口 2能做到我们想要的一切（并且为了避免重复，我不想重写它！），但是它提供的方法被命名为different_action(other，arguments)。适配器类将会实现make_action接口并将参数匹配到已有的接口。

这里的优势在于，代码一次性地对一个接口到其他所有接口进行了全部K配。另一种方法则使得我们每次访问这个代码时，都需要在多个不同的地方对其进行直接转换。

举例来说，假设我们有以下的类，以格式“YYYY-MM-DD”接收一个字符串的日期并通过R期计算一个人的年龄：

```
class AgeCalculator:
      def  init—(self, birthday):
            self.year, self.month/ self.day =(
                        int(x) for x in birthday.split(¹-'))
      def calculate_age(self, date):
            year, month, day =(
                         int(x) for x in date.split('-*))
            age = year - self.year
            if (month,day) < (self.month,self.day):
                  age -= 1
            return age
```

这是一个相当简单的类，可以一眼看出它在做什么。但是，我们必须考虑一下程序员的想法，这里使用了一个特定格式的字符串，而不是Python中十分有用的内置datetime库。我们编写的大多数程序都将要与datetime对象，而不是字符串进行交互。

我们有多种方式来解决这种问题；我们可以重写这个类来接收datetime对象，这将可能会更加准确。但是，如果这个类是由第=方提供的，我们不知道它的内部是什么，或者我们根本就不被允许去更改他们.我们就需要尝试一下别的东西。我们可以按照它原本

的模样来使用这个类，每当我们要通过datetime.date对象来计算年龄时，我们可以调用datetime. date, strftime（ l%Y-%M-%dM函数将其转换为适当的格式。但是转换将发生在很多地方，更糟的是，如果我们错误地将输入成%M，它会采用当前的而不是我们输入的月份！试想一下，如果你已经在十几个不同的地方写过代码，当你意识到自己的错误时必须得挨个回去更改它们会有多么痛苦。这不是可维护的代码，它违反了 DRY 原则。

或者，我们可以编写一个适配器，允许将正常的日期插入到一个普通的 AgeCalculator 类中：

```
import datetime
class DateAgeAdapter:
      def _str_date(self, date):
            return date. strftime (f·%Y-%m-%d'1)


      def —init—(self, birthday):
            birthday = self._str_date(birthday)
            self.calculator = AgeCalculator(birthday)


      def get_age(self, date):
            date = self .__str__date (date)
            return self.calculator.calculate_age(date)
```

该适配器将datetime, date和datetime, time（它们对函数strftime有相同的接口）装换成我们普通的AgeCalculator可以使用的字符串。现在我们可以在新接口上使用原来的代码。我将方法名改成了 get_age以证明调用接口也可能在寻找不同的方法名，而不仅仅是一个不同类型的参数。

创建一个类作为适配器是实现一种模式的常用方式，但是，像往常一样，还有许多其他的方法可以做到这一点。继承和多重继承可以用于将功能添加到一个类中。例如，我们可以从date类中添加适配器，这样它的T. 作原理就与原来的AgeCaloulator相同。

```
import datetime
class AgeableDate(datetime.date):
      def split(self, char):
            return self.year, self.month, self.day
```

像这样的代码在Python中出现真的很让人怀疑是否是合法的。我们这里所做的就是添

加一个split方法，该方法接收一个参数（我们将其忽略掉），并返回一个年、月、日的元组。这与我们的AgeCalculator能够完美结合，因为该代码是以一种特殊的格式调用strip函数，而在这种情况下，strip会返回一个年、月、日的元组。这个AgeCalculator只在意strip函数是否存在，并返回可接收的值；它不在意我们是否真的在传递一个字符串。但它确实有效！

```
»> bd = AgeableDate (1975, 6, 14)
»> todacy = AgeableDate . today ()
»> today
AgeableDate(2010, 2, 23)
»> a = AgeCalculator (bd)
»> a • calculate-age (today)
34
```

在这个特定实例中，这样一个适配器将会很难维护，因为我们很快就会忘记为什么我们需要将一个strip方法添加到一个date类中。这个方法名称是相当令人闲惑的。这可能就是适配器的本质，但我们如果已经创建了一个适配器来显式地替代继承，它的R的是什么将会更加明显。

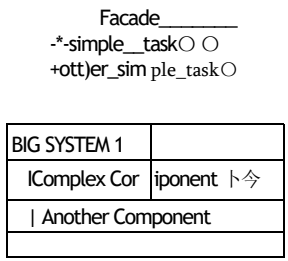
和继承不同，你有时还可以使用猴子补丁法添加方法到现有的类中。它不会与datetime对象一起T.作，因为它不允许在运行时添加新属性，但在一般的类中，我们只需要通过添加方法就能够为调用它的代码提供一个可适配的接口。

我们也可以将函数当成适配器;这并不完全符合标准的适配器模式，但在通常情况下，你可以简单地传递数据到一个函数，然后将它以一个适当的形式返回作为进入另一个接口的人口。

外观模式

外观模式的目的是为拥有多个组件的复杂系统提供简单的接口。这个系统内的对象为了完成复杂的任务和交互需求会进行各种直接的交互。然而，该系统通常存在某些“典型”的用途，并且在这种情况下，这些复杂的交互是不必要的。外观模式允许我们定义一个新的对象来封装该系统的典型使用$_3$任何想要使用这个典型.功能的代码都可以使用这个对象的简化接口。如果另一个项目或这个项目的另一部分发现这个接口过于简单并且需要访问某些更复杂的功能时，它仍然能够直接与系统进行交互。外观模式的UML结构图非常依

赖子系统，但通过另一种方式，它看起来会像这样：



外观模式在很多方面和适配器模式相同。最主要的区别在于，外观模式试图从一个复杂的系统中抽象出一个简单的接口；而适配器只不过是想将一个现有的接口匹配到另一个。

让我们来为电子邮件应用程序编写一个简单的外观模式。正如我们在第7章中看到的，在Python中，发送电子邮件的低层类库是相当复杂的，用于接收消息的两个库的情况更加糟糕。

如果有这样一个简单的类就好了，它允许我们发送一封电子邮件，并通过IMAP或POP3连接列出收件箱中的邮件。为了使我们的例子简短一些，我们将坚持使用丨MAP和SMTP协议：两个处理邮件的完全不同的子系统。我们的外观模式将只执行两个任务；发送电子邮件到一个特定的地址，并通过IMAP连接检查收件箱。程序将对连接做一些常见必要的假设，如主机的SMTP和1MAP是同一台机器，用户名和密码是相同的，并且他们使用标准的端口。这涵盖了电子邮件服务器的大多数情况，但如果程序员需要更多的灵活性，他们可以随时绕过外观模式，直接访问两个子系统。

这个类通过电子邮件服务器主机名以及登录的用户名和密码进行初始化：

```
import smtplib
import imaplib


class EmailFacade:
      def _____init_____(self, host, username, password):
            self.host = host
            self.username = username
            self.password = password
```

send_email方法简单格式化r邮件地址和消息，并使用smtplib来发送它。这不是一个复杂的任务，但它需要相当多细小的调整工作来使这些“自然而然”的输入参数（正

在被传递到外观）变为正确的格式以便smtplib发送消息：

```
def send_email(self, to_email, subject, message):
    if not           in username:
        from_email = "{0}@{1}".format(
                     self.username, self.host)
    else:
        from_email = self.username
    message = ("From: {0}\r\n"
               "To: {1}\r\n"
               "Subject: {2 }\r\n\r\n{ 3} •' ) • format (
                    from_email,
                    to_email,
                    subject,
                    message)

smtp = smtplib.SMTP(self.host)
smtp.login(self.username, self.password)
smtp.sendmail(from_email, [to_email], message)
```

方法开始的if语句决定username是否为邮件地址或者只是@符号左侧的部分；不同主机处理登录的细节不尽相同。最后，收取当前收件箱中消息的代码是非常混乱的；IMAP是一种被过度设计的糟糕协议，而imaplib标准库仅仅在该协议上覆盖了薄薄的一层抽象：

```
def get_inbox(self):
    mailbox = imaplib.IMAP4(self.host)
    mailbox.login(bytes(self.username, 'utf8 *),
        bytes(self.password, 'utf8'))
    mailbox.select()
    x, data = mailbox.search(None, * ALL')
    messages =[]
    for num in data[0].split():
        x, message = mailbox•fetch(num, '(RFC822)')
        messages.append(message[0][1])
    return messages
```

现在，如果我们把这些代码整合到一起，我们就有了一个简单的外观类，可以以一种

相当简洁的方法发送和接收消息，这比我们直接与复杂的类库进行交互要简单得多。

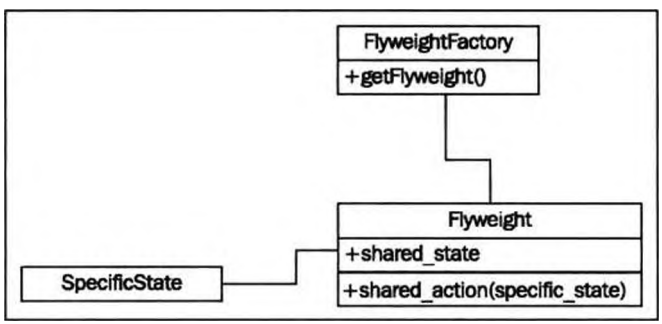
享元模式

享元模式是一种内存优化模式。Python的编程新手往往会忽略内存优化问题，以为内置的垃圾收集器会处理它们。这当然是完全可以接受的，但在开发拥有许多关联对象的大型应用程序时，关注内存是否充裕可以带来巨大的收益。

在现实生活中，经常是程序仅仅在暴露出内存问题后才想起实现享元模式。在某些情况下，从最初就设计优化配置是很有意义的，但记住，过早的优化能够非常有效地让你的程序极其复杂以至于难以维护。

享元模式背后的基本思想是，保证共享同一状态的对象可以同时使用该共享状态的内存。想象一下汽车销售的库存系统。每辆车都有一个特定的序列号与一种特定的颜色。但同一型号所有汽车的大部分细节都是相同的。例如，本田飞度的DX车型是一种具备少M-功能的最基本车型。LX车型多出了 A/C、倾斜传感器、巡航系统、电动车窗和门锁而运动款则带有花哨的轮毂、USB充电器和一个扰流板。（我最近买了飞度LX, 我非常喜爱这辆车！这就是为什么在许多章节我们都能看到汽车销售的例子）如果没有享元模式，每个单独的汽车对象都要存储一个长长的清单，包含它有什么功能，没有什么功能。考虑到本田汽车的年销量，全部统计会产生巨大的内存浪费。使用享元模式，我们可以使用共享对象存储与型号相关的特性列表，每辆车在拥有自己的序列号和颜色的同时只需要简单地对该型号进行引用。

让我们来看看享元模式的UML结构图：



每个享元都没有指定状态；在任何需要对特定状态执行操作的时候，该状态就需要被代码调用，传递到该享元处。一般来说，享元返回的工厂是一个单独的对象；它是为一个

能够识别享元的给定关键词而返回的。它的T. 作原理就像我们在第8章中讨论的单件模式那样；如果享元存在，我们返回它；否则，我们重新创建一个。在许多语言中，工厂不是作为一个单独的对象，而是作为Flyweight类本身的静态方法来执行。

这两种方式都行得通，但在Python中，享元工厂经常使用新的_new_构造函数来实现，类似于我们实现单件模式的方式。不同于单件的是，单件只需要返回类的一个实例，而享元则需要我们能够依赖于关键词返回不同的实例。我们可以将项目存储在字典中，并根据键值来查找他们。然而这种解决方案是有问题的，因为只要项目在字典中存在，它将一直保留在内存中。如果我们卖完了飞度LX车型，那么享元就不再被需要了，但它仍然会保留在字典中。当然，当我们卖出一辆车后，我们完全可以对它们进行清理，但这不正是垃圾收集器应该做的事吗？

Python的weakref模块可以帮助我们解决这个问题。该模块提供了一个WeakValueDictionary对象，这基本上可以让我们将项目存储在一个字典中，而不需要垃圾收集器来处理它们。如果某个值存在于一个弱引用的字典中，而且还没有任何其他对它的引用（就像我们卖完的LX车型），垃圾收集器将最终为我们把它处理掉。

让我们首先为这些汽车享元建立工厂：

```
import weakref

class CarModel:
    ^models = weakref.WeakValueDictionary()

    def —new—(els, model_name, *args, **kwargs):
        model = els._models.get(model_name)
        if not model:
            model = super().__new__(els)
            cls._models[model_name] = model
        return model
```

基本上，每当我们通过给定的名字构造一个新的享元时，我们首先在弱引用的字典中查找看看；如果它存在，我们就返回这个车型，如果没有，我们就创建一个新车型。无论采用哪种方式，享元函数都将被调用一次，不管它是一个新的还是已有的对象。因此，我们的_:^11:_可能看起来像是这样的：

```
    def —init__(self, model_name, air=False, tilt=FalsG,
                cruise-control=False, power_locks=False,
```

```
            alloy_wheels=False, usb_charger=False):
        if not hasattr(self, "initted"):
            self.model_name = model-name
            self.air = air
            self.tilt = tilt
            self•cruise-control = cruise_control
            self ,power__locks = power_locks
            self.alloy_wheels = alloy_wheels
            self.usb_charger = usb_charger
            self.initted=True
```

if语句确保当_init_函数第一次被调用时，我们才会初始化对象。这意味着我们在稍后通过车型名称调用工厂方法就可以获得相同的享元对象。然而，如果外部没有对它的引用存在，享元对象将被作为垃圾回收，所以我们要小心，不要意外地创建一个新的包含空值的享元。

让我们为我们的享元增加一个方法，假装查看某一特定车型的序列号，以确认此车是否出过任何事故。这个方法需要对不同车辆的不同序列号进行访问；序列号是不能以享元的形式存储的。因此.这个数据必须通过代码调用传递到这个方法中：

```
    def check_serial(self, serial-number>:
        print ("Sorry, we are unable to check •'
                •' the serial number {0} on the { 1 } "
                '' at this time" • format (
                    serial-number, self .model__name))
```

我们可以定义一个类来存储附加的信息，以及对享元的引用：

```
class Car:
    def —init_(self, model, color, serial):
        self.model = model
        self.color = color
        self.serial = serial

    def check-serial(self>:
        return self.model.check_serial(self.serial)
```

我们还可以跟踪记录可用的车型以及这种车型中的某辆汽车:

```
»> dx = CarModelC'FIT DXM)
»> lx = CarModel("FIT LX", air=True, cruise- control=True,
... power_locks=True, tilt=True)
»> carl = Car (dx, "blue", "12345")
»> car2 = Car(dx, "black", "12346")
»> car3 = Car (lx, "red", "12347")
```

现在，让我们来展示一下弱引用的工作：

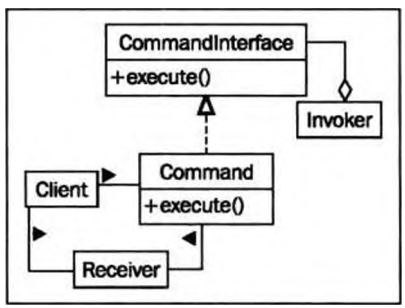
```
»> id (lx)
3071620300
»> del lx
»> del car3
»> import gc
»> gc. collect ()
0
»> lx = CarModel("FIT LX", air=True, cruise- control=True,
•…power- locks=True, tilt=True)
»> id (lx)
3071576140
»> lx = CarModel FIT LX")
»> id (lx)
3071576140
»> lx.air
True
```

id _数能够告诉我们某个对象的唯一标丨只符。当我们第2次调用它并删除所有对LX车型的引用，并强制进行垃圾收集之后，我们可以看到丨D已经发生了更改。在CarM_Odel_neW__T.厂字典中的值被删除了，一个新的对象被创建出来。然而，如果我们再试图构建第2个CarModel实例，它仍会返回相同的对象（丨D是相同的），并且，即使我们在第2次调用时没有提供任何参数，变量air仍将被设置为True。这意味着该对象第2次没有被初始化，就像我们设计的那样。

M而易见，采用享元模式可能比单纯将特性存储在某个汽车类中更加复杂。那我们应该在什么时候使用它呢？享元模式是专为节省内存而设计的；如果我们有数十或成百上千相似的对象，将它们的相同特性整合进一个享元中可以极大地减少对内存的消耗。

命令模式

命令模式在必须被完成的行为和调用这些动作（通常发生在一个稍晚时刻）的对象之间添加了一个抽象层。在命令模式中，客户端代码创建一个可以在日后执行的Command对象。当该命令在其上执行时，这个对象可以知道管理自己内部状态的接收器对象。Command对象实现一个特定的接口通常它都会有一个execute或do_action方法，并且还记录执行操作所需的任何参数$_3$最后，一个或多个Invoker对象在适当的时间执行命令。这里是它的UML结构图：



命令模式的一个常见例子是在图形窗口中的操作。通常情况下，一个操作可以被菜单栏的菜单项、快捷键、工具栏图标或右键菜单所调用。这些都是Invoker对象的实例$_a$实际发生的操作，例如Exit、Save或Copy, 全都由CommandInterface的命令实现$_a$GUr窗口接收到退出指令，文样接收到保存指令，ClipboardManager接收到复制命令，都是Receivers的例子。

让我们来实现一个提供Save和Exit操作的简单命令模式。我们将从一些最简单的接收者类开始：

```
import sys


class Window:
              def exit(self):
                  sys.exit(0)


class Document:
      def —init_____(self, filename):
            self.filename = filename
            self.contents = "This file cannot be modified"
```

```
    def save(self):
        with open(self.filename, 'w') as file:
            file.write(self.contents)
```

这些模拟的类模型对象在真实工作环境中需要做更多的工作。窗口需要处理鼠标移动和键盘事件，同时文件需要处理字符的插入、删除和选择。但对我们来说，这两个类将只做我们所需要的。

现在让我们来定义一些调用类。这些类将模仿工具栏、菜单和键盘等可能事件的发生；而实际上它们没有连接到任何东西，但我们可以看出它们是如何从命令、接收者和客户端代码中解耦的：

```
class ToolbarButton:
    def __init__(self, name, iconname):
        self.name = name
        self.iconname = iconname

    def click(self):
        self.command.execute()

class MenuItem:
    def __init__(self, menu_name, menuitem_name):
        self.menu = menu_name
        self.item = menuitem_name

    def click(self):
        self.command.execute()

class KeyboardShortcut:
    def __init__(self, key, modifier):
        self.key = key
        self.modifier = modifier

    def keypress(self):
        self.command.execute()
```

注意各种操作方法是如何通过各自的命令调用execute方法的。该命令实际上没有被设置到对象上；它们可以被传递给 init 函数，但因为它们可能被改变了（例如，

用一个可定制的按键组合编辑器），我们可以稍后在对象上设置属性。不同的情况需要不同的设计，但Python让程序员们能更灵活地做出对我们而言最有意义的事情。

现在，让我们把命令连接起来：

```
class SaveCommand:
    def __init__(self, document):
        self.document = document

    def execute(self):
        self.document.save()

class ExitCommand:
    def _init—(self, window):
        self.window = window
    def execute (self):
        self.window.exit()
```

这些命令是非常简单的；他们展示出了基本的模式，但需要特别注意的是，我们可以在必要时用命令存储状态和其他信息。例如，如果我们有一个插人字符的命令，我们可以为目前正在插人的字符保留状态。

我们现在要做的就是将一些客户端代码和测试代码连接，以使命令正常工作。作为基本测试，我们可以在脚本的末尾将如下代码包含进去：

```
window = Window()
document = Document("a_document.txtᴹ)
save = SaveCommand(document)
exit = ExitCommand(window)

save_button = ToolbarButton(ˌsave•,ˌsave.png')
save_button•command = save
save_keystroke = KeyboardShortcut("s", "ctrl">
save_keystroke.command = save
exit—menu = Menultem("File", "Exit")
exit_menu.command = exit
```

首先，我们需要创建两个接收者和两个命令。然后，我们再创建几个可用的调用者，并为他们每个都设置正确的命令。为了测试，我们可以使用python3 -i filename.py

并且运行类似于exit_menu.click()的代码，这将结束程序，或save_keystroke.keystroke()，这将保存伪控件。

然而，上面的例子感觉不是很符合Python风格，难道不是吗？它们有很多的“样板代码”（即不能完成任何事情，而只把结构提供给模式的代码），并且Command类彼此都是非常相似的。也许我们可以创建一个将函数作为回调的通用命令对象？

等一下，为什么要这么麻烦呢？或许我们可以对每个命令只使用一个函数或方法对象？和包含一个executed方法的对象不同，我们可以直接写一个函数，并使用它作为命令。在Python中，命令模式是一种常见的范式：

```
import sys


class Window:
    def exit(self):
        sys.exit(0)


class MenuItem:
    def click(self):
        self.command()


window = Window()
menu_item = MenuItem()
menu_item.command = window.exit
```

现在，它看起来更像是Python。乍一看，我们好像已经将所有命令模式的内容移除，而且我们已经将menuItem和Window类紧密地连接在一起。但仔细看看就能发现，它们根本就没有紧耦合。在MenuItem上，任何可调用的函数都可以像以前一样设置为命令。并且Window.exit方法附加至任意调用者。命令模式具备的灵活性大部分被保持了。我们为了可读性牺牲了完全解耦，但是这段代码在我和很多的Python程序员看来，比完全抽象的版本更易于维护。

当然，由于我们可以给任何对象添加一个__call__方法，我们也就不会被任何函数限制。当被调用的函数不需要维护状态时，上面的例子是一个有用的捷径，但在更高级的用法中，我们还可以使用下面这段代码：

```
class Document:
    def __init__(self, filename):
```

```
            self.filename = filename
            self.contents = "This file cannot be modified"

        def save(self):
            with open(self.filename, 'w*) as file:
                file•write(self•contents)

    class KeyboardShortcut:
        def keypress(self):
            self.command()

    class SaveCommand:
        def  init—(self, document):
            self.document = document

        def __call__(self):
            self.document.save()

    document = Document ("a__file. txt**)
    shortcut = KeyboardShortcut()
    save_command = SaveCommand(document)
    shortcut.command = save_command
```

在这里，有一些地方看起来和第一个命令模式相类似，但更具有Python风格。正如你所看到的那样，改变调用者去调用一个可调用的，而不是包含一个执行方法的命令对象并没有以任何方式限制我们；事实上，这可以给我们更多的灵活性。当它运行时，我们可以直接连接到函数，但当情况要求时，我们可以建立一个完整的可调用的命令对象。

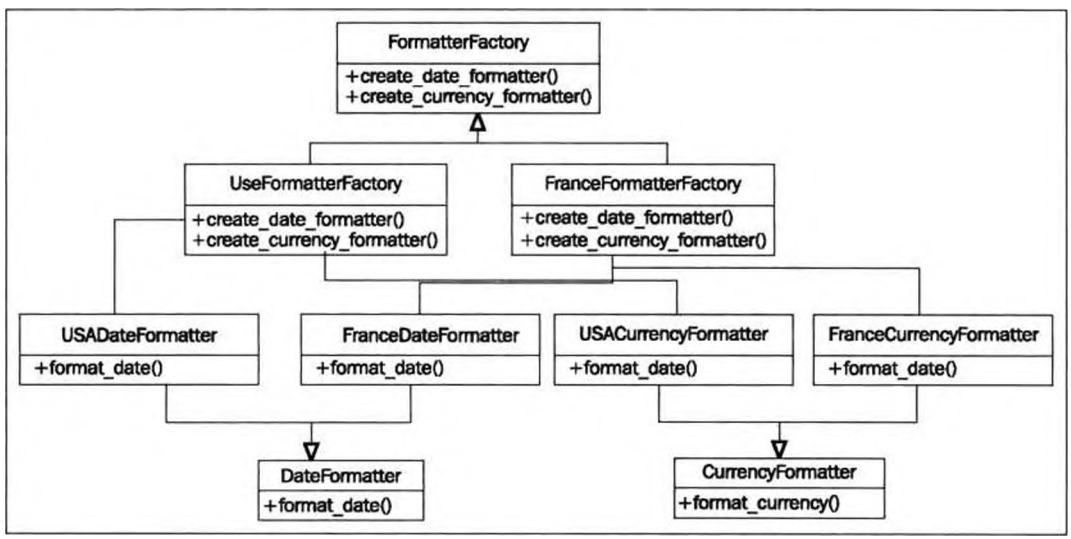
命令模式经常被扩展以支持可撤销的命令。例如，一个文本程序可以将闪人方法和删除插人的undo方法同时封装进一个execute命令。一段图形程序则可以将每个绘图操作（矩形、线条、写意像素等）和将像素恢复到原始状态的undo方法封装进一个命令包ˆ在这种情况下，命令模式的解耦显得更加有用，因为每个动作都需要维持足够的状态使得稍后能撤销该操作。

抽象工厂模式

抽象I厂模式一般用在我们的一个系统根据配置和平台的问题拥有多个可能实现的情况。调用代码从柚象V.厂中请求对象，但不知道哪个类的对象会被返回返回的底层实现可能取决于多种因素，如，前位置、操作系统或本地配置。

实际工作中常见的抽象T.厂模式的例子包括独立于操作系统的T.具包、数据库后端，以及针对具体_家的格式化或计算器代码。一个独立丁•操作系统的GUI T.具包可能会使用一个抽象T厂模式使它能在Windows下返回一套WinForm组件，在Mac下返回Cocoa组件，在Gnome下返回GTK控件以及在ICDE下返回QT控件。Django提供了一个抽象工厂根据当前站点的配置用来返回一组与类相关的对象来和指定的数据库后端（MySQL、PostgreSQL、SQLite和其他）进行交互$_3$如果应用程序被部署在多个地点，每个站点只需改变一个配置变量就可以使用不同的数据库后端。不同的国家有不同的系统来汁算税收、分类汇总以及零售商品总额；一个抽象工厂可以返回一个特定的计税对象。

如果没有一个具体的例子，抽象工厂模式的UML类图是很难理解的，让我们反过来，先创建一个具体的例子。我们可以创建一套依赖于特定语言环境的格式化工具来帮助我们格式化日期和货币。这将是可以挑选特定工厂的抽象工厂类，以及一对包含实际具体例子的厂厂，一个是法国，另一个是美闺。它们每一个都可以创建格式化的日期和时间对象，并且可以被奄询，格式化成指定的值。下面是它的结构图：



将上而简单的文字描述和这幅图比较一下，可以看出形象的_面并不总是胜过千言万

语，特别是考虑到我们甚至还没有在这里使用工厂来选择代码。

当然，在Python中，我们没有必要实现任何接口类，这样我们就可以弃用DateFormatter CurrencyFormatter 和 FormatterFactory。格式化类本身是非常简单的：

```
class FranceDateFormatter:
    def format_date(self, y, m, d):
        y, m, d = (str(x) for x in (y,m,d>)
        y = 120' + y if len(y) == 2 else y
        m = '0 *+ m if len(m) == 1 else m
        d = '0'+ d if len(d) == 1 else d
        return("{0}/{1}/{2}"•format(d,m,y)>
class USADateFormatter:
    def format_date(self, y, m, d):
        y, m, d = (str(x) for x in (y,m,d>)
        y = *20' + y if len(y) == 2 else y
        m = '0'+ m if len(m) == 1 else m
        d = '0'+ d if len(d) == 1 else d
        return ('• {0} - {1} - {2} " • f ormat (m, d, y))

class FranceCurrencyFormatter:
    def format—currency(self, base, cents):
        base, cents = (str(x) for x in (base, cents))
        if len(cents) == 0:
            cents = '00'
        elif len(cents) == 1:
            cents = •0* + cents

    digits =[]
    for i,c in enumerate(reversed(base)):
        if i and not i % 3:
            digits.append(1 ')
        digits.append(c)
    base = '*.join(reversed(digits))
    return "{o}€{ 1 }'*. format (base, cents)

class USACurrencyFormatter:
```

```
    def format_currency(self, base, cents):
        base, cents = (str(x) for x in (base, cents))
        if len (cents) == 0:
            cents = '00'
        elif len(cents) == 1:
            cents = '0' + cents

        digits =[]
        for i,c in enumerate(reversed(base)):
            if i and not i % 3:
                digits.append(',')
            digits.append(c)
            base = join(reversed(digits))
        return '${ 0 }.{ 1 format (base, cents)
```

这些类使用一些基本的字符串操作来把各种可能的输人（整数、不同长度的字符串及其他）变为如下格式：

	USA	France
Date	Mm-dd-yyyy	dd/mm/yyyy
Currency	$14,500.50	14 500€50

这里的代码可以对输人做更多的验证，但是不值得如此，让我们使这个例子保持简单！现在我们已经设置了格式化程序，我们只需要创建格式化工厂：

```
class USAFormatterFactory:
    def create_date_formatter(self):
        return USADateFormatter()
    def create_currency_formatter(self):
        return USACurrencyFormatter()

class FranceFormatterFactory:
    def create_date__formatter (self):
        return FranceDateFormatter()
    def create_currency_formatter(self):
        return FranceCurrencyFormatter()
```

现在，我们只需要创建可以挑选合适格式化工具的代码。由于这些事情只需要被创建一次，我们可以让它变成一个单件，只不过单件在Python中并不非常有用。那就让我们将

当前的格式化器变成模块级变S:

```
country_code = "US"
factory_map = {
        "US": USAFormatterFactory,
        "FR": FranceFormatterFactory}
formatter_factory = factory_map.get(country_code)()
```

在这个例子中.我们硬编码了当前国家代码；实际上，我们需要仔细考虑语言环境、操作系统或配置文件来选择代码它通过字典将国家代码与工厂类联系在一起。因此.我们只要从字典中选择正确的类并将它实例化就可以了。

当我们要为更多的国家添加支持时，可以很容易看出需要做什么事情；我们只需创建新的格式化类和抽象r厂本身。

抽象1:厂通常返同一个单件对象，虽然这是不必要的；在我们的代码中，每当它被调用时，都会返回格式化T具的一个新实例。格式化「.具没有理由不能被存储为实例变量并且每个丁.厂都要返回同一个实例。

回头再看看这些例子，我们再一次发现，似乎有很多适用于丁.厂的样板代码在Python似乎是不需要的。通常，每个工厂类沏（比如美国和法国）使用单独的模块都可以很容易地满足对抽象厂的需求，然后确保在工厂模块中，我们访问的是正确的模块。这种模块的封装结构通常是这样的：

```
localize/
    -init__.py
    backends/
        -init__.py
        USA.py
        France.py
```

这里的诀转是localize包中的_init_.py包含了将所有请求重定向到正确后端的代码。有多种方式可以这样做。显然，我们在后端模块中重复每个方法，然后通过_init_.py作为代理模块来进行路由选择。但我们可以做得更为整洁。如果我们知道后端永远不会动态地改变（即不会重新启动），我们可以简单地把一些if语句放到_init_.py巾来检查当前的国家代码，并在语句中使用通常是不可接受的from.backends.USA import★语法，从适当的后端导人所有变量。或者，我们可以将

它导人每一个后端，并设置currentjoackend变量指向一个特定的模块：

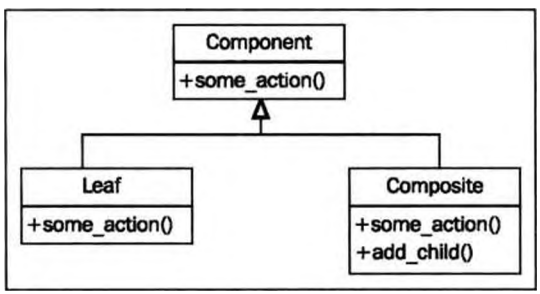
```
from .backends import USA, France

if country_code == "US":
    current_backend = USA
```

根据我们选择的解决方案，我们的客户端代码只需要简单地调用localize.format_date 或 localize .current_backend.format_date 来获得当前国家的格式化日期。最终的结果是比原来的抽象工厂模式更加Python化，而且在多数情况下同样灵活。

组合模式

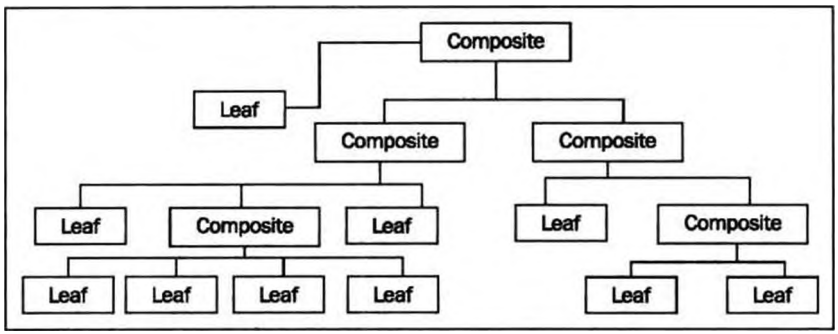
组合模式可以通过简单的组件构建复杂的树状结构。组合对象是简单的容器对象.容器中的内容则可能是另一个组合对象。

传统上讲，组合对象中的每个组件必须是一个叶节点（即不能包含其他对象）或组合节点。关键之处是组合节点和叶节点可以被同等对待。它的UML结构阁是很简单的：



然而，这个简单的模式却允许我们创建非常复杂的元素搭配并满足组件对象接口的要求。作为一个例子，这里有这样一个复杂的元素搭配：

组合模式在文件/文件夹树中很有用。无论在树中的节点是一个正常的文件还是文件夹,它都可以进行诸如移动、复制或删除节点的操作。我们可以创建一个支持这些操作的组件接口，然后使用组合对象来表示文件夹，用叶节点表示普通文件。



当然，在Python中，我们可以又一次利用鸭子类型来隐式地提供接口，因此我们只需要编写两个类。让我们先来定义这些接口 ：

```
class Folder:
    def —init—(self, name):
        self.name = name
        self.children = {}

    def add-child(self, child):
        pass

    def move (self, new__path):
        pass

    def copy(self, new-path}:
        pass

    def delete(self):
        pass

class File:
    def __init__(self, name, contents):
        self.name = name
        self.contents = contents

    def move(self, new_path):
        pass
```

```
    def copy(self, new-path):
        pass

    def delete(self):
        pass
```

对于每个文件夹（组合）对象，我们可以保留一个子节点的字典。通常，一个列表就足够了，但在这种情况下，字典将是按照名字查找子节点的有效方式。我们的路径将被指定为节点名，并通过“/”字符分隔，这类似于UNIX shell中的路径。

思考一下涉及的方法，我们就可以看出，不论是一个文件还是文件夹节点，移动或删除它的方式都是相似的。但是，复制文件夹节点时必须要进行递归复制，而复制文件节点则只是一个很小的操作。

为了充分利用相似的操作，让我们将一些常用的方法提取到一个父类中。让我们丢弃Component接口，并将它变为基类：

```
class Component:
    def _____init_(self, name):
        self.name = name

    def move (self, new__path):
        new_folder =ⁱget_path (new_path)
        del self.parent.children[self.name]
        new_folder.children[self.name] = self
        self.parent = new_folder

    def delete(self):
        del self.parent.children[self.name]

class Folder(Component):
    def —init________(self, name):
        super().______ init_____ (name)
        self.children = {}

    def add_child(self, child):
        pass

    def copy(self, new_path):
```

```
            pass


class File(Component):
    def __init__(self, name, contents):
        super().__init__(name)
        self.contents = contents

    def copy(self, new_path):
        pass


root = Folder()
def get_path(path):
    names = path.split('/')[1:]
    node = root
    for name in names:
        node = node.children[name]
    return node
```

我们在这里的Component类中创建了 move和delete方法。他们都要访问一个我们还没有设置的神秘变fl parent。move方法使用一个模块级的get_path函数从预定义的根节点中通过给定的路径找出一个节点。所有的文件都将被添加到该根节点或它的子节点。对于move方法，目标应该是当前存在的文件夹，否则我们将会出现错误。如同这本书中的许多例子一样，没有进行错误处理，这帮助我们将考虑的精力集中于原理。

让我们先设置这个神秘的parent变量；当然，是在文件夹对象的add_child方法中进行：

```
        def add_child(self, child):
            child.parent = self
            self.children[child.name] = child
```

嗯，这是很简单的。现在让我们来看看组合文件层次结构是否在正常丁.作:

```
$ python3 -i 1261_09_18_add_child.py

>>> folderl = Folder ('folderl')
>>> folder2 = Folder ('folder2')
>>> root.add_child(folder1)
>>> root.add_child(folder2)
```

```
»> folderll = Folder ('folder 11¹)
»> f older 1 .add child (folderll)
»> filelll = FileCfilelll', ¹ contents')
>» folderll .add child(filelll)
»> £ile21 = File (' file21' , ' other contents ')
»> f older2 . add child (file21)
»> folder2 . children
{'£ile21': < main____.File object at 0xb7220a4c>}
»> £older2 .move (' /folderl/folderll')
»> folderll. children
{ ' £older2 ' : < main______.Folder object at 0xb722080c>, 'filelll*: < main__. File
object at 0xb72209ec>)
»> £ile21 .move (' /folderl')
»> folderl. children
{'£ile21¹: < main____.File object at 0xb7220a4c>, * folderll': < main____ .Folder
object at 0xb722084c>)
>»
```

是的，我们可以创建文件夹，将文件夹添加到其他文件夹，将文件添加到文件夹并移动它们！在文件层次结构中，我们还能做什么呢？

嗯，我们可以实现复制操作，但为了节省篇幅，我们把这个作为练习！

组合模式对这样的树状结构是极为有用的，包含图形用户界面窗口的小部件层次结构、文件层次结构、树集、图形和HTML DOM。在Python中，按照传统方式来实现的组合模式是非常有用的，正如先前的例子演示的那样。有时候，如果正在创建的只是一棵很浅的树，我们可以不使用嵌套列表或嵌套字典，并且不需要像我们先前做的那样来执行自定义组件，叶节点和组合类。其他一些时候，我们也可以只实现一个组合类，并把叶节点和组合对象视为同一个类=> 另外，Python的鸭子类型可以很容易地将其他对象添加到组合层次中，只要它们有正确的接丨]。

练习

在深人练习每个设计模式前，先花一点时间实现前几章中File和Folder对象的copy方法。File方法相肖简单；就是创建一个具有相同名称和内容的新节点，并将其添加到新的父义件夹中3 Folder中的copy方法就复杂一些，因为你首先需要对文件夹进

行复制，然后递归地复制它的每一个子类到新的位置。你可以随意调用子类的copy（）方法，不用在意它是文件还是文件夹对象。这足以让我们理解组合模式有多么强大。

现在，和上一章一样，看看我们已经讨论过的模式，并考虑实现它们的理想场所。你可能希望在已有的代码中应用适配器模式，因为它更适合应用于现有库之间的连接，而不是新的代码。你将如何使用一个适配器使得两个接口可以正常交流？

你能想到一个足够复杂，需要使用外观模式的系统吗？考虑一下现实生活中用到的外观，比如汽车的前窗接口或者工厂的控制面板。软件其实是类似的，所不同的只是外观接口的用户是其他程序员，而不是被培训过可以使用它们的工人。你的最新项目中是否有复杂系统可以受益于外观模式？

就算你现在没有任何巨大的、内存消耗严重的代码可以受益于享元模式，但你能想到它可能起作用的场景吗？任何需要处理大量重复数据的地方都可以使用享元模式。它在银行业有用吗？在网页程序中呢？在什么时候使用享元模式会比较合理？什么时候又会显得过度？

命令模式又是什么样的？你能想到一些常见的（或者更好一些，不常见的）调用解耦操作可以起作用的例子吗？看看你每天使用的程序，并想想他们在内部是如何实现的。他们中的很多很可能因为某些原因或目的在使用命令模式。

抽象工厂模式，或者我们所讨论过的某些更具Python风格的产物，对于创建一键配置式系统是非常有用的。你能想到这样的系统可以用在什么地方吗？

最后，我们来说说组合模式。我们在编程中到处会遇到树状结构；它们中的一些，就像我们的文件层次的例子那样，是非常显而易见的；有些则是相当不易察觉。这些组合模式可以用于什么样的情况？你能想到在自己代码中使用它们的地方吗？如果你稍微调整一下模式，例如，将不同类型的叶节点或组合节点封装进不同类型的对象肖中，那又会怎样？

总结

在本章中，我们仔细讨论了另外几种设计模式，介绍了它们的规范描述以及在Python中实现它们的另一种方法，往往比传统的面向对象编程语言更加灵活和通用。我们还特别讨论了：

- 匹配接口的适配器模式。
- 简化复杂系统的外观模式。
- 减少内存消耗的享元模式。

- 隔离调用程序的命令模式。
- 分开实现的抽象|:厂模式。
- 树状结构的组合模式。

在下一章，我们将介绍一些用于操作文件、配置和进程的常见工具。

10

第10章 文件和字符串

现在，让我们从高级模式退回一步，来看看在我们用过的所有例子中的几个Python结构，但不会涉及任何细节。我们将会关注文件、I/O、序列化以及数据加载。在这一过程中，我们将会发现Python中过度简单的字符串其实有多么复杂。特别是，我们将会看到：

- 字符串、字节、字节数组的复杂性。
- 字符串格式化的来龙去脉。
- 如何打开文件。
- 上下文管理。
- 几种序列化数据的方法。

字符串

字符串是Python中的一个基本体，到目前为止，几乎在我们已经讨论过的每个例子中，我们都使用了它。它所做的不过是代表一个不变的字符序列。当然，“字符”是一个有点模棱两可的词，Python中的字符串可以代表重音字符序列吗？汉字呢？再或是希腊文、斯拉夫文、波斯文呢？

在Python 3中，答案是肯定的。Python字符串都是通过Unicode来展示的，而Unicode是一种特殊的字符定义，它可以表示几乎在这个星球上任何语言的任何字符。在大多数情况下，它做得真的天衣无缝。现在，让我们认定Python 3的字符串都是一个不可变的Unicode字符序列。

所以，对于不可变的序列我们能做些什么呢？在前面的例子中我们已经接触到了许多

字符串操作方法。但是，还是让我们快速地将一切都放在一起，来一个字符串理论速成课！

字符串操作

如你所知，Python中的字符串可以通过用单引号或双引号包装字符序列来创建。而多行字符串也可以很容易地使用3个引号来创建。多个硬编码字符串可以通过把它们挨着个排在一起来将它们连接起来（对于在函数调用中放置超过你的编辑器文本宽度限制的长字符串，这是非常有用的）。下面就是一些例子：

```
a = "hello"
b = " world "
c = '1 " a multiple
line string " 1'
d =, "."More
multiple"""
e = (' ' Three " "Strings "
            '.Together,')
```

以上，可以说是创建字符串相关的所有东西了。最后一个字符串会自动地由解释器组合成一个字符串。当然，使用“+”操作符连接字符串也是可以的（如"hello " + "world"）。

当然，字符串并不必须硬编码，它们也可以来自不同的外部源，比如文本文件、用户输入或是网络。

像其他序列一样，字符串也是可遍历（通过字符）、可切片及可连接的。语法与列表是一样的。

str类中有很多简化字符串操作的方法。Python解释器中的dir和help命令可以告诉我们如何使用它们。在这里，我们会直接讨论其中一些比较常见的。

有几个方便的布尔型方法可以帮助我们确定字符串中的字符是否与某个特定模式匹配。下面是这些方法的总结：

方法	目的
Isalpha	当字符串中的字符全部都是某些语言中的字母字符时，返回True。当有任何空格、标点或数字出现在字符串中时，返回False
isdigit isdecimal isnumeric	这些方法分别告诉我们，字符串中的字符是否全部都是数字 Unicode十进制字符或Unicode数值集请小心，这几个函数中，点字符并不属于十进制字符，所以1 45 • 2 1 • isdecimal（>将返回False!对于45.2,用Unicode表示真正的十进制字符值成是0660

续表

方法	目的
isalnum	如果字符串中只包含字母或数字字符，将返回True;如果包含任何的标点或空白，将返回False
isspace	如果字符串中只包含空白字符（空格、制表位、新的一行等），将返回True;否则返回False
isupper islower	根据调用的方法，如果字符串中的字母字符全部都是大写或小写时，返回True
istitle	如果字符串是个标题，则返回True。是个标题意味着每个单词都只有第一个字符是大写的，而其他字符都是小写的。请小心这一点，因为它并不严格符合语法定义的标题格式。例如，诗歌“The Glove and the Lions”是一个有效的标题，尽管并不是所有的单词都大写“The Cremation Of Sam McGee”也是一个有效的标题，尽管在最后一个单词中，中间有一个大写字母
isidentifier	如果字符串的值可以用作Python的变量名，则返回True。如果它含有空格、连字符或以数字开头，将会返回False
isprintable	如果字符串中的所有字符都可以在屏幕或打印机上打印.则返回True。不包括终端控制字符，如退出键。值得注意的是，空白字符是可以打印的，即便它们在印刷时是不可见的

其他的方法可以用来判断一个字符串是否匹配一个特定的模式。当字符串以传入函数的字符串开始或结束时，startswith和endswith方法返回True。count方法告诉我们一个给定的子串在字符串出现的次数，同时find、index、rfind和rindex则告诉我们给定的子串在原始字符串中的位置。这两个’r•(表示“右”）方法会从字符串结尾的位置开始搜索。如果找不到子串，find方法会返回-1，而index对于这种情况则会抛出一个ValueError。让我们看一看这些方法中的一部分是怎么工作的：

```
»> s = "hello world"
»> s.startswith('hi')
False
»> s . endswith (' Id *)
True
»> s.findri')
2
»> s . index (' m')
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
»〉
```

剩下的字符串方法中，很多都会返回对字符串的转换。upper、lower capitalize和title方法会将字母字符以给定的格式创建一个新的字符串。translate方法使用一个字典将任意字符输人映射为特定的输出。

这些方法中，每个方法都需要注意输人字符串仍然是不变的，相反会返回一个崭新的str实例。如果我们需要操作生成的字符串，我们应该将它分配给一个新的变量，如new_value = value. capitalize ()。通常，一旦我们执行了转换，我们就不再需要旧的值了，所以通常的惯例是将其分配给同一个变量，如value = value.title

最后，某些字符串方法返回或操作的是列表。split方法接收一个子字符串，并在每个子串出现的地方将字符串分裂成一个字符串列表的元素。partition方法只在第一个子串出现的地方，将字符串分隔开，并返回一个3个值的元组：子串前的字符、子串自身和子串后的字符。

作为split的操作，join方法接收一个字符串列表，返回一个结合了全部字符串的列表，同时将原字符串置于每个字符串之间。replace方法接收两个参数，返回一个字符串，其中返回字符串中每个出现第1个参数实例的地方都被第2个参数所取代。下面是其中一些方法的运行情况：

```
»> s = "hello world, how are you'1
»> s2 = s • split (′ • >
»> s2
[|hello, , 'world,', 'how', 'are', , you, ]
»> ' # ' . join (s2)
'hello#world,#how#are#you'
»> s . replace (• * , ' ** ')
′ hello**world,**how**are**you1
»> s . partition ('')
(′hello', 1 •, 'world, how are you')
```

就是这样，我们对str类中最常见的方法走马观花了一番！现在，让我们看看Python 3中将字符串和变量组合在一起成为新字符串的方法。

字符串格式化

Python 3有一个强大的字符串格式化机制，让我们可以很容易地构造由硬编码文本和分散的变量组成的字符串。在先前的很多例子中，我们已经使用过它了，但比起我们用过的简单的格式化符，其实它还有更广泛的用途。

任何字符串都可以通过调用它自己的formats方法变成一个格式字符串。该方法会返回一个新的字符串。在新字符串中，输入字符串中的特定字符会被传入函数的参数和关键字参数提供的值所代替。format方法没有固定的参数集，在内部，它使用了我们在第7章中讨论过的*args和**kwargs语法。

在格式化的字符串中，被替换的特殊字符是左花括号和右花括号：{和}。我们可以在一个字符串中插人多对这些括号，它们将会按照传入str. format方法的任何位置参数有序地被替换掉：

```
template = "Hello {}, you are currently {}."
print(template.format('Dusty',            'writing *))
```

如果我们运行这些语句，变s：就会有序地替换掉花括号：

```
Hello Dusty, you are currently writing.
```

这个基本语法在我们想要在一个字符串中重用变虽，或决定在不同位置使用它们时，并不是特别有用。这时，我们可以将从〇起的整数索引放在花括号中，用来告诉格式化器变M应该在字符串中的哪个位置被插入。下面，让我们重复一次名字：

```
template = "Hello {0}, you are {1}. Your name is {0}.' '
print(template.format('Dusty',            'writing'))
```

如果使用这些整数索引，我们必须对所有的变M都使用它们。我们不能混合使用空括号和位置索引。例如，下面这个代码就会失败：

```
template = "Hello {}, you are {}. Your name is {0}."
print(template.format('Dusty1,            * writing *))
```

运行这段将会抛出一个恰当的异常：

```
Traceback (most recent call last):
  File "1261_10_04_format_some_positions-broken.py", line 2, in
<module>
     print(template.format(1 Dusty',            'writing1))
```

```
ValueError: cannot switch from automatic field numbering to manual field
specification
```

避免花括号

除了格式化之外，花括号在字符串中也经常会用到。因此，我们需要一种方法来避免在我们希望它们就显示为自己的情况下被替换掉。这可以通过再增加一个括号来实现。例如，我们可以使用Python来格式化基本的Java程序：

```
template =
public class {0} {{
    public static void main(String[] args) {{
        System.out.printIn({1});
    } }
|^dMII


print(template.format("MyClass", "print('hello world')"));
```

无论任何时候我们在模板中看到{{或}}序列，即上例中封闭Java类和方法定义的括号，我们知道format方法将会用单个括号取代它们，而不是用传人格式化方法的某个参数。下面就是输出：

```
public class MyClass {
    public static void main(String[] args) {
        System.out.printIn《"print(1 hello world*)">;
```

输出中，类的名称和内容被两个参数所替换，而双括号只是被换成单括号，从而给了我们一个布效的Java文件。完成！这就是一个尽可能简单的Python程序，用来打印一个尽可能简单的Java程序，而这个java则用来打印尽可能简单的Python程序！

关键字参数

如果我们想要格式化复杂的字符串，记住参数的顺序或如果我们想在列表的开始处插人另一个参数来更新模板，这都是非常冗长乏味的。因此，format方法允许我们在括号内指定名字来代替数字。然后，将对应名字变量的值会以关键字参数的形式提供给format

方法，而不再是位置参数:

```
template = "•••
From: <{from_email}>
To: <{to_email}>
Subject: {subject}
{message}*""'
print(template.format(
    from_email = "a@example.com",
    to_email = "b@example.com",
    message = "Here's some mail for you."
    "Hope you enjoy the message!",
    subject = "You have mail!"
    ))
```

3—/ 注意Python是怎样自动将我们传入的两个作为消息参数的字符串连接起来的？这对行包装非常有用，但请小心不要在文件的顶级做这个.，你只需要在一组括号里边（无论是组成元组、函数调用还是只是显式地对几个在一起的字符串进行分组）来启用这种连接.：

我们也可以混合索引和关键字参数（与所有的Python函数调用一样，关键字参数必须在位置参数之后〉。我们甚至还可以混合无标签的位置括号和关键字参数：

```
print('M) {label} {}".format("x", "y", label="zn))
```

正如所料，这段代码会输出：

```
x z y
```

容器查询

传人format方法的并不限制为只是简单的字符串变ih任何原语，如整数或浮点数都可以被打印出来更有趣的是，包括列表、元组、字典和任意对象在内的复杂对象也都4以被使用。并且我们可以在format字符串内部访问那些对象内部的索引和变量（但不能是方法)。

例如，如果我们的邮件消息已经根据由寄件人和收件人的邮箱地址组成的元组进行了分组，并且主题和消息也因为某些原因被放人了一个字典（也许是因为我们想要使用的一

个已经存在的send_mail函数需要这样的输人），那么我们可以像下而这样做格式化：

```
emails = ("a@example.com", "b@example.com")
message = {
        •subject*: "You Have Mail!",
        'message': "Here's some mail for you!"
        }
template = f"""
From: <{0[0]}>
To: <{0[1] }>
Subject: {message[subject]}
{message [message]
print(template.format(emails, message=message))
```

模板字符串中，括号内的变量看起来有点奇怪，所以让我们看看它们到底在做什么。我们已经传人了一个基于基于的位置的参数和一个关键字参数。两个电子邮件地址通过0[x]来查询，其中x可以是0或1。最初的零表示传人format的第一个位置参数（本例中就是emails元组），其他的基于位置的参数也一样。有数字在内的方括号和我们在常规Python代码中见到的索引嗇找是一样的，听以0[0]会映射到emails元组中的emails [0]。这种索引语法适用于任意的可索引对象，所以当我们访问message [subject]时可以看到类似的行为，除了这次我们在字典中查找一个字符串键的情况。注意，与Python代码不同的是，我们不需要在字典查找时，将字符串放在丨号中。

我们甚至可以做多级查找，如果我们有嵌套的数据结构。但我不推荐经常这样做，因为这样模板字符串会迅速变得难以现解。如果我们有一个包含元组的字典，我们可以这样做：

```
emails = ("a@example.com", "b@example.com")
message = {
        •emails*: emails,
        'subject *: "You Have Mail!",
        'message': "Here's some mail for you!
        }
template ="""
From: <{0[emails][0]}>
To: <{0[emails] [1 ] }>
Subject: {0[subject]}
```

```
{0[message])·
print(template.format(message))
```

在这个例子中，我们只访问一个变fi,在零位置上；为了在字典中查找电子邮件地址的值，我们在该位置上做了第2个索引查找。格式化索引体系是非常灵活的，但我们必须i己住，我们承担的主要目标是使我们的代码，包括模板的可读性尽可能地好。我们需要查看传人到模板中的变量，以及我们如何在其内部访问它们。将它们分解成单独的位置或关键字参数后再交给格式化方法，而不是传递一个单独的大对象是不是更好呢？这取决于对象需要从什么样的结构开始，以及在格式化字符串中变量内容的结构和数量。

对象查找

索引使format查找十分强大，但我们还没有完全讲完！我们还可以将任意对象作为参数传入，并使用点符号查找这些对象中的属性。让我们再一次改变我们的邮件数据，这次使用一个类：

```
class EMail:
      def —init—(self, from_addr, to_addr, subject, message):
            self.from_addr = from_addr
            self.to^addr = to_addr
            self.subject = subject
            self.message = message

email = EMail("a@example·com", "b@example.com",
            "You Have Mail!",
            "Here's some mail for you!")

template =
From: <{0.from_addr}>
To: <{0.to_addr}>
Subject: {0.subject}

{ 0 .message}
print(template.format(email))
```

本例中的模板比前面例子中的更易读，但额外的开销是创建了一个电子邮件类从而增加了 Python代码的复杂性我们通常不会为表达将对象包括在模板中这样的目的而创建一

个类。通常情况下，如果我们试图格式化的对象已经存在，我们才会使用这种方式的查找，这对于所有例子都是真实的，如果已经有了元组、列表或字典，也许我们会将它直接传人模板，否则我们就只是创建一个简单的位置参数和关键字参数的集合。

让它看起来是对的

能够在模板字符串中包含变量是非常好的，但有时变量需要附加一点强迫才能让它们的输出看起来是正确的。例如，如果我们正在做货币相关的计算，最终可能会得到一个很长的小数，但我们不想让它出现在我们的模板中：

```
subtotal = 12.32
tax = subtotal * 0.07
total = subtotal + tax

print("Sub: ${0} Tax: ${1} Total: ${total}".format(
      subtotal, tax, total=total))
```

如果我们运行这段格式化代码，它的输出并不完全像正确的货币形式:

```
Sub: $12.32 Tax: $0.8624 Total: $13.1824
```

> 从技术上讲，我们不应该在货币计算中使用浮点型数字，而是应该构造一个decimal. Decimal ()对象来代替。由于浮点计算固有的不准确超过了需要的精度水平，所以它们会比较危险。但我们看到的是字符串，而不是浮点数，所以货币对于格式化来说是一个很好的例子！

为了修改上面的format字符串，我们可以在花括号里面包括一些额外的信息用来调整参数的格式。我们可以定制的东西有很多，但括号内的基本语法是相同的；首先，我们使用先前的布局方式中（位置、关键字、索引、属性访问）任意一种适合的，来指定我们想在模板字符串中放置的变M,再在其后跟一个冒号，然后是格式化的特定语法。下面是改进的版本：

```
subtotal = 12.32
tax = subtotal * 0.07
total = subtotal + tax

print("Sub: ${0:0.2f} Tax: ${1:0.2f)"
```

```
            "Total: ${total:0.2f}".format(
                subtotal, tax, total=total))
```

冒号后的格式化符0.2f，从左到右，每一位主要表示：如果值小于1,则确保0可以显示在小数点左边；显示小数后两位；将输人的值格式化为浮点数。

我们也可以通过在精度的句点前放置一个值来指定每个数字应该在屏幕上占据字符的特定数量。这对输出表格数据非常有用，例如：

```
orders = [('burger', 2, 5),
          ('fries', 3.5, 1),
          ('cola', 1.75, 3)]

print("PRODUCT QUANTITY PRICE SUBTOTAL")
for product, price, quantity in orders:
    subtotal = price * quantity
    print('{0:10s}{1:^9d} ${2:<8.2f}${3:>7.2f}'.format(
        product, quantity, price, subtotal))
```

好，这真是一个相当可怕的格式字符串，所以在我们把它分解为可理解的部分之前，让我们看看它是如何工作的：

```
PRODUCT          QUANTITY    PRICE       SUBTOTAL
burger              5        $2.00     $   10.00
fries               1        $3.50     $    3.50
cola                3        $1.75     $    5.25
```

漂亮！这究竟是如何发生的？在for循环的每行中，我们要格式化4个变量。

第一个变量是一个字符串，以{0:10s}格式化。s指一个字符串变量，10指它将占据10个字符。默认情况下，对于字符串，如果它比指定的字符数短，那么它将在字符串的右边附加空格来使它足够长（但是注意：如果原始字符串过长，它并不会被截断！）。我们可以改变这种行为（用其他字符填满，或改变格式字符串的对齐方式），就像我们对下面的值所做的，quantity。

格式化器中quantity的值为{1:^d}代表是个整数值，9告诉我们值应该占据9个字符。但对于整数，默认情况下，填充的额外字符是0,而不是空格。这看起来有些奇怪。所以我们可以显式地指定用一个空格（紧跟冒号）作为填充字符。克拉符^告诉我们有多少这种可用填充字符应该被中心对齐；这会让列看起来更专业一点。尽管说明符都

是可选的，但它们必须按照正确的顺序排列：先是填充字符，然后对齐，然后是大小，最后是类型。

我们对价格(price)和小结(subtotal M故了类似的事情。对于price，我们用{2 : <8.2f}，对于subtotal, 使用{3: >7.2f}。对这两个案例，我们都指定一个空格作为填充字符，但我们分别使用<和>符号来表示应该左对齐或右对齐的最小空格字符数为8或7。此外，每个浮点数都应格式化为两位小数。

不同类型的“类型”字符也会影响格式化输出。我们已经见过了 s、d和f类型，分别对应字符串、整数、浮点数。大多数其他的格式说明符都是它们的替代版本。例如，O表示八进制而X表示十六进制整数。类型说明符n可能对当前区域格式下格式化整数分隔符是非常有用的。对于浮点数.%类型将乘以丨〇〇从而将浮点数格式化为百分比。

我们只触及了字符串格式的表面。这些标准格式化器适用于大多数的内置对象，但实际上，其他对象也可以定义非标准的说明符。例如，如果我们将一个datetime对象传入format, 我们就可以在datetime, strftime函数中使用说明符来格式化它，如下：

```
import datetime
print("{0:%I:%M%p}".format(
      datetime.datetime.now()))
```

我们甚至可以为我们自己创造的对象来编写自定义的格式化器，但这超出了本书的范围。如果你需要在你的代码中这样做，看看重写特殊的_f〇rmat_方法[3]最全面的说明可以在位于http: //www.python. org/dev/peps/pep-3101/although 的 PEP 3101 中找到.尽管细节会有点枯燥。你也可以通过网页搜索找到更易于理解的学习材料。

字符串是Unicode的

在本节的开始，我们将字符串定义为不变的Unicode字符集合^实际上在当时这让事情变得非常复杂了，因为Unicode并不是一个真正的存储格式。例如，如果你从一个文件或套接字中得到了一个字节字符串，它们不会是Unicode的。事实上，它们将是内置类型bytes（字节）。字节是不可变的序列……嗯，字节。字节是计算中的最低级别存储格式。它们代表8比特，并通常以一个0 ~ 255之间的整数，或等效于十六进制的0 ~ FF之间的数字来描述。字节并不特别代表任何东西，一个字节序列可能存储的是一个编码字符串的字符，也可能是一张图片的像素。

如果我们打印一个字节对象，任何映射能够到ASCII表示的字节将打印为它们原始的字符，而非ASCII的字节（无论它们是二进制数据还是其他字符）都会打印为以\x分隔的

十六进制代码序列。你可能会对一个表示为整数的字节可以映射成一个ASCII字符感到奇怪。但ASCII实际上只是一个代码，每个字母都会被不同的字节模式所表示，因此可以对应成不同的整数。例如字符“a”就和数字97,也就是十六进制数0x61,由同样的字节所表示。特别地，这些都是对二进制模式01100001的一种解释。

许多I/O操作只知道如何处理bytes,即使字节对象指向的是文本数据:> 因此，知道如何在bytes和Unicode之间转换就变得非常重要了。

不过问题是，有很多方法可以映射bytes为Unicode文本。字节是机器可读的值，而文本是人类可读的格式。在两者之间的是一种编码，可以将给定的字节序列映射为一个给定的文本字符序列。

然而，有多种这样的编码（ASCII只是其中一种）。使用不同的编码，相同的字节序列可以表示为完全不同的文本字符！所以解码bytes时，使用与编码时相同的字符集就非常重要了，否则我们得到的就是垃圾数据。不知道字节是怎样被编码的，就不可能从字节中得到文本。如果收到了没有特定编码的未知字节，我们最多只能是猜测它的编码格式.而且很可能我们猜错了。

将字节转换为文本

如果我们从某个地方得到一个bytes数组，我们可以使用bytes类中的.decode方法将它转换为Unicode:该方法接收一个字符串作为参数，这个字符串即字符编码的名字。有许多这样的名字，对于欧洲语种来说比较常用的包括ASCII、UTF-8和丨atin-1。

字节序列（十六进制）63 6c 69 63 68 e9代表的实际上是单词click用latin-丨编码的字符。下面的例子可以将这个字节序列编码，并使用丨atin-丨编码将其转换为Unicode字符串：

```
characters = b'\x63\x6c\x69\x63\x68\xe9'
print(characters)
print(characters.decode("latin-1"))
```

第1行创建了一个bytes对象，紧跟字符串前的b告诉我们，我们定义的是bytes对象，而不是正常的Unicode字符串。本例中，字符串中的每个字节都被规定为一个十六进制数。字节字符串中的每一个\x分隔符表明，“接下来的两个字符表示一个使用十六进制数字的字节”。

假设我们正在使用一个能够理解丨atin-1编码的shell,那么两个打印语句就会输出以下字符串：

```
b'clich\xe9'
cliche
```

第1个打印语句使所有的ASCII字符的字节都显示为它们对应的字符。未知的（对ASCII来说是未知的）字符则依旧保持它的十六进制分隔格式。输出中，在行首还包括一个b, 来提醒我们这是一个bytes的表示，而不是一个字符串。

下一个语句使用latin-1解码了字符串，解码方法返回了一个含有正确字符的正常（Unicode)字符串。假如我们用斯拉夫字母的’’ is$_{\bigcirc}$8859-5″编码方法解码相同的字符串，我们会最终得到字符序列klichm’！这是因S\xe9字节在两种编码中映射到了不同的两个字符上。

将文本转换为字节

如果我们需要将输入字节转换为Unicode，显然我们也会遇到需要将输出的Unicode转换成字节序列的情况。这可以通过str类中，与decode相似的encode方法完成，它也需要一个字符集。下面的代码创建了一个Unicode字符串并用了几个不同的字符集对它进行编码：

```
characters = "cliche"
print(characters.encode("UTF-8")>
print (characters .encode ("latin-l'*))
print(characters.encode("CP437")>
print(characters.encode(nascii"))
```

前3个编码对重音字符创建了一组不同的字节，而第4个甚至不能处理它：

```
b * clich\xc3\xa9[1]
b‘clich\xe9‘
b‘clich\x82‘
Traceback (most recent call last):
  File "1261_10_16_decode_unicode.py", line 5, in <module>
    print (characters. encode ("ascii[1]*))
UnicodeEncodeError: * asciif codec can't encode character [1]\xe9' in
position 5: ordinal not in range(128)
```

你现在理解了编码的重要性了吗？重音字符是在不同的编码中表现为不同的字符；当我们解码字节到文本时，如果我们用错了编码只会得到错误的字符。

最后一例中的异常并不总是想要得到的行为，可能有些情况下，我们希望未知字符用另一种不同的方式来处理。encode方法接收一个名为errors的可选字符串参数，它可以定义应该如何处理这样的字符。这个字符串可以是如下之一：

- strict
- replace
- ignore
- xmlcharrefreplace

我们刚刚看到的是默认的strict替换策略，当遇到一个在请求编码中没有有效表示的字节序列时，就会抛出一个异常。当使用replace策略时，字符会被替换为不同的字符。在ASCI丨中是一个问号，其他编码中可能使用不同的符号，比如一个空方块。ignore策略会简单地丢弃任何不能理解的字节，而xmlcharref replace策略则会创建一个代表这个Unicode字符的xml实体。它可以在使用XML文档转换未知字符串时用到。下面是每个策略是如何影响我们的样例同的：

策略	"cliche".encode("ascii", strategy)
replace	b'clich?'
ignore	b'clich'
xmlcharrefreplace	b'cliché'[1]

不传递任何编码字符审作为参数来调用str . encode和bytes . decode方法也是可以的。编码将会被设置为当前平台的默认编码。这将取决于当前的操作系统和地区或K域设置，你可以使用sys • getdefaultencoding ()函数来查看它。显式地指定编码通常都是个好主意，因为一个平台的默认编码可能会改变，或者程序可能有一天被扩展成与更广泛的文本来源一起丁作。

如果你在编码文本，但不知道使用哪种编码好，可能最好的选择是使用UTF-8编码。UTF-8能够代表任何Unicode字符。现代软件中，UTF-8是一种事实上的标准编码，它能确保任何语言的文档，甚至是多语言的文档，都可以被转换。其他的各种可行编码，对于遗留文档或默认情况下仍然在使用不同字符集的地区更有用。

UTF-8编码使用一个字节来表示ASCII和其他常见字符，对于更复杂的字符则使用4个字节。UTF-8的特殊性源于它向后兼容ASCII,任何使用UTF-8编码的ASCI丨文档都和原始的ASCII文档是完全一样的

可变字节字符串

和str一样，bytes类型也是不可变的。我们可以对bytes对象使用索引和分片符号来搜索一个特定的字节序列，但是我们不能扩展或修改它们。这在处理I/O时会很不方便，因为经常需要缓冲输入和输出字节，直到它们准备好被发出去。例如，如果我们从一个轻接字接收数据，在完整收到一个消息前，可能需要调用recv函数好几次。

这就是内置类型bytearray的由来。该类型的行为或多或少像个列表，除了它只用于储存字节。bytearray类的构造函数可以接收一个bytes对象来进行初始化。extend方法可以将另一个bytes对象添加到已经存在的这个数组中（例如，当从一个套接字或其他I/O通道发来更多数据时）。

分片符号可用于在bytearray中内联地修改项目。例如，下面的这段代码通过一个bytes对象构建了一个bytearray, 然后替换了其中两个字节：

```
b = bytearray(bMabcdefghM)
b[4:6] = b"\xl5\xa3"
print(b)
```

小心，如果我们想要操纵bytearray中的单个元素，期望传人的是一个包含在0 ~ 255之间的整数。这个整数表示一个特定的bytes模式。如果我们试图传人一个字符或bytes对象，它将会抛出一个异常3

单字节字符可以通过ord（序数的缩写）函数转换成一个整数。这个函数返回的是单个字符的整数表示：

```
b = bytearray(b'abcdef')
b[3] = ord(b•g ·)
b[4  =6 8
print(b)
```

构建数组后，我们用字节103替换掉索引为3的字符（和列表一样，由于索引从0开始，所以是第4个字符）。103这个整数是ord函数返回的，它是小写字母g的ASCII字符。为了方便说明，我们还同时用大写字母D映射的ASCII字符68替换掉了下一个字符。

bytearray类型不仅包含很多方法，使它能像一个列表一样T. 作（例如，我们可以对其追加整数字节），而且它还像bytes对象一样，我们可以以相同的方式使用如count和find方法，它们和在bytes或str对象上工作是一样的。不同的是，bytearray是可变类型，它在从特定输人源建立复杂的字符序列时很有用。

文件I/O

当我们的例子接触到文件的时候，到目前为止整本书，我们操作的是整个文本文件。然而，操作系统实际上把文件看作字节序列而不是文本。

因为对于文件操作来讲，读取字节然后转换数据成文本是很常见的，Python把传人（或者传出）的字节流通过适当调用decode（或者encode）包装一下，这样我们就可以直接处理str对象了。这样节省了很多总是编解码文本的样板代码。

open（）函数用于打开一个文件。为了从文件中读取文本，我们只需要给这个函数传人文件名即可。文件会被打开并用于读取，而且通过使用平台的默认编码，字节会被转换成文本。就像对于bytes和str对象操作的decode和encode方法，打开一个文本文件，open函数可以接收encoding和errors参数，用于指定字节编码，或者在这个编码中对于那些非法字节选择一个特定的替换策略。这些通常作为关键字参数提供给open函数，例如，我们可以用下面的代码来读取一个ASCII格式的文本文件里的内容，使用替换策略转换掉任何未知字节：

```
file = open(1 filename', encoding='ascii *, errors='replace')
print(file.read(> >
file.close()
```

当然，我们不总是想读文件；我们经常想往文件里写数据！当写文本文件的时候，encoding和errors参数同样可以传递。此外，为写打开一个文件，我们需要传递一个mode参数来作为第2个位置参数，它的值是"w"：

```
contents = "an oft-repeated cliche"
file = open("filename", "w", encoding="ascii", errors=·freplace")
file.write(contents)
file.close()
```

我们也可以提供一个"a"作为mode参数，当我们不想完全覆盖现有的文件内容，而只是想添加内容的时候。

这些文件与包装它们用于把字节转换成文本的功能是非常好的，但是如果我们想要打开的文件是一个图片、可执行文件或者其他二进制文件，它将会非常不方便，不是吗？

为了打开一个二进制文件，我们只需要给mode字符串添加一个，b，。所以i_Wb •将打开一个文件用于字节写人，而'rb •允许我们去读它们。它们会像文本文件那样，但是不会有自动的编码把文本转换成字节。我们读这样一个文件的时候，它将返回bytes而不是

str, 并且当我们写这个文件的时候，如麦我们传入一个Unicode对象，写入会失败。

一旦为了读而打开一个文件，我们可以调用read readline或者readlines方法来获取文件的内容。read方法会根据模式里是否有'b'而把整个文件的内容以str或者bytes对象返回。对于大文件，小心不要不带参数地使用这个方法。如果你试图把这么多数据加载到内存中，你不会想知道会发生什么！

还可以从文件中读叹同定字节的内容；我们可以简单地给read方法传入一个整数参数，这个参数描述r你想读取多少字节$_a$下一次再调用read方法就会加载下一字节序列，以此类推我们可以在一个while循环里以一个可管理的代码块来读取整个文件。

readline方法会从一个文件里返回一行数据；通过重复调用它我们可以获取更多行数据。复数的readlines方法会返回一个包含文件所有行的列表。就像read方法一样，对于大文件这么用是不安全的。当文件是以bytes模式打开的时候，这两个方法同样可以工作，但是只有我们解析的是文本数据，这么做才有意义。例如，一个图片或者音频文件不会有换行（除非换行符代表一个像素或者声音），所以使用readline没有意义。

我们也可以使用for循环直接一次读取一个文件对象的一行，并且处理它，这样可以保证可读性以及避免一次性把一个大文件加载到内存中。

写文件也很简单；文件对象的write方法简单地把一个字符串（或者对于二进制文件的字节）对象写人文件；可以通过一个接一个地重复调用它来写多行。writelines方法接收一个迭代器并且把里而的每一个迭代项写到文件里。特别指出它并不是把参数变成多行，然后写完一行换行再写。如果希望迭代器里的每一个元素是单独一行，它们结尾就必须都有换行符。writelines基本上是一个方便的方法，它把迭代器里的内容写到文件里而无须显式地使用for循环遍历它。

文件对象最后一个盘要的方法是close方法。为了保证任何缓存的写人都被写人文件，文件被妥善清理并且所釘和文件相关的资源郎释放回操作系统，必须在我们完成对文件读/写操作以后调用close方法。从技术上来讲，当脚本退出的时候，这些将自动发生，但是最好明确一下并且我们自己清理一下。

把它放在上下文

这个必须要关闭文件的需求，会导致一些丑陋的代码。因为在文件丨/0期间可能会发生异常，我们应该把所有对文件的调用代码包装到一个try...fi$_n$all$_y$语句里，并且不管I/O是两成功，都在finally语句中关闭文件。这不是很符合Python的特点；一定有一个更优雅的方式去实现，

如果我们对一个文件类对象运行dir命令，我们会看到两个叫_611^1：_和_exit_的特殊方法。这些方法把文件对象变成了我们熟悉的所谓的上下文管理器。基本上来讲，如果我们使用一个叫with语句的特殊语法，这些方法会在嵌套代码调用之前和之后被调用。对于文件对象，_exit_方法保证了文件会被关闭，即使发生了异常。我们不再需要显式地管理文件的关闭。下面是在实践中with语句的样子：

```
with open('filename') as file:
    for line in file:
        print(line, end='')
```

调用open方法会返回一个文件对象D这个对象有_enter_和_exit_方法。通过as语句，返回的对象分配给了一个叫file的变量。我们知道当代码返回到外面的缩进级别时，文件对象会被关闭，并且即使发生了异常也照常关闭。

with语句在标准库中被用在很多需要执行启动或者清理代码的地方。例如，调用urlopen返回的对象可以用在with语句中，当我们完成时，它可以用来清理套接字。当执行with语句的时候，会自动释放线程模块里的锁。

最有趣的是，因为with语句可以用于任何具有适当特殊方法的对象，所以我们可以在我们的框架里使用。继续我们的字符串例子，让我们创建一个简单的上下文管理器，它允许我们构建一个字符序列并且在退出时自动把它转换成字符串：

```
class StringJoiner(list):
    def —enter_(self):
        return self


    def ___exit____(self, type, value, tb):
        self.result = "".join(self)
```

这段代码简单地添加了两个特殊的方法，这个继承于list的类需要一个上下文管理器。_enter_方法执行必需的设置代码（这个例子里没有），然后返回的对象将会赋给with语句里as后面的变量。通常，就像我们在这里做的一样，这只是上下文管理器自身。

_exit_*法接收3个参数。在正常的情况下，3个参数都是None。然而，如果在with语句块里发生了一个异常，它们会被设置成异常输出的类型和值相关的值。这将允许_exit_*法在即使有异常发生的情况下执行一些清理代码。在我们的例子中，我们简单地通过把字符串起来创建一个结果字符串，而不管是否发生异常。

因为这是我们写的最简单的上下文管理器之一，并且它的实用性也值得怀疑，但是它

确实能在With语句里工作。实际看一看：

```
import random, string
with StringJoiner() as joiner:
    for i in range(15):
        joiner.append(random.choice(string.ascii_letters))

print(joiner.result)
```

这段代码简单地构建了一个有15个随机字符的字符串。通过使用StringJoiner从列表里继承的添加方法把它串了起来。当出了 with语句的范围（回到外缩进级别）后，就调用T_exit_方法，然后result属性在joiner对象里就可用了。我们通过打印它的值看到一个随机字符串。

伪造文件

有时候我们需要代码提供一个类文件的接a,但是实际上并不读或者写任何真正的文件。例如，我们可能需要从一个第三方库检索一个字符串，而这个库只知道如何写文件。这是一个实际的适配器模式的例子；我们需要一个适配器，这个适配器可以把类文件的接口转换成类字符接口。

在标准库里已经有两个这样的适配器，StringlO以及BytesIO。它们很相似，除了第丨个是处理字符文本而第2个是处理bytes数据以外。这两个类都在io包里。为了读而模仿打开一个文件，我们可以给构造函数提供一个字符串或者bytes对象。调用read或者readlines将会像一个文件那样解析字符串。为了写而模仿打开一个文件，我们简单地构造一个StringlO或BytesIO者对象，然后调用write或者writelines方法。当写人完成后，我们可以通过getvalue方法看到最终写人“文件”的内容。真的非常简单：

```
# coding=utf-8
from io import StringlO, BytesIO
source_file = StringlO("an oft-repeated clich6")
de3t_file = BytesIO()

char = source_file.read(1)
while char:
    dest_file.write(char.encode("ascii", "replace"))
```

```
        char = source_file.read(1)
print(dest-file.getvalue 0)
```

这段代码，从技术上讲，无非是把一个str对象编码成bytes对象。但是是用类文件的接丨丨来执行这个任务的。我们i•先创建了一个包含字符串的源“文件”，并且是一个要写人的目的“文件”。然后我们从源一次读取一个字符，通过“replace”的错误处理策略把它编码成ASCII码，并R把结果的字节写人目的文件。这段代码并不知道调用write方法的对象不是一个文件，也不关心它是不是。

文件接口通常是为了读取和写人数据，即使它不是一个文件或者字符串。例如，网络I/O通常使用相同的协议（一组方法）来读取和把数据写人网络，并且压缩库使用它来存储乐缩过的数据:，这就是鸭子类型在工作；我们可以编写代码来操作一个类文件对象，并a它永远不会知道是否数据实际上是来自一个JE缩文件、一个字符串或者互联网。

存储对象

如今，我们能够把数据写人文件并且在以后的任何时间理所当然地可以检索它。这样做非常方便（想象一下如果我们不存储任何东西将会有多少计算状态！），我们可能经常发现丨H己去转换存在一个不错的对象里的数据或者内存里的设计模式，成为某种笨拙的文本或者二进制存储格式

Python的pickle模块允许我们把对象直接存储成一个特殊的存储格式。它本质上是把一个对象（和它拥有的所有的对象属性）转换成一种可以存储到文件或者类文件对象或者一个字节字符串的格式，然后我们就可以对它做任何我们想做的了。

对丁-基本T.作，pickle模块有一个非常简单的接口。它由4个基本的用于存储和加载数据的功能组成；两个为了操作类文件对象，另外两个为了操作bytes对象（后者只是类文件接口的快捷模式，所以我们不需要自己创建一个BytesIO类文件对象

dump方法接收一个被写入的对象以及一个要写人序列化字节的类文件对象作为参数$_3$这个对象必须有一个write方法（否则它不是类文件），并且这个方法必须知道如何处理一个bytes参数（所以一个为文本输出而打开的文件将无法工作）。

load方法完全相反；它从一个类文件的对象里读取序列化的对象。这个对象必须有合适的类文件read以及readline参数，当然，每一个必须返回bytes。pickle模块从这些字节中加载这个对象，并且load方法会返回完全重构的对象。这里有一个例子，在一个列表对象里存储并且加载一些数据：

```
import pickle

some_data = ["a list", "containing", 5,
             "values including another list",
             ["inner", "list"]]

with open("pickled—list",          'wb*) as file:
    pickle.dump(some— data, file)

      with open("pickled_list", 'rb*) as file:
       loaded_data = pickle.load(file)

print(loaded_data)
assert loaded—data == some_data
```

这段代码就像我们说的那样可以工作：对象存储在文件里，然后从同一文件加载进来。在每一种情况下，我们用With语句打开一个文件，它会自动关闭。首先打开的文件是为了M人，然后第2次打开是为f读取，这取决于我们是保存数据还是加载数据。

如果新加载的对象和原先的对象不同，最后的assert语句会触发一个错误。相同也并不意味着它们是同一个对象。事实上，如果我们打印这两个对象的id（>,我们会发现它们是不同的。然而，因为它们是内容相同的列表，所以这两个列表也被认为是相同的。

转存dumps和加载loads功能表现的和它们的类文件同行一样，除了它们返回或者接收的是bytes而不是类文件对象以外。转存dumps功能只需要一个参数；就是要存储的对象，并且它返回一个序列化的bytes对象loads功能需要一个bytes对象并且返回一个恢复的对象。方法名里的4 •是字符串的简称；这是从老版本.Python遗留下来的名字，那是str对象用来代替bytes。

dump方法接收一个n丨选参数叫protocol。如果我们保存和加载的pickle对象只会用于Python 3程序的话，我们不需要提供这个参数。不幸的是，如果我们存储的对象可能被一些老版本的Python程序加载的话，我们必须使用一个老的并且低效率的协议。这通常不应该成为问题„通常，加载pickle对象和存储它的是同一个程序。pickle是一种不安全的格式，所以我们不想通过互联网把它们不安全地发送到未知的解释器。

提供的参数是一个整数版本号。默认的版本是3,代表了目前使用Python 3最高效的存储系统。2是旧版本号，它存储的对象可以被Python 2.3之前所有解释器加载。因为2.3是外界目前仍被广泛使用的最老的Python版本.版本2的pickle通常就足够了。版本0和

1在老的解释器上受到支持；O是ASCI丨码格式的，1是二进制格式的。

然后，作为一个经验法则，如果你知道你正在序列化的对象将只会被Python 3程序加载（例如，只有你的程序会去加载它们），那么就使用默认的序列化协议。如果它们可能被未知的解释器加载，那么就使用协议号2,除非你真的相信它们可能会被一个古老的Python版本加载。

如果我们传递给dump或者dumps 一个协议，我们应该使用关键字参数来指明：pickle.dumps（my_object，protocol=2)。这并不是严格必需的，因为这个方法只接收两个参数，但是输人两个完整的关键字参数可以提醒读者我们代码里这个数字的0的是什么。在方法调用里有一个随机整数将会很难阅读。这是两个什么？可能是存储了两个对象的副本？所以要记住，代码总是应该具有可读性。在Python中，更少的代码通常比更长的代码可读性强，但并非总是如此。这点要明确。

对一个打开的文件不止一次地调用dump或者load是可能的。每次调用dump都将存储一个单独对象（加上组成或者包含的对象），而一次调用load方法将只会加载和返问一个对象。所以对于单一文件，当存储对象时，每一次单独调用dump都应该有一个对象的load调用对稍后的数据重新加载。

定制pickle

对于许多常见的Python对象，pickle “可以工作”。基本的原始数据类型像整型、浮点型，以及字符串可以被序列化，任何的容器对象，像列表或者字典，提供内容的这些容器也可以序列化。进一步，重要的是，任何对象都可以序列化，以及它们的厲性也可以。

那么什么可以让一个属性无法序列化呢？通常，对于一些时间敏感的属性，如果在将来加载它们就没有意义了。例如，如果我们有一个开放的网络套接字，打开文件、运行中的线程，或者作为属性存储在一个对象里的数据库连接，序列化这些对象是没有意义的；当我们要在稍后加载它们的时候，很多操作系统的状态就没有了。我们不能简单地假装一个线程或者套接字连接存在并且使其出现！不，我们需要某种定义的方式来存储和恢复这些数据。

这里有个类，它每小时加载一次网页的内容来确保它的更新。它使用了threading. Timer类来安排下一次更新：

```
from threading import Timer
import datetime
from urllib.request import urlopen
```

```
class UpdatedURL:
    def —init—(self, url):
        self.url = url
        self.contents =''
        self.last_updated = None
        self.update()

    defupdate(self):
        self.contents = urlopen(self.url).read()
        self.last_updated = datetime - datetime.now()
        self.schedule()

    def schedule(self):
        self.timer = Timer(3600, self.update)
        self.timer.setDaemon(True)
        self.timer.start()
```

url、contents和last_updated都可以序列化，但是如果我们尝试序列化这个类的一个实例，对于self .timer实例事情会有点疯狂：

```
»> u = UpdatedURL ("http://news .yahoo. com/")
»> import pickle
»> serialized = pickle. duxi^ps (u)
Traceback (most recent call last):
  File ,,<stdin>", line 1, in <module>
  File M/usr/lib/python3.1/pickle.py", line 1358, ir    dumps
    Pickier (f, protocol, fix iaqports=fix in^>orts) • dmp(obj)
__pickle.PicklingError: Can't pickle <class •method*>:
attribute lookup builtins.method failed
```

这不是一个非常有用的错误，但是它看起来像我们在试图序列化一些我们不该序列化的东西。就是Timer;在调度方法里，我们存储了一个self .timer的引用，这个属性不能被序列化。

当pickle试图去序列化一个对象时，它简单地尝试去存储这个对象的性；_dict_属性是对象里所有属性名字以及它们的值的字典映射。幸运的是，在检查_diet_之前，pickle检查了是否存在一ˆt*_getstate_的方法。如果它存在，它就会存储这个方法的返回值而不是diet 。

让我们给我们的UpdatedURL添加一个_getstate_方法，这个方法会返回一个没有定时器的_dict_副本：

```
def  getstate—(self):
    new_state = self.—diet—.copy()
    if * timer * in new_state:
        del new_state['timer']
    return new_state
```

如果我们现在序列化这个对象，它将不再失败。我们甚至可以通过loads成功地恢复这个对象。但是，恢复的对象没有计时器属性，所以它就不会像设计的那样去刷新页面内容。当反序列化对象的时候，我们需要以某种方式创建一个新计时器（替代丢失的那个）。

正如我们期待的那样，有一个互补的_3613七31：$_{6}$_方法可以用于实现定制序列化，这个方法接收一个单一参数，简单的就是_getstate_返回的对象。如果我们实现这两个方法，_getstate_不需要返回一个字典，因为_setstate_将知道如何处理

择返回的对象。在我们的例子中，我们简单地想要恢复_〇11(：1：_，然后创建一个新的计时器：

```
def __setstate__(self, data):
    self.—diet_ = data
    self.schedule()
```

pickle模块是非常复杂的，并且提供了其他工具来进一步定制化你需要的序列化过程。然而，这些都超出了本书的范围。我们讨论的工具对于基本的序列化任务是足够的。要序列化的对象通常是相对简单的数据对象；比如，我们不会序列化一整个正在运行的程序或者复杂的设计模式。

序列化Web对象

从一个未知或者不可信的源加载一个序列化对象不是一个好主意。它可能会往序列化的文件注人任意代码。这可以通过pickle用来攻击一个计算机。pickle的另一个缺点是它只可以被其他Python程序加载，不能很容易地和其他系统共享。

JavaScript Object Notation （ JSON ）是一个特殊的用于原始数据交换的格式 JSON是一种标准的格式，可以广泛地被异构客户端系统解释。因此，JSON对于在完全解耦的系统之间传递数据是非常有用的。此外，对于可执行代码JSON没有任何支持，只有数据可

以序列化；因此，给它植人恶意代码要闲难得多。

因为JavaScript引笮很容易解析JSON结构，所以它常用于把从Web服务器的数据传输到一个支持JavaScript的浏览器。如果提供数据的Web应用程序是用Python写的，那么它需要一种方法来把内部的数据格式转换成JSON格式。

正如我们想的那样，做这个事情的模块叫json。这个模块提供了一个和pickle模块相似的接口.有dump、load dumps以及loads闲数对于这些默认函数的调用几乎和在pickle中一样，所以我们不会重复细节。有一些差异，显然这些调用的输出是有效的jSON符号，而不是被序列化的对象。此外，json函数会对str对象操作.而不是bytes。因此，当存储或者从文件中加载的时候，我们需要创建的是文本文件而不是二进制文件。

JSON的序列化不像pickle模块那么健壮；它只能序列化基木的类型，像整型、浮点锢、字符串以及像字典和列表那样简单的容器。这些都会直接映射到JSON展示，似是JSON无法表示类、方法或者函数。不可能把完整的对象输出成这种格式。因为我们转存成JSON格式的对象的接收者通常不是一个Python对象，它不可能像Python那样观解类或者方法，无论如何，JSONI是一种数据符号；正如你所记住的，对象是由数据和n•为组成的。

如果我们想要序列化的对象只是数据的话，我们总是可以序列化对象K_di$_C$t__®性。或者我们可以通过提供定制化代码，来创建或者解析一个JSON从特定对象类锢序列化的字典，进而让这个任务半自动化

在j son模块里，对象的存储和加载函数都接收可选的参数来实现自定义的行为。dump和dumps方法接收一个els关键字参数。如果传递了这个参数，这应该是JSONEncoder类的一个子类，并且default方法被重写了。这个方法接收一个对象作为参数并且把它转换成json可以消化的字典。如果它不知道如何处理这个对象，一般就去调用super方法，以便序列化基本类型。

load和loads方法也接收一个els参数，这个参数可以是JSONDecoder类的子类。然而，通过object_hook关键字参数给这些方法传递一个函数作为参数通常就足够了。这个函数接收一个字典作为参数然后返回一个对象；如果它不知道如何处理传人的字典，它可以简单地把没有修改的字典返回。

但是这只是理论，让我们看一个例子吧！试想一下我们有下面一个简单的contact类想要序列化：

```
class Contact:
    def ___init____(self, first, last):
```

```
            self.first = first
            self.last = last


        Qproperty
        def full_name(self):
            return ('{} {}'*.format(self.first, self.last))
```

我们可以只序列化_dict_:

```
»> c = Contact ("John", "Smith1')
»> json. dun^>s (c.__diet__)
'{"last" •• "Smith" , "first" : •• John"}'
```

但是以这种方式访问特殊的（双下画线开头的）属性是比较原始的。另外，如果接收代码（可能是一些Web页面上的JavaScript)想要你提供full_name属性怎么办？当然，我们可以手工构造这个字典，但是如果我们需要很多这样的，创建一个定制的编码器是很有用的：

```
import json
class ContactEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, Contact):
            return {'is_contact': True,
                    •first*: obj.first/
                    1 last': obj.last,
                    'full' : obj . f ull__name}
    return super().default(obj)
```

default方法基本检查了我们试图序列化的对象是什么类型的；如果它是一个contact,我们手工把它转换成字典，否则我们让父类来处理序列化（假设它是一个json知道如何处理的基本类型)。请注意我们传人了一个额外的属性来识别这个对象是否是一个contact,因为加载后就无法告诉它了。这只是一个约定；对于一个更通用的序列化机制来讲，把一个字符串类型存储在字典里或者可能甚至完整的类名，包含包和模块也存到里面，这样也许会更有意义。记住，字典的格式取决于接收端的代码；必须有一个协议来说明数据如何指定。

我们可以使用这个类来编码一个contact类，就是通过把这个类(不是一个实例化对象）传递给dump或者dumps函数来实现：

```
»> c = Contact ("John", "Smith")
»> json.dumps (c, cls=ContactEncoder)
•{"is-contact": true, "last": "Smith", "full": "John Smith",
"first": "John"},
```

对于解码来讲，我们可以写一个函数，这个函数接收一个字典作为参数并且通过检查is_contact变量是否存在来决定是否把它转换成一个contact:

```
def decode—contact(die):
        if die.get('is_contact'):
            return Contact(die[* first'], die[1 last'])
        else:
            return die
```

通过使用Ob j ect_hook关键字参数，我们把这个函数传递给load或者loads函数：

```
»> data = •{ "is - con tact" : true, "last": "smith",
        "full": "john smith", "first": "john"},
»> c = json.loads(data, object—hook=decode-contact}
»> c
< main____.Contact object at 0xa02918c>
»> c. full-name
'john smith'
```

练习

我们在本章涵盖了各式各样的主题，从字符串到文件，到对象序列化，然后又回来。现在是时候考虑一下这些想法如何能用到你自己的代码中了。

Python的字符串非常灵活，并且对于基于字符串的操作来讲 Python是一个极其强大的工具。如果在你的日常工作里不进行大量的字符串处理，试着设计一个工具专门用于处理字符串。试着想出一些创新，但是如果你够坚持，可以考虑写一个Web log的分析器（每小时多少请求？有多少人访问超过5个页面？）或者一个用于把某些特定变量名替换成其他文件里的内容的模板工具。

花了很多时间来玩弄字符串格式化操作c•简单地写一堆模板字符串和对象来传入格式化函数，并且看一下你会得到什么样的输出。试一下具有异国情调的格式化操作，比如百分比或者十六进制符号。尝试填充和对齐操作，并且观察它们的行为对于整型、字符串以

及浮点型有什么不同。试着写一个方法的属于你自己的类；我们这里不会讨论细节，但是要探索一下对于自定义格式你可以做多少。

确保你理解了 bytes对象和str对象的区别。在老版本的Python（老版本的Python没有bytes，str的作用包含了 bytes和str，除非我们需要一些非ASCU码字符，在这种情况下，有一个单独的Unicode对象，这个对象和Python 3里的str类很相似。实际上要比听起来更混乱！）里这个区别非常复杂。现在它是清晰的；bytes是为了处理二进制数据，str是为了处理字符数据.，唯一需要注意的地方是，要知道如何以及什么时候对两者进行转换。为了实践，试着往一个打开的文件里写入文本数据，这个文件是为了写入bytes（你需要自己编码文本）而打开的，然后从这个文件里读取内容。

使用bytearray做一些实验；看一下它如何像一个字节对象以及列表或者容器对象那样都可以使用。试着写一个缓存区，里面以字节数组的形式持有数据，直到它达到一定长度再返回这个数组。你可以模拟这个代码，把数据放到缓存，为了确保数据不会太快到达指定长度，可以使用time. sleep。

如果在你的日常编码中不使用with语句，查看你的代码并且找出你打开文件并a忘记关闭它们的地方。特别注意那些你确实关闭了文件的地方，只是假设没有异常出现。对于每一个你找到的地方，使用with语句替换这些代码。

你有没有写过一些代码，在里面,一个自定义的上下文管理器可能会有用？考虑一下，在一些未知或者部分未知的任务里，你必须任意地设置或者清理的情况。这样的任务将可以在With语句里实现，设置或者清理的代码会由对象里的_61^6!：_和_6父；11：_方法处理。这是一个非常有用的结构，虽然它不是一个非常常见的模式；如果你想不出任何一个地方会适合一个旧的项目，作为练习. 试着在新的形势下写一个新的。

如果你曾经写过一个适配器，它从文件或者数据库加载少M数据并且把它们转换成一个对象，考虑一下使用pickle替代。对于存储海fi：数据pickle效率不高，但是对于加载配置或者其他简单对象是有用的。尝试多种编码方式；使用pickle、一个文本文件或者一个小型数据库。哪一种最容易工作？

尝试实验序列化数据，然后修改持有数据的类，并且把序列化的数据加载到一个新类里。哪一个可以工作？哪一个不能？有没有方法可以对一个类做大的改变，像重命名一个属性或者把它分成两个新属性，并且从一个老的pickle里仍然能获取数据？（提示：试着为每一个对象设置一个私有的pickle版本号并且每一次你改变这个类的时候更新它；然后你可以里放置一个聚合的路径。）

如果你在做任何的Web开发，特别是Web 2.0应用，使用JSON序列化做些实验，就

个人而言，我更喜欢序列化唯一标准的JSON可序列化对象，而不是写一些自定义的编码器或者ob ject_hooks，但是预期的效果取决于前端（通常是JavaScript）和后端代码的交互。在JavaScript和Web符号之外JSON格式不是很常见，所以如果你不做Web开发，你可能想跳过这个练习。但请注意，虽然JSON是YAML的子集，但是如果你有机会产生有效的YAML，jSON序列化仍然是有用的！

心i□

我们讨论了字符串处理、文件I/O,以及对象序列化。讨论了如何把硬编码的字符串和程序变M结合在一起，使用强大的字符串格式化系统把它们变成可输出的字符串，并学会了二进制数据和文本数据的区别。总之，我们看到：

- 如何使用不同的str方法。
- 字符串格式化。
- bytes 和 str〇
- 可变的 bytearrays。
- 二进制和文本格式的文件。
- 上下文管理器和with语句。
- 使用pickle和j son序列化数据3

在下一章，我们会讲解在Python编程中最重要的话题之一：如何测试我们的代码以确保它按照我们希望的那样运行。

11

第11章测试面向对象的程序

大部分熟练使用Python的程序员都赞成，测试是Python软件开发中的一个重要方面。尽管这一章放在了临近书的结尾，但这绝不是马后炮。我们迄今为止学习的一切都将帮助我们编写测试。我们将学习到：

. 单元测试和测试驱动开发的重要性。

- 标准的unittest模块。
- 自动化测试套件py. test。
- 代码覆盖率。

为什么要测试

越来越多的程序员认识到测试他们的代码是多么重要。如果你是他们中的一员，那么请跳过本节。下一节你会觉得更精彩，因为你会看到我们真的在学习如何在Python中测试。但如果你还不相信测试的重要性，我敢保证你的代码一定有问题，只不过你还不知道. ，所以请继续读下去！

一些人认为因为Python代码的动态特性，测试对Python来说更为重要。编译型语言，如Java和C++, 有时在某种程度上会被认为“更安全”，因为它们在编译时强制进行了类型检查。但事实是，即便有检查，Python测试也很少去检查类型，它们检查的是值。它们确保的是正确的属性会在正确的时间被设置，或是序列具有正确的长度、顺序和值。它们并不绝对保证诸如一个元组或自定义容器能够满足肯定返回一个列表的要求。这些高层次的东西在任何语言中都需要测试。相比其他语言的程序员，Python程序员更多地进行测试

的真正原因是因为在Python中进行测试非常容易！

但为什么要测试？我们真的需要测试吗？如果我们没有测试会怎么样？为了回答这些问题，可以从头编写一个没有任何测试的程序ˆ从开始到结束，直到完全写完它再运行。不要选择一个大项目，只要有几个相互作用的类就行。如果你的脑子里什么题材也没想出来，试试两个玩家的井字游戏。如果两个玩家都是人类玩家的话（不需要人T.智能），这将会非常简单。你甚至都不需要去判断谁是获胜者。

现在运行你的程序，并修复所有错误。有多少个错误？在我的井字游戏实现中出现了8处错误，而且我还不确定是不是找出了全部。你呢？

我们需要测试來确保代码是可以工作的。运行程序，就像我们刚刚做的，然后修复错误.这只是低级的测试。大多数程序员倾向于写几行代码就运行程序，以确保这些行实现了他们的预期。但几行代码的改变可能会影响到开发人员本没有预料到会受到影响的部分程序。由于他们没有考虑到这种影响，所以他们也就不会测试其他的功能。正如我们可以从刚才编写的程序中看到的.如果他们没有测试这个程序，它几乎肯定是有问题的。

为了处理这个问题，我们需要编写自动化测试。编写测试程序是非常简单的，只需要通过其他程序（或者更常见的是其他程序的某个部分，比如一个函数或者类）自动运行特定的输人。我们不但可以立即运行这些测试程序，并且还能涵盖比一个程序员每次做了修改后想到需要测试的更多的可能性输人。

编写测试有4个主要原因：

•确保代码以开发人员认为它应该工作的方式工作。

•确保当我们做出修改后代码能继续工作。

•确保开发人员理解需求。

•确保我们正在编写的代码具备可维护的接口。

第一点并不能证明值得花时间去编写测试代码，因为我们可以简单地直接测试代码（比如也许可以通过交互式解释器）。但是一旦我们必须很多次地执行相同的测试序列，将这些步骤一次性自动化，然后在必要时运行它们，就可以大大缩短所花费的时间。每当我们不得+修改代码时都运行一次测试程序是个好习惯，无论是在最初的开发阶段还是维护发行版本阶段。当我们有了一个全面的自动化测试集之后，我们可以在每次修改代码之后运行它们，从而知晓我们有没有在无意中破坏任何测试过的东西。

但是，最有趣的应该是上述列表中的最后两点。当我们为代码编写测试程序时，实际上它可以帮助我们设计代码采用的AP丨、接口或模式。因此，作为程序员，如果我们误解丫管理方或客户的需求，编写测试程序可以对找出这些误解起到很大帮助。另一方面，如

果我们不确定如何设计一个类，我们可以先编写与这个类进行交互的测试程序，从而让我们可以找到什么是测试它最自然的方式。事实上，在我们编写需要测试的代码之前就写好测试程序通常是很有益的！

测试驱动开发

先写测试代码是测试驱动开发的精髓。测试驱动开发较之“认为未经测试的代码都是无法T作的代码”的理念更进一步，它认为只有还没写的代码是可以未经测试的。除非写完了一段代码的测试代码，否则你就不该开始写任何代码。所以第一步就是编写用以证明代码可以T.作的测试代码。显然，测试将会失败，因为代码还没有开始编写。之后开始编写代码，并确保可以通过测试。然后开始为下一段代码编写测试。

测试驱动开发非常有趣。它让我们提出一个个待解决的小难题，这些就是测试。然后，我们通过实现代码来解决这些难题。再然后我们提出更复杂的难题，在解决完之前的难题后，通过编写代码解决新难题。

测试驱动方法论中有两个目标。第一个是确保真的写了测试代码。在我们写完一段代码后，很容易就会说出：“嗯，它看起来能T.作，我不用为它写测试代码了。因为这只是一个小小的改变，不会有问题的/但是，如果测试代码已经在我们编写代码之前就写好了，我们就能确切地知道它是什么时候时以工作的（因为需要通过测试），而且在未来我们也能知道它有没有因为我们或者其他人做了修改而出现问题。

第二，先编写测试代码可以迫使我们考虑清楚如何与之交互。它能告诉我们需要对象有什么方法以及用什么方式去访问属性。它帮助我们将最初的问题分解为更小、可测试的问题，然后通过与测试方案的再结合得到更大的测试方案。编写测试可以成为设计过程的一部分。通常情况下，如果我们想要为一个没有完全说明的对象编写测试，我们会发现很多异常现象，这就迫使我们去考虑设计软件的新的方面。

作为一个具体的例子，我们可以编写一段代码使用对象关系映射器将对象属性存储在数据库当中。通常我们可以在这样的对象中使用一个自动分配的数据库ID,我们的代码可以用这个ID做很多事情，就像字典中的键一样。如果我们为这样的代码编写测试，在我们开始写之前，就可能会意识到这样的设计是错误的，因为对象在保存到数据库之前是没有这个丨D的。如果我们想要在测试代码中操纵一个没有存人到数据库中的对象，它就能在我们开始写任何代码之前突显这个问题。

测试会使软件更好。在发布软件之前写好测试代码，比起让最终用户看到或购买一个满是bug的软件，会使这个产品变得更好（我曾为一家兴盛于“用户会测试软件”衍学的

公司T：作，这真的不是一个健康的商业模式！）。在写软件之前编写测试会使得它在第一次开始写的时候就更为出色。

单元测试

ih我们开始探索Python内置的测试库，这个库为单元测试提供了一个通用的接口。单元测试是一种专注于测试一个测试例中最少代码的特殊的自动化测试。每个测试都去测试令部可用代码中的一个单元。

Python库中针对单元测试的库不出所料的就叫作unittest。它为创建和运行单元测试提供了多个工具，其中最重要的就是TestCase类。这个类提供了一组方法，让我们可以比较值，安装测试，并在运行完毕后清理它们。

当我们想为某个特定的任务写一套单元测试时，我们可以创建一个叫TestCase的子类，编写不接收任何参数的方法来执行实际的测试。这些方法的名字都必须以字符串test为汗始，如果按照此约定，它们就会向动成为测试过程中的一部分。通常，测试会为一个对象设S一些值，然后运行方法，并使用内置的比较方法确保正确的计算结果。这里有一个非常简单的例子：

```
import unittest

class CheclcNumbers (unit test. TestCase):
    def test_int_float(self>:
        self.assertEquals(1, 1.0)
if __name_ == "_main____":
    unittest.main()
```

这段代码简单继承了 TestCase类，并添加了一个方法来调用TestCase.assertEquals方法该方法根据两个参数是否相等，或是成功运行，或是抛出异常。如果我们运行这段代码，unittest中的main函数会给我们以下输出：

```
Ran 1 test in 0.000s

OK
```

还记得浮点数和整数是可以相等的吗？让我们新添加一个会失败的测试：

```
    def test__str_f loat (self):
        self . assertEquals (1, " 1 *')
```

如果我们运行这段代码，输出就有点吓人了，因为整数和字符串不会认为是相等的：

```
FAIL: test str float ( main .CheckNumbers)

Traceback (most recent call last):
  File "simplest_unittest.py", line 8, in test_str_float
    self•assertEquals(1, "1”>
AssertionError: 1 != '11

Ran 2 tests in 0.001s

FAILED (failures=l)
```

第1行的点表明，第1个测试（我们之前写的）成功了，它后面的F表明第2个测试失败了。在最后，它给出了一些信息告诉我们测试为什么以及在哪里失败，同时还有一个失败次数的汇总。

在一个TestCase类中，我们想写多少个测试方法都可以。只要方法名以test开始，测试运行器就会把每个这样的函数当作一个单独的测试执行一次。每个测试应该完全独立于其他测试，先前的测试结果或运算不应该影响当前的测试。写好单元测试的关键就是保持每个测试方法都尽可能地短，每个测试用例的代码都只测试一个小单元。如果你的代码看起来拆成这样的可测试单元并不自然，那可能表明你需要重新思考你的设计。编写测试代码不仅让我们可以确保我们的代码能够工作，而且还能帮助测试设计。

断言方法

一般来说，一个测试用例的总体设计是将某些变量设置为已知值，运行一个或多个函数、方法或进程，然后通过使用TestCase断言方法“证明”返回或计算出的结果是正确的。

有几种不同的断言方法可以用来确认得到的特定结果。我们刚刚见过了assertEqual,当传递给它的两个参数无法通过相等性检查时就会导致测试失败。与其相

反，assertNotEqual会在传递给它的两个参数相等的时候失败。assertTrue和assertFalse方法分别接收一个单一的表达式，如果表达式无法通过一个if测试，那么就会失败。这些测试不仅可以检查布尔值True和False,当使用一个if语句的时候，它们还可以返回与该语句返回结果相同的值。False、None、0或空列表、字典、元组都可以通过assertFalse,同时非零的数字、含有值的容器以及True值将可以通过assertTrue〇

assertRaises方法接收一个异常类，即一个可调用对象（函数、方法或一个具有_call_方法的对象），以及需要传递的任意参数和关键字参数。断言方法将以提供的参数调用函数，当所调用的方法未将预期的异常类抛出时，断言方法失败或抛出一个错误。

此外，这些方法都接收一个名为msg的可选参数。如果提供了这个参数，那么当断言失败时，它将被包含在错误消息中。这对澄清预期值是什么或解释哪里可能有bug导致断言失败是非常有用的

Python 3.1中的附加断言方法

unittest库在Python 3.1中进行了大范围的更新。现在，这个库又包含了几个新的断言方法，并且允许assertRaises方法利用with语句的优势。下面这段代码只能在Python 3.1及之后的版本运行。它展示了两种调用assertRaises的方法：

```
import unittest

def average(seq):
    return sum(seq) / len(seq)

class TestAverage(unittest.TestCase):
    def test-python3o_zero(self):
        self.assertRaises(ZeroDivisionError,
                          average,
                          [])

    def test_python31_zero(self):
        with self.assertRaises(ZeroDivisionError):
            average([])

if name
```

```
        unittest.main()
```

上下文管理器允许我们用通常的方式来编写代码（通过调用函数或直接执行代码），而不必把函数调用包装在另一个函数调用中。

此外，Python 3.1提供了一些非常有用的新的断言方法：

- assertGreater、assertGreaterEqual、assertLess 和assertLessEqual。所有这些方法都接收两个可比较的对象作为参数，并以名字中包含的不平等关系来确定断言的成功与否。
- assertin用来确认传人它的两个参数，第1个参数中的元素是否存在于第2个参数中的容器对象（列表、元组、字典等）中。assertNotln方法是它的逆操作。
- assertlsNone测试一个值是否为None。与assertFalse不同的是，零值、False或空容器对象都不能通过，能通过的值必须只能是None。当然assertlsNotNone方法和它完全相反。
- assertSameElements接收两个容器对象作为参数，确认它们是否包含相同的一组元素，无论顺序如何。
- assertSequenceEqual考虑顺序问题。，此外，如果断言失败，它将会显示两个列表比较出的差异，从而告诉你它为何失败了。
- assertDictEqual、assertSetEqual、assertListEqual 和 assertTupleEqual，这些方法和assertSequenceEqual做了同样的事情，除了它们还要确认容器对象是正确的类型。
- assertMultilineEqual接收两个多行字符串来确认它们是否是相同的。如果它们不同，则在错误消息中显示出不同的地方。
- assertRegexpMatches接收文本和一个正则表达式来确认正则表达式是否匹.配文本。

减少样板和清理

在编写了一些小的测试代码后，我们会发现我们经常要为几个相关联的测试编写同样的安装代码。例如，下面这个简单的List子类包含3个进行简单统计计算的方法：

```
from collections import defaultdict


    class StatsList(list):
        def mean(self):
```

```
        return sum(self) / len(self)

    def median(self):
        if len(self) % 2:
            return self[int(len(self) / 2)]
        else:
            idx ~ int (len(self) / 2)
            return (self[idx] + self[idx-1]) / 2

def mode(self):
    freqs = defaultdict(int)
    for item in self:
        freqs[item] += 1
    mode_freq = max(freqs.values())
    modes =[]
    for item, value in freqs.items():
        if value == mode_freq:
            modes.append(item)
    return modes
```

显然，我们想要测试这3个方法的情景有着非常相似的输入，我们希望看到在空列表。或包含非数字值的列表，再或包含正常数据集的列表分别会发生什么。我们可以使用TestCase类中的setup方法为每个测试进行初始化。这个方法不接收参数，并允许我们在每个测试运行之前做任意设置。例如，我们可以用一个相同的整数列表对上述3个方法进行测试，如下：

```
from stats import StatsList
import unittest

class TestValidInputs(unittest.TestCase):
    def setup(self):
        self.stats = StatsList([1,2,2,3,3,4]>

    def test_mean(self):
        self.assertEqual(self.stats.mean(), 2.5)

    def test median(self):
```

```
        self.assertEqual(self.stats.median(), 2.5)
        self.stats.append(4)
        self.assertEqual(self.stats.median(), 3)

    def test_mode(self):
        self.assertEqual(self.stats.mode(),          [2,3])
        self.stats.remove(2)
        self.assertEqual(self.stats.mode(),          [3])

if __name__ == '__main__':
    unittest.main()
```

如果我们运行这个例子，它会显示所有测试都可以通过。首先需要注意的是 setup 方法一直也没有在3个test_ *方法中显式地被调用 这是因为测试套件为我们做了这一步。更重要的是，注意test_median方法中添加了一个额外的4,从而改变了列表，然而当test_mode被调用时，列表返回的是setup中指定的值（如果不是，就会有两个4在列表中.那么mode方法就会返回3个值）。这表明setup在每个测试之前都被独立地调爪了，从而确保测试类对所有的测试都有一个干净的状态。所有的测试可以以任何顺序执行，一个测试的结果不会依赖于另一个测试的结果。

除了 setup方法，TestCase还提供了无参数的tearDown方法，该方法可以用于在类中的每一个测试运行完毕后进行清理。这在需要清理继承而不仅仅是让一个对象被垃圾收集时是非常有用的例如 如果我们测试一段进行文件I/O的代码，我们的测试代码可能带来每次都创建文件的副作用 tearDown方法可以用来删除这些文件并确保系统处在与运行测试之前相同的状态。这样测试就不会有副作用了。

一般来说，我们会根据安装代码的共同之处，通过TestCase的子类将测试方法分为独立的几组。安装要求相同或相似的几个测试会被放置在一个类中 而测试中所需要的与安装无关的部分则需要置于它们向己的类中。

组织和运行测试

用不了多久一组单元测试就会增长得非常庞大并且笨拙。它很快就会变得复杂得难以一次性加载并运行所有测试3简单地运行所有测试程序并对“我最近的改变有没有破坏任何现有的测试？ ”这个问题快速地给出一个“对或错”的答案，这是单元测试的首要H标。

可以将包含测试的**TestCase**对象组或模块收集起来放到一个叫作**TestSuites**的

容器中，并且可以在指定的时间加载指定的测试。在老版本的Python中，这会导致出现许多仅用于加载和执行一个项B中所有测试的样板代码。如果需要这种更强的控制能力，该方法的功能仍然是奏效的，但大多数程序员会使用测试发现（test discovery），一种可以自动找到并运行在当前包或子包中测试的方法。

测试发现内置于Python 3.2及之后的版本（在Python 2.7中也同时开发了），但只要安装了 **discover**模块，也可以用于Python 3.1或更老版本的Python中。该模块可以从**http：//pypi .python. org/pypi/discover/**中找至IJ〇

discover模块基本上是通过寻找在当前文件夹或子文件夹中的任何以**test**字符开头的模块来工作的。如果在这些模块中发现了任何**TestCase**或者**TestSuite**对象，则执行测试。这是一个可以确保你不会错过运行任何一个测试的无痛的方法。要使用它，只需确保你的测试模块命名为**test_ 〈something〉.py**，然后根据安装的Python版本运行下面两个命令之一：

- Python 3.1 及之前版本：**python3 -m discover**
- Python 3.2 及之后版本：**python3 -m unittest discover**

忽略失败的测试

有时某个测试已知会失败，但我们并不希望测试套件在那些情况下报告失败。这可能是由于损坏了的或未完成的功能模块已经编写了测试，但目前我们并不关注去改善它。更多的时候，它可能是因为这个功能只在某些特定的平台上、特定的Python版本或特定的库的高级版本上才足可用的。Python为我们提供了一些修饰符来标记测试是桢期失败的或在已知条件下是可以被忽略的。

这些修饰符是：

- expectedFailure()
- **skip (reason)**
- **skiplf(condition, reason)**
- **skipUnless(condition, reason)**

通过Python修饰符语法就可以使用这些修饰符。第一个修饰符不接收参数，并简单地告诉测试运行器不要记录测试是失败的，即便事实上它确实失败了。**skip**方法更进一步，它甚至省去运行测试的麻烦它期待一个字符串参数来描述测试被跳过的原因。其他两个修饰符可以接收两个参数，一个布尔表达式用来表明测试是否应该运行，另一个是类似的原因描述。在实践中，这3个修饰符可以用下面的方式来使用：

```
import unittest
import sys

class SkipTests(unittest.TestCase):
      Qunittest.expectedFailure
      def test_fails(self):
            self.assertEqual(False, True)

      @unittest.skip ('*Test is useless")
      def test_skip(self):
            self.assertEqual(False, True)

      @unittest.skiplf(sys,version_info.minor
                  "broken on 3.1")
      def test_skipif(self):
            self.assertEqual(False, True)

      @unittest.skipUnless(sys.platform.startswith(1linux1),
                  "broken on linux")
      def test_skipunless(self):
            self.assertEqual(False, True)

if ____name_____ == •' —main—
      unittest.main()
```

第1个测试会失败，但它会报告这是一个预期的失败；第2个测试根本不会运行。另外两个测试会根据当前运行的Python版本和操作系统来确定是否运行。在我的Linux系统使用Python 3.1运行时，输出看起来像下面这样：

```
xssF

FAIL: test—skipunless (—main_.SkipTests)

Traceback (most recent call last):
   File "skipping—tests.py", line 21, in test—skipunless
        self.assertEqual(False, True)
AssertionError: False != True
```

```
Ran 4 tests in 0.001s

FAILED (failures=1, skipped=2, expected failures=1)
```

第1行的x表明是个预期的失败，两个s字符代表测试被跳过，F表明是一个真正的失败，因为在我的系统上skipUnless的条件是True。

用py.test测试

Python的unittest模块是非常冗长的，需要耗费大M的样板代码用来设置和初始化测试。它基于Java中非常受欢迎的JUnit测试框架，甚至使用了相同的方法名（你可能已经注意到它不符合PEP-8中使用下_线分割方法名中的单词这一命名标准，而是使用驼峰拼写法的方式）和测试布局。虽然这对于Java的测试是有效的，但它不一定是Python测试的最佳设计方案。

由于Python程序员喜欢他们的代码是简单明了的，所以一些其他的测试框架已经在标准库之外被开发出来。其中非常受欢迎的两个是py.test和nose。由于后者不支持Python 3，所以我们将在这里着重介绍py. test。

由于py.test不是标准库的一部分，所以你需要A己下载并安装它。你可以从py.test的主页http: //pytest.org/中找到它。该网站有详细的在各种不同解释器和平台下的安装说明。

py. test和unittest模块有着截然不同的布局。它不需要将测试用例写成一个类，相反，它利用函数就是对象这一优势，允许任何正确命名的函数都表现得像一个测试。相较于提供一串断言相等的自定义方法，它只是使用assert语句来验证结果。这使得测试的可读性更强，并且易于维护。

当我们运行py.test时，它将从当前文件夹开始，搜索在该文件夹和以字符test_开始的子包中的任何模块。如果在此模块中有任何阐数以test开头，则执行一个独立的测试,,

此外，如果模块中有任何类的名字以Test开始，该类中所有以test_开头的方法也会在测试环境中被执行。

所以让我们尽可能简单地将之前写的unittest例子转换为py. test:

```
def test int float():
```

```
    assert 1 == 1.0
```

对于这个完全相同的测试，相比我们之前在unittest例子中用6行写的代码，我们现在只需要写2行可读性更强的代码就能完成。

但是，我们也不排除编写基于类的测试。类在为相关联测试或需要访问类中的相关联属性或方法的测试进行分组上是非常有价值的下面这个例子显示了一个具有一个测试通过和一个测试失败的扩展类，我们会发现错误输出比unittest模块提供的更为全面：

```
class TestNumbers:
    def test_int-float(self):
        assert 1 == 1.0


    def test_int_str(self):
        assert 1 == "1"
```

注意，类不需要扩展任何特殊的对象来成为一个测试。如果我们对这个文件运行py.test,输出会像下面这样：

```
============== test session starts ==============
python: platform linux2 ——Python 3.1.2 -- pytest-1.2.1
test object 1: class_pytest.py

class_pytest.py .F
=================== FAILURES====================
_____________  TestNumbers . test_int__str _____________________________

self = <class_pytest.TestNumbers object at 0x85b4fac>

    def test int str(self):
>
E       assert 1          '1*

class_pytest.py:7: AssertionError
======1 failed, 1 passed in 0.10 seconds =======
```

输出始于一些关于平台和解释器的有用信息。这可以用于在不同系统下共享bug。第3行告诉我们被测试的文件的名称（如果有多个测试模块，它们都会显示在这里）并跟着一个熟悉的.F。这在unittest模块中已经看到过，.表示测试通过，而F表示测试失败。

所有的测试运行之后，每个测试的错误输出都会显示出来。py. test提供了一个摘要，包括局部变量（在本例中，只有一个self参数传人到函数中 >、发生错误的源代码、错误消息的总结。此外，如果一个AssertionError以外的异常被抛出，py.test会显示出包括源代码引用的完整的堆栈引用信息（traceback ）。

默认情况下，如果测试成功，py. test会抑制所有print语句的输出。这对于调试测试是非常有用的如果有一个测试失败，我们可以将打印语句添加到测试代码中，用来检查随着测试进行特定变量和厲性的值。如果测试失败，那么将输出这些值用以帮助诊断，然而，如果测试成功，打印语句的输出就不会昆示出来，允许它们简单地被忽略。最重要的是，我们不需要通过删除打印语句来“清理”输出，可以让它们一直留在测试中，如果之后测试由于我们修改了代码而失败. 凋试输出将立即可用。

一个处理安装和清理的方法

py. test支持类似于在unittest中的安装（setup ）和拆卸（teardown ）方法，但它提供了更好的灵活性[3]我们将简要讨论这些，因为对它们我们已经比较熟悉了。不过它们并不像在unittest模块中那样广泛地使用，因为py.test为我们提供了一个强大的funcarg设施，我们将在下一节中对它进行讨论。

如果我们编写基于类的测试，我们可以基本按照在unittest中使用setup和tearDown的方式来使用两个分别叫作setup_method和teardown_method的方法。它们分别在类中的全部方法运行之前和之后被调用，用来做任何设置和清理T作。不过仍然有一个与unittest中方法的不同之处，即这两种方法都接收一个参数：一个表示被调用方法的函数对象。

此外，py. test还提供了其他的安装和拆卸方法，让我们能够获得更多的控制设置和清除代码执行时的能力。setup_class和teardown_class方法预期为类方法，它们只接收一个参数（没有self参数）用来代表当前的类。

最后，我们还有setup_module和teardown_module方法，它们分别在模块中的全部测试（函数或类）运行之前和之后被立刻凋用。这些方法对于“仅一次”的设置，如创建一个将被用于所有的测试模块套接字或数据库连接时，是非常有用的。请小心使州这些方法，因为如果设置的对象需要存储状态的话，它可能意外地将依赖关系引人到测试中。

简短的描述可能不能很好地解释什么时候这些安装和拆卸方法该被调用，所以让我们来看个例子，这个例子将告诉我们都发生了些什么：

```
def setup module (module):
```

```
        print("setting up MODULE {0}".format(
              module._______name_____))

def teardown_module(module):
        print("tearing down MODULE {0}".format(
              module._______name_))

def test_a_function():
        print("RUNNING TEST FUNCTION")

class BaseTest:
        def setup_class(els):
              print("setting up CLASS {0}".format<
                    els.__name__))

        def teardovm_class(els):
              print("tearing down CLASS {0}\n".format(
                    els._____name_____))

        def setup_method(self, method):
              print("setting up METHOD {0}".format(
                    method._______name__))

        def teardown_method(self, method):
              print ("tearing down METHOD {0}'*. format (
                    method.__name__))

class TestClassl(BaseTest):
        def test__method__1 (self):
              print("RUNNING METHOD 1-1")
        def test_method_2(self):
              print("RUNNING METHOD 1-2")

class TestClass2(BaseTest):
        def test_method_l(self>:
              print("RUNNING METHOD 2-1")
        def test method 2(self):
```

```
print("RUNNING METHOD 2-2")
```

BaseTest类的唯一目的是用来提取测试类中相同的4个方法，并使用继承来减少重复的代码。因此，从py. test的角度来看，两个子类并不是各只有两个测试方法，而是两个安装和两个拆卸方法（一个处于类级别，另一个处于方法级别)。

如果我们使用py.test运行这些测试，输出会显示各个函数是在何时与测试本身以怎样的关系被调用的。我们还必须禁用抑制执行print语句输出的功能，这可以通过将-s（或--capture=no)标志传人py • test来实现：

```
py.test setup_teardown.py -s

setup_teardown.py
setting up MODULE setup_teardown
RUNNING TEST FUNCTION
.setting up CLASS TestClass1
setting up METHOD test_method_1
RUNNING METHOD 1-1
.tearing down METHOD test_method_1
setting up METHOD test_method_2
RUNNING METHOD 1-2
.tearing down METHOD test_method_2
tearing down CLASS TestClass1
setting up CLASS TestClass2
setting up METHOD test_method_1
RUNNING METHOD 2-1
.tearing down METHOD test_method_1
setting up METHOD test_method_2
RUNNING METHOD 2-2
.tearing down METHOD test_method_2
tearing down CLASS TestClass2

tearing down MODULE setup_teardown
```

为模块进行安装和拆卸的方法在会话开始和结束的地方被执行。然后，我们添加的单独的模块级测试函数被执行。接下来，第一个类中的安装方法被执行，紧随其后的是类中的两个测试。但是这些测试每个都被单独地包装在独立的setup_method和teardown method调用之中。当方法执行完毕后，类中的拆卸方法会被调用。相同的执

行顺序也对第2个类有效，在teardown_module方法最终被调用之前，将第2个类执行一次。

一种完全不同的变量设置方式

对于各种不同的安装和拆卸函数，一种最常见的使用方式就是确保在各个测试方法运行之前，特定的类或模块变ft是一个可用的已知值。

py.test提供了一种完全不同的方式来实现这个功能，这就是所谓的funcarg,即函数参数的简写。ftincarg,简单来说就是在测试的配置文件中预先设置的命名变量。它允许我们将测试的配置和执行分隔开来，允许funcarg跨越多个类和模块使用。

为了使用它们，我们只需将参数添加到我们的测试函数中。参数的名字用来查找在特定的命名函数中的指定参数。例如，如果我们想测试之前用过的StatsList类，尽管unittest已经做过了，我们又想重复测试一个包含有效整数的列表。我们可以像下面这样编写测试代码去代替使用安装方法：

```
from stats import StatsList

          def pytest—funcarg—valid_stats(request):
              return StatsList<[1,2,2,3,3,4])

def test_mean(valid_stats):
               assert valid_stats.mean() == 2.5

def test—median(valid_stats>:
       assert valid—stats.median(> == 2.5
       valid—stats•append(4 >
                assert valid_stats.median() == 3

def test—mode(valid_stats)：
       assert valid_stats.mode() == [2,3]
       valid__stats. remove (2)
                assert valid_stats.mode() == [3]
```

这3个测试方法都接收一个名为valid_stats的参数，这个参数在调用定义在文件顶部的pytest_funcarg_valid_stats函数时会被重新创建。如果有很多模块都需要这个ftmcarg，也可在一个文件名为conftest.py的文件中定义它。conftest.py文

件在加载任意一个“全局”测试配置时都会被py. test解析，它就好比对py. test的体验进行了包罗万象的定制。实际上，为了使配置完全独立于测试代码，把funcarg放在这个模块中而不是在你的测试文件里是很正常的。

对于配合py. test的其他功能，返回funcarg X厂的名字就非常重要了。funcarg只是名为 pytest_funcarg一< valid_identif ier>的函数，其中<valid_identif ier>是一个可以作为测试函数参数的有效变鲎名。这个函数接收一个神秘的请求参数，然后返回应该作为参数传入到独立测试函数的对象。每次调用一个独立的测试函数时，funcarg都会被重新创建。这就允许我们，例如，去改变某个测试中的一个列表，同时知道它将会在下一个测试中重置为它的原始值。

Funcarg能做的远比只是返回简单的变量要多得多。传给funcarg 丁.厂的request对象给我们提供了一些非常有用的方法和属性，用来修改fUncarg的行为ˆ module、els和function属性允许我们确切地知道是哪个测试正在请求funcarg。config厲性允许我们检查命令行参数和其他的配置数据。我们没有足够的空间在这个主题上深人细节，但是通过传人特定参数来运行特定的测试（对只需要很少去运行的慢测试有效），或是提供连接到数据库、文件或硬件设备的参数，自定义命令行参数可以用于自定义测试体验。

更有趣的是，请求对象（request)提供了一些方法使我们能够对flincarg做额外的清理或在不同测试中重用它。前者可以ih我们使用funcarg去代替编写自定义的拆卸函数来清理打开的文件或连接，而后者可以帮助减少运行测试套件所花费的时间，如果普遍funcarg的安装是耗费时间的这通常用在创建和销毁都是非常耗时的，且不需要在每次测试之后都重新初始化(尽管数据库仍然通常需要在测试间重置到一个已知状态）的数据库连接上。

request. addfinalizer方法接收一个回调函数，用于完成每个使用了 funcarg的测试函数在调用后的清理工作。它提供了相当于拆卸方法的作用，使我们能够清理文件、关闭连接、清空列表或重置队列。例如，下面的代码通过创建一个临时目录的funcarg来测试os . mkdir的功能：

```
import tempfile
import shutil
import os.path


def pytest—funcarg—temp_dir(request):
      dir = tempfile.mkdtemp()
      print(dir)
```

```
        def cleanup():
            shutil.rmtree(dir)
        request.addfinalizer(cleanup)
        return dir


def test_osfiles(temp_dir):
        os.mkdir(os.path.join(temp_dir,                    'a'))
        os .mlcdir (os .path.  join (temp_dir_f 'b'))
        dir_contents = os.listdir(temp_dir)
        assert len(dir_contents) == 2
        assert * a' in dir_contents
        assert 'b'  in dir—contents
```

funcarg创建了一个新的空临时目录来存放创建的文件，之后添加一个终结器(finalizer ）调用，在测试完成后删除该目录（使用shutil.rmtree，一个递归删除目录和任何其内部的函数)。这样文件系统就处于和开始时相同的状态了。

我们还有个request. cached_setup方法，该方法允许我们创建一个比一个测试存在时间更长的函数参数变量。当需要安装设置一个耗费很大的操作，且该操作可以在多个测试中重用，而重用不会破坏原子性或单元测试的性质（即一个测试不含有依赖关系，也不会受到前一个影响）时，这个方法是很有用的。例如，如果我们想要测试以下回显服务器，我们可以只在一个单独的进程中运行一个服务器实例，然后多个测试就可以连接到该实例上了。

```
import socket


s = socket. socket(socket. AF_INET, socket. SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET_f socket.SO_REUSEADDR/ 1)
s.bind((*localhost',1028))
s. listen(1)


        while True:
            client, address = s.accept()
            data = client. recv(1024)
            client. send(data)
            client. close()
```

所有这些代码只是监听一个特定端口，然后等待客户端套接字的输人。当它接收到输

人时，只是将相同的值送回。为了测试它，我们可以在一个单独的进程中开启这个服务器，并在多个测试中缓存测试结果。下面就是可能的测试代码：

```
import subprocess
import socket
import time

def pytest_funcarg___echoserver(request):
    def setup():
        p = subprocess.Popen(
                ['python3', ' echo_server.py' >
        time.sleep(1)
    return p

    def cleanup(p):
        p.terminate()

    return request.cached_setup(
            setup=setup,
            teardown=cleanup,
            scope="session")

def pytest_funcarg_clientsocket(request):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(('localhost', 1028))
    request.addfinalizer(lambda: s.close())
    return s

def test_echo(echoserver, clientsocket):
    clientsocket.send(b"abc")
    assert clientsocket.recv(3) == b*abc

def test_echo2(echoserver, clientsocket):
    clientsocket. send (b"def *')
    assert clientsocket.recv(3) == b' def*
```

我们在这里创建了两个funcarg。第1个在一个单独的进程中运行回显服务器，并返回

进程对象；第2个为每个测试实例化一个新的套接字对象，并在测试完成后使用addfinalizer关闭它第1个funcarg是我们0前感兴趣的，它看起来就像传统单元测试中的测试安装和拆卸。我们创建一个不接收参数的setup闲数并返回正确的参数在这个例子中，进程对象实际上被测试忽视了，因为它们只关心服务器是不是正在运行。之后我们创建一个cleanup函数（函数的名称可以是任意的，因为它只是一个我们传人另一个函数的对象），它接收一个特定参数，即setup返同的那个参数。这段清理代码可以简单地终止该进程

父峨数并非直接返回一个funcarg,而是返丨111调用request .cached_setup的结果；〉它接收两个参数作为setup和teardown闲数（我们刚刚创建的），以及-个scope参数这个最后的参数只能是3个字符串“function”（函数）、″module”（模块）或“session”（会ifT)中的一个，它决定参数将被缓存多长时间。我们在这个例子中将它设置为“session”，所以它将在整个py. test运行期间都被缓存直到所有的测试运行完毕，这个进程都不会被终止或重新启动。“module”范围即只为在该模块中的测试进行缓存，而“function”范围则使该对象就像普通的f'uncarg 一样，在每个测试函数运行完毕之后都会复位。

用py. test跳过测试

^unittest模块一样，出于各种原因，在py.test中跳过某些测试也经常需要。比如尚未编写测试代码，某个测试只在特定的解释器或操作系统上运行，或者某个测试非常耗费时间，应该只在某些情况下运行。

我们可以在任何时候使用py. test. skip函数来跳过代码中的测试。它接收一个特定参数：一个字符串用来描述为什么它被跳过。该函数可以在任何地方被调用。如果我们在一个测试函数内部调用.那么这个测试将被忽略。如果我们在一个模块中调用，则该模块中的所有测试都将被忽略如果我们在一个丨\mcarg函数中调用，则所有凋用了这个funcarg的测试将被忽略

3然，在所有这些地方，通常只有在特定的条件满足或没有满足的情况下，才需要跳过测试。巾于我们可以在Python代码的任何地方执行跳过函数，如在if语句中执行它，所以我们″丨以编写一段如下的测试代码：

```
import sys
import py.test


def test_simple_skip():
```

```
if sys.platform != "fakeos":
        py.test.skip("Test works only on fakeOS")

fakeos.do_something_fake()
assert fakeos.did_not_happen
```

这是一段非常愚蠢的代码，真的。没有任何一个Python平台叫作fakeos，所以这个测试在所有操作系统下都会被跳过。它显示了我们如何有条件地跳过测试，并且由于if语句可以检查任何有效条件，所以我们对什么时候需要跳过测试就有了很大的控制权。通常，我们查看sys .version_info来检查Python解释器的版本，查看sys .platform来检查操作系统，查看spme_library._version_来检查我们是否有一个给定API足够新的版本。

由于基于一定的条件去跳过一个测试方法或函数是最常见的一种使用测试跳过的方法，所以py.test还提供了一个便利的修饰符，使我们通过一行代码就可以完成它。修饰符可以接收一个字符串，这个字符串可以包含任何用于判断一个布尔值的可执行Python代码。举例如下，下面的测试只能运行在Python 3或更高版本上：

```
import py.test

@py.test.mark.skipif("sys.version_info <= (3,0)")
def test_python3():
    assert b"hello".decode() == "hello"
```

py.test.mark.xfail修饰符的行为类似，除了它用于标识一个测试将会失败，正如unittest • expectedFailure（>做的。如果测试成功了，它将被记录为失败；如果测试失败了，它被认为是个预期中的行为。对xfail而言，条件参数是可选的。如果没有提供，该测试就会在任何条件下都预期标记为失败。

py. test的补充

py.test是一个令人难以置信的强大的库，它能做得比我们在这里讨论的基础用法要多得多。我们甚至都还没开始接触它的分布式测试框架（允许所有测试可以跨网络、跨平台和解释器版本运行），它有数量众多的内置或第方插件，还有编写一个自己的插件是多么容易，以及框架提供的众多的定制化和配置架构。对于所有这些有趣的细节，你只能通过阅读在http：//pytest.org上的文档来了解了。

然而，在结束这段之前，我们还应该再讨论一些内置于py. test中的命令行参数，它们是非常有用的=和大多数命令行程序一样，我们可以通过运行命令py. test--help来得到一个所有可用的命令行参数的列表。但是，与许多程序不同的是，可用的命令行选项取决于py. test安装了哪些插件以及我们自己是否在项目的conf test, py中写了任何参数：>

首先，我们来看看几个能帮助我们调试测试代码的参数。如果我们有一个大型J则试套件，其中的许多测试都失败了（可能由于我们做的某些变更入侵了代码，比如将该项0从Python 2移植到Python 3)，py. test的输出很快就会因为数M太大而远离我们。通过将-x或一exitfirst命令行参数传给py. test, 可以强制使测试运行器在第一次失败后就退出。这样我们就可以在再次运行py. test检查下一个失败之前来解决任何导致该次测试失败的问题。

—pdb参数也是相似的，除了它并不在测试失败后退出而是停下来调用Python调试器shell。如果你知道如何去使用调试器，那么这一特性允许你快速地对存在问题的代码反思变fi或逐步运行。

py. test还支持一个有趣的--looponfail或-f参数，尽管它只在安装了 py. test xdist 插件后才of用。该插件可以在 http://pypi. python. org/pypi/pytest-xdist上找到。如果安装了它，我们可以通过将--looponfail选项传给py. test，使测试套件在遇到一个失败的测试时自动重新运行该测试。这意味着，我们可以等待一个测试失败，然后编辑该测试并修复问题代码。当我们保存文件时，测试会ft动再次运行，并告诉我们修复是否成功。这基本上和使用一exitfirst参数相似，在我们某个时间修复了一个测试后重复该测试，但是它自动设置了重启位!

M重要的py. test参数是-k, 它接收一个关键字用来搜索测试。它可以运行一个在全名中（包括包、模块、类和测试名字）包含特定关键字参数的测试。例如，如果我们有下面的这样一个结构：

```
package: something/
   module:   test_something.py
         class: TestSomething
                method: test一first
                method: test_second
```

我们可以用py.test -k test_first或只是py.test -k first来只运行其中的一个test_first方法。或者，如果其他方法也有这样的名字，我们可以运行py.test

-k TestSomething.test_first 或甚至 something.test—something.TestSomething • test_first • py • test。因为py. test会首先收集完整的测试名称，写成以圆点分隔的字符串，然后检查字符串是否包含请求的关键字。

多少测试才算够

我们已经证实未测试的代码是有问题的代码。但是我们依据什么才能告诉自己我们的代码已经测试到有多好了呢？我们又怎么知道有多少代码已经被测试了？第1个问题尤为重要，但却很难回答。即使我们知道已经测试了应用程序中的每一行代码，我们也不知道我们的测试是不是准确的,，例如，如果我们写了一个语句测试，用来检查当我们提供一个整数列表时会发生什么，那么它就仍然可能在我们提供了一个浮点数列表，或是字符串列表，再或是我们自定义的对象列表时突然失败。设计完整测试套件的责任仍在于程序员自身。

第2个问题，有多少代码真的被测试了，实际上是容易验证的。代码覆盖率本质上就足用来估计有多少行代码被程序执行了的。如果我们知道这个数字以及程序中总的代码行数，我们就可以得到一个估汁的表示有多少代码真的被测试或覆盖的百分比。如果另有一项指标表示哪些行没有被测试到，我们就可以史容易地编写新的测试来确保这些行不会有大问题了。

Python中有两个比较流行的测试概盖工具：figleaf和coverage.py。由于只有coverage .py是兼容Python 3的，所以我只写了这个，并将专注于它。这个工具可以从http：//nedbatchelder.com/code/coverage/下载到

我们无法在有限的空间中去覆盖coverage中所有API的细节，所以我们将看一些典型的例子。假如有一个Python脚本可以为我们运行所有的单元测试（例如，使用unittest .main或自定义的测试运行器或discover ），我们可以使用下面的命令来执行搜盖率分析：

```
»> coverage run coverage__unittest.py
```

这个命令将会正常退出，但同时会创建一个名为.coverage的文件，这个文件含有所有的运行数据。我们现在可以使用coverage report命令来得到一个代码覆盖率的分析报告：

```
»> coverage report
```

它的输出如下:

```
Name                          Stmts       Exec        Cover
-------------------------------------------------------------
coverage_unittest               7           7         100%
stats                          19           6          31%
-------------------------------------------------------------
TOTAL                          26          13          50%
```

这个简单的报告列出了所有执行的文件（我们的单元测试以及一个引人的模块)。列表的每行显示了每个文件中代码的行数，在测试中被执行的代码行数也同样列了出来。然后结合这两个数字可以估算出代码覆盖率。如果我们将-m选项传人report命令，它将另外添加一列，看起来像这样：

```
Missing

8-12,    15-23
```

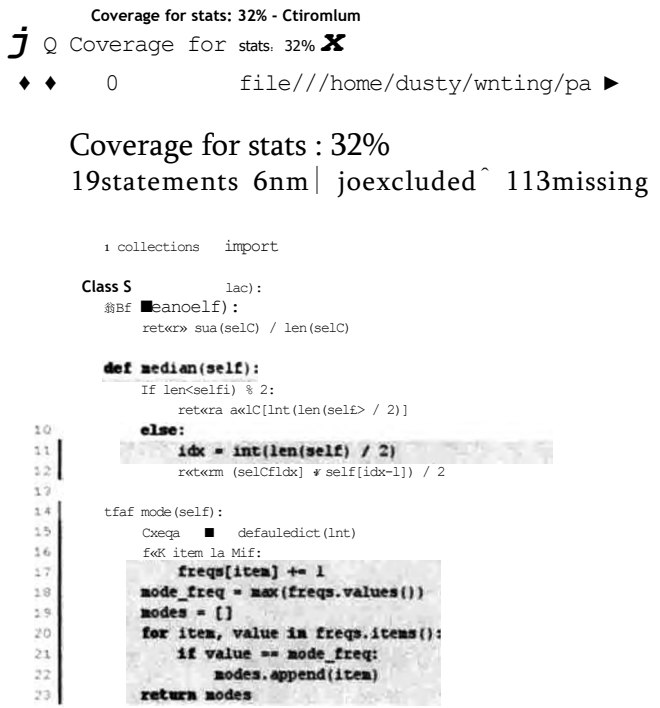
这里列出的行号范丨葡指出了在stats模块中未在测试中被执行的部分。

我们在刚刚测试过代码覆盖率的例子中使用了贯穿整章的相同的stats模块，但我们使用了一个故意忽略了测试文件中大量代码的单独的测试：

```
from stats import StatsList
import unittest

class TestMean(unittest.TestCase):
    def test_mean(self):
        self•assertEqual(StatsList([1,2,2,3,3,4]).mean(), 2.5)
if __name__ == "__main__
    unittest.main()
```

这段代码没有测试中值和取模两个函数，这两个函数的位置就在coverage输出中的告诉我们错过的行号上。

这个简单的报告在早些时候就算足够了，但如果我们使用coverage html命令，我们可以得到一个更漂亮的交互式HTML报告，这使我们可以在Web浏览器中查看它。网页中甚至对源代码中哪些行测试了哪些行没葙测试进行了高亮显示。下面就是它看起来的样子：



绿色的线表示在测试中被执行了的部分，红色的表示没有。报告中还有一些其他的交互功能，你可以自己去试试。

我们可以与py.test同时使用coverage.py模式，只需要为代码覆盖率安装py.test插件，这个插件可以从 http://pypi .python .org/pypi/pytest-coverage/中获得。该插件对py.test增加了几个命令行选项，其中最有用的是--cover-report,该选项町以被设置为html、report或annotate（后者实际上修改了源代码，用来显示全部没有被覆盖到的代码行）。

如果对本章这一小节生成一个覆盖率报告，我们会发现对于代码覆盖率我们应该了解的大部分内容都还没有覆盖到！可以在我们自己的程序（或测试套件）中使用coverage API来管理代码覆盖率，coverage.py可以接收许多我们还没有接触到的配置选项ˆ我们也还没有讨论语句覆盖和分支覆盖之间的区别（后者更为有用，同时也是最新版本的coverage ■ py的默认方式），以及其他类型的代码覆盖率。

记住，100%的代码覆盖率是一个崇高的目标，我们都应该努力追求，但100%的覆盖率仍是不够的！因为一个语句经过了测试并不意味着对于所有可能的输人都可以测试正确

案例学习

让我们通过编写一个经过测试的密码学小程序来过一遍测试驱动开发。别担心，你不需要了解现代加密算法，如Threefish、RSA等背后复杂的数学原理。取而代之的是，我们将实现一个16世纪的算法，叫作Vigen^e密码。这个程序只需要能够再给定一个编码的关键字，对信息进行编码和解码，然后就可以使用这个密码了。

首先，我们需要理解如果我们用手算（没有计算机），该密码是如何丁.作的。我们先从一个如下的表开始：

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
B C D E F G H I J K L M N O P Q R S T U V W X Y Z A
C D E F G H I J K L M N O P Q R S T U V W X Y Z A B
D E F G H I J K L M N O P Q R S T U V W X Y Z A B C
E F G H I J K L M N O P Q R S T U V W X Y Z A B C D
F G H I J K L M N O P Q R S T U V W X Y Z A B C D E
G H I J K L M N O P Q R S T U V W X Y Z A B C D E F
H I J K L M N O P Q R S T U V W X Y Z A B C D E F G
I J K L M N O P Q R S T U V W X Y Z A B C D E F G H
J K L M N O P Q R S T U V W X Y Z A B C D E F G H I
K L M N O P Q R S T U V W X Y Z A B C D E F G H I J
L M N O P Q R S T U V W X Y Z A B C D E F G H I J K
M N O P Q R S T U V W X Y Z A B C D E F G H I J K L
N O P Q R S T U V W X Y Z A B C D E F G H I J K L M
O P Q R S T U V W X Y Z A B C D E F G H I J K L M N
P Q R S T U V W X Y Z A B C D E F G H I J K L M N O
Q R S T U V W X Y Z A B C D E F G H I J K L M N O P
R S T U V W X Y Z A B C D E F G H I J K L M N O P Q
S T U V W X Y Z A B C D E F G H I J K L M N O P Q R
T U V W X Y Z A B C D E F G H I J K L M N O P Q R S
U V W X Y Z A B C D E F G H I J K L M N O P Q R S T
V W X Y Z A B C D E F G H I J K L M N O P Q R S T U
W X Y Z A B C D E F G H I J K L M N O P Q R S T U V
X Y Z A B C D E F G H I J K L M N O P Q R S T U V W
Y Z A B C D E F G H I J K L M N O P Q R S T U V W X
Z A B C D E F G H I J K L M N O P Q R S T U V W X Y
```

给定关键字TRAIN, 我们将消息ENCODED IN PYTHON进行编码如下：

1. 将关键字和消息不断重复，使字母可以很容易地从一个映射到另一个:

```
E N C O D E D X N P Y T H O N
T R A  I N T R A X N T R A I N
```

2. 对明文中的每个字母，都找到表中以该字母开头的那一行。
3. 对选中的明文字母映射到的关键字中的字母，找到该字母所对应的列。
4. 使用行列交叉位置上的字母进行编码。

例如，以E开始的行与以T开始的列交汇于字母X,因此密文中的第一个字母为X。以N开始的行与以R开始的列交汇于字母E,使密文成为XEc_ C与A交汇于C、O与I交汇于W， D和N映射为Q,而E和T映射为X。所以完整的编码消息就是XECWQXUIVCRKHWA

解码基本上就是个相反的过程。首先找到共享关键字中字符对应的行（T行），然后在那一行找到编码消息中字符 X)的位置。对应的明文字符就在这行对应列的顶部 E)。

实现它

我们的程序需要一个encode方法来接收关键字和明文并返回密文，一个decode方法来接收一个关键字和密文并返回原始消息。

但在编写这些方法之前，让我们遵循测试驱动开发的策略。我们将使用py. test来编写单元测试。我们需要一个编码方法，并且也知道它需要做什么。丨h我们为这个方法来写一段测试代码：

```
def test_encode():
    cipher = VigenereCipher("TRAIN")
    encoded = cipher.encode("ENCODEDINPYTHON")
    assert encoded == "XECWQXUIVCRKHWA"
```

这个测试自然会是失败的，因为我们没有在任何地方引人VigenereCipher类。所以让我们创建一个新模块来保管这个类。

让我们从下面这个VigenereCipher类开始：

```
class VigenereCipher:
    def ____init—(self, keyword):
        self.keyword = keyword
```

```
        def encode(self, plaintext):
    return "XECWQXUIVCRKHWA"
```

如果我们在测试类的顶部添加一行from vigenere_cipher import VigenereCipherline，然后运行py.test，上面的测试就可以通过r!我们完成r我们第1个测试驱动开发的周期。

显然，返回一个硬编码的字符串对于一个密码类来说并不是最明智的实现方法，所以让我们添加第2个测试：

```
def test_encode_character():
    cipher = VigenereCipher("TRAIN")
    encoded = cipher.encode("E")
    assert encoded == "X"
```

啊，这个测试肯定会失败。看来我们还得更加努力地工作才行。但我有个想法，如果有人试图用空格或小写字母来编码字符串会怎么样？在开始实现编码之前，我们最好在忘i己它们之前添加一些测试来应对这些情况假设预期的行为是删除空格，以及将小写字母转换为大写字母：

```
def test_encode_spaces():
    cipher = VigenereCipher("TRAIN")
    encoded - cipher • encoded ("ENCODED IN PYTHON*')
    assert encoded == "XECWQXUIVCRKHWA"


def test_encode_lowercase():
    cipher = VigenereCipher (r,TRain")
    encoded = cipher.encoded("encoded in Python")
    assert encoded == "XECWQXUIVCRKHWA"
```

如果运行这个新的测试套件，我们会发现新的测试可以通过（因为它们期望得到的是相同的硬编码字符串）。但如果我们忘了说明这些情况，它们应该就会失败。

现在我们必须考虑如何来实现编码算法。使用像上面我们手工算法中的一个表来编写代码是可行的，但考虑到每一行只是一个根据偏移量进行旋转得到的字母，这个表看起来就有些复杂了。

事实证明，我们可以利用模运算并结合字符，来取代对表的查找。给出明文和关键字字符，如果我们把这两个字母转换为它们对应的数值（A为0，Z为25），再将它们的和对

26取模得余数，这样就可以得到密文字符了！这个计算很简单，但由于它出现在以字符为基础的场景下，所以我们应该把它放到一个自己的函数中。在这么做之前，我们需要为这个新函数编写一个测试：

```
from vigenere_cipher import combine_character
def test_combine__character ():
      assert combine_character (*'E[illegible], "T"> == "X"
      assert combine_character ("N[illegible], "R") == "E"
```

好了，像往常一样，我们可以编写代码使这个函数工作起来。说真的，在我把这个函数完全正确地实现之前，不得不多次运行测试。第一次，我的函数返回了一个整数，之后我又忘了把字符从基于0的范围转换到正常的ASCII码范围。使用测试代码可以使对这些错误的测试和调试都变得容易起来，而这也是测试驱动开发的另一个好处。

```
def combine_character(plain, keyword):
      plain - plain.upper()
      keyword = keyword.upper()
      plain_num = ord(plain) - ord(* A')
      keyword_num = ord(keyword) - ord('A*)
      return chr(ord(* A') + (plain_num + keyword-num) % 26)
```

我想我们已经准备好来实现编码函数了。然而，在函数内部，我们首先想要做的是得到一个和明文一样长的关键字字符串的重复版本。让我们先来实现这个功能，更确切地说，让我们先来实现测试代码！

```
def test_extend_keyword():
      cipher = VigenereCipher (··TRAIN")
      extended = cipher ,extend__keyword (16)
      assert extended == "TRAINTRAINTRAINT"
```

在写这个测试之前，我希望先写一个独立的函数extend_key$_{W}$Ord，接收一个关键字和一个整数作为参数。但当我开始起草测试，我意识到将它变成VigenereCipher类中的一个辅助方法会更有意义。这显示了测试驱动开发是如何帮助我们设计更合理的API的。下面就是该方法的实现：

```
def extend_keyword(self, number):
      repeats = number // len(self.keyword) + 1
      return (self.keyword * repeats)[:number]
```

再一次，在运行了几次测试之后我得到了正确的功能函数。实际上，我使用了两个测试版本，一个包含15个字母，一个包含16个字母，这样做是为了确保它可以在整数除法后出现偶数的情况下可以正常工作。

现在我们终于准备好开始写我们的编码方法了：

```
def encode(self, plaintext):
    cipher =[]
    keyword = self.extend_keyword(len(plaintext))
    for p,k in zip(plaintext, keyword):
    cipher•append(combine—character(p, J0 }
    return "".join(cipher)
```

看起来没问题，那么现在我们的测试套件应该可以通过了，对吗？

实际上，如果我们运行它，我们将会发现有两个测试仍然失败了。我们完全忘记了对空格和小写字母的处理！编写测试来提醒真的是件好事。现在我们不得不在方法的开始添加这么一行：

```
plaintext = plaintext.replace(" ", "").upper()
```

现在所有测试都成功通过了。为了节约空间，我们会压缩解码的例子。这里给出两个测试：

```
def test_separate_character():
    assert separate_character("X", "┐"} == "E"
    assert separate_character("E", "R"> == "N"


def test—decode(>:
    cipher = VigenereCipher("TRAIN")
    decoded = cipher.decode("XECWQXUIVCRKHWA")
    assert decoded == "ENCODEDINPYTHON"
```

下面就是 separate_character 函数：

```
def separate_character(cypher, keyword):
    cypher = cypher.upper()
    keyword = keyword.upper()
    cypher_num = ord(cypher) - ord('A')
    keyword_num = ord(keyword) - ord('A')
```

```
        return chr(ord('A') + (cypher_num - keyword_num) % 26)
```

以及decode方法：

```
def decode(self, ciphertext):
      plain =[]
      keyword = self•extend—keyword(len(ciphertext〉 >
      for p,k in zip(ciphertext, keyword):
            plain.append(separate—character(p,)
      return ″ .join (plain)
```

这些方法与用于编码的函数有很多相似之处。写好全部的测试并都通过了，当我们再回过头去修改代码时，知道它仍然能安全通过测试也是件极具意义的事情。例如，如果我们将现有的编码和解码方法用下面这些重构方法替代，我们的测试仍然能够通过：

```
def _code(self, text, combine—func):
     text = text.replace(″ ″, ″″).upper()
     combined =[]
     keyword = self•extend—keyword{len(text}>
     for p,k in zip(text, keyword):
           combined.append(combine—func 〈p, k〉)
     return ″″.join(combined)


def encode(self, plaintext):
      return self._code(plaintext, combine_character)


def decode(self, ciphertext):
      return self._code(ciphertext, separate—character)
```

这是测试驱动开发决定性的优点，也是最重要的。一旦写完测试代码，我们就可以如我们所愿地对代码进行改进，同时对我们的修改不会破坏任何东西充满信心，因为我们已经测试过了。此外，我们还能知道什么时候我们的修复全部完成了：答案就是当测试全部通过的时候。

当然，我们的测试不可能像我们需要的那样全面地测试所有东西。代码维护或重构仍可能造成不能在测试中出现的未识别bug。自动化测试并非万无一失。但是，即便发生了bug,仍然可以遵循测试驱动的计划。第一步先编写一个测试（或多个测试）来再现或“证明”所讨论的bug的发生，当然，这将会导致测试失败。然后，编写代码使测试不再失败。

如果测试是全面的，那么bug将会被修复，并且当我们运行测试套件时，我们还将知道它会不会再次发生。

最后，我们可以试着对这段代码运行的测试成果做个判断。安装py.test-coverage插件，py.test –coverage–report=report会告诉我们测试套件是不是有100%的代码覆盖率。这是一个很好的统计信息，不过我们也不应该过于自信。我们的代码还没有在对含有数字的信息进行编码时做过测试，因此在这种输人下，其行为是未经定义的。未测试的代码是有问题的代码，未经测试的用例只能是没有定义的场景。

练习

实践测试驱动的开发。那是你的第一个练习。如果你正要开始一个新项目，那么会更容易做到这一点，但如果你是拿已有的代码进行尝试，那么你可以以你要实现的每个新功能编写测试为开始。这可能会因为你变得更加迷恋自动化测试而让你感到沮丧，因为维护旧的未经测试的代码会开始让你觉得不舒服，你会感觉你做的变更正在破坏代码，但你却没有办法知道，只因为缺乏测试。

所以在开始尝试测试驱动开发时，从一个新的项目开始。一旦你尝到甜头（你一定会的）并意识到编写测试所花费的时间很快就因为更易于维护的代码而弥补回来了，你将会想要为现有的代码编写测试。这才是当你应该开始做它的时间，不要在这之前。为我们“知道”可以工作的代码编写测试是很无聊的，很难对这个项目感兴趣，直到你意识到我们“以为”工作的代码是怎么被破坏的。

尝试编写相同的一组测试，分别使用内置的unittest模块和py.test。你更喜欢哪一个？ unittest更像是其他语言的测试框架，而py.test可证明是更Python化的。它们都允许我们编写面向对象的测试并使测试面向对象的程序更为轻松。

我们在案例学习中使用过py.test，但我们并没有涉及任何使用unittest不易测试的功能。试着将测试代码使用测试跳过或ftmcarg适配，尝试各种安装和拆卸方法，使用flmcarg做个对比：哪个你感觉更自然？

在我们的案例学习中，有很多的测试都使用了类似的VigenereCipher对象，试试使用funcarg使代码重新T作。看看能节省多少行代码？

尝试对你写过的测试运行覆盖率报告。你有没有错过对某行代码的测试？即便你达到了 100%的復盖率，你测试了所有可能的输入吗？如果你使用测试驱动开发，100%的覆盖率是相当自然的，因为你会先写测试再使代码去满足测试。然而，如果为现有的代码编写

测试，就很有可能存在没有测试的边界条件。

仔细思考下面几个不同的值：当期望填满的列表时出现的空列表；零、1或无限与中间的整数做比较；没苟精确确定的小数位的浮点数；期望是数字的字符串；或是无处不在的None值，当你预期的是一些有意义的东西时。如果你的测试覆盖了这些边界情况，你的代码将不会有错。

总结

最后，我们已经介绍了 Python编程中最重要的话题：自动化测试。在证明了它的重要性以及思考了最佳设计原则之后，我们讨论了两个流行的Python 3测试框架的基本API$_a$特别地，我们介绍了 ：

- 单 元 测 试 。
- 测试驱动开发。
- unittest 模块〇
- 断言方法和代码安装/清除。
- py.test 框架。
- 代码覆盖率。

在下一章中，我们将以一个在Python 3中工作的面向对象的框架和库的概要来结束我们的学习。

12

第12章 常用 Python 3库

我们已经讨论了面向对象编程的原则并且我们已经把它应用到了 Python中，我们已经看到了面向对象设计的来龙去脉，以及能构建优秀程序的更高层次的设计模式。我们已经看到了 Python简化面向对象解决方案的趋势。我们知道如何测试我们的Python程序。然后，我们可以做日常编程里的常见任务了吗？

是的，我们知道Python的语法，理论上，我们可以从头写一个Web框架或者数据库引擎。然而Python真正的威力在于，前人已经帮我们做了很多事情了。在整书的例子当中，我们看到了很多Python标准库模块在工作。然而我们还没有覆盖到今天Python程序员所面对的最常见的任务。我们完全绕过了阁形应用程序和它们的窗口小部件、输人框和按钮：一个用户今天最常见的接丨:丨之一。并且在我们也没有触及网络后台开发：目前Python最广泛使用的地方。

这些都是复杂的主题，并且在这里我们将一一介绍它们。在我写本章的时候，我们会把注意力放到可用于Python 3的Python库。很多流行的库，比如Django或者wxPython，目前只兼容旧版本的Python，所以这里忽略它们。

本章我们将会讲解：

- 数据库相关的库以及关系型对象的管理。
- 能点击的图形应用程序。
- 用于Web应用的CherryPy。
- 使用XML。

数据库访问

在Python中，与数据库交互是常见的任务，特别是在Web开发的世界。不幸的是，并没有太多数据库相关的库成熟地迁移到Python 3。我们将看到一些可用的数据库解决方案。

对于SQLite3，Python提供了内置的支持。在前面的章节我们看过了一些它的例子。SQLite不适合多用户、多线程访问，但是对于存储配置或者本地数据，它是个不错的选择。把所有数据存到单个文件并且允许我们使用SQL语法去访问这些数据是很简单的。我们需要做的就是把sqlite3导入然后查看帮助文档。下面是一个让你快速上手的小例子：

```
import sqlite3
connection = sqlite3.connect("mydb.db")
connection.execute(
        "CREATE TABLE IF NOT EXISTS "
        "pet (type, breed, gender, name)")
connection .execute ("INSERT INTO pet VALUES ("
        "'dog', 'spaniel', 'female', 'Esme')")
connection.execute("INSERT INTO pet VALUES("
        "'cat', 'persian', 'male', 'Oscar')")
results = connection.execute("SELECT breed, name"
        "from pet where type='dog'")
for result in results:
    print(result [1])
connection.close()
```

这段代码首先连接到一个名字叫mydb.db的本地文件（如果这个文件不存在，会创建它）并且运行了一些SQL查询语句把一个简单的表放到了数据库里。然后它查询了相同的关系并且打印了结果之一。

作为结果返回的是可迭代的元组序列。每一个元组代表一个在查询结果里匹配的行。每一个结果里的数据顺序和查询结果里的数据顺序是完全一样的。在查询中name是第2列（第1列是type），所以我们打印result [1]是打印了查询到的pet的名字。

Python的SQLite API｜使用符合数据库API｜规范的DBAP｜2。这个AIM是一个标准，旨在使代码使用同一种方式号不同数据库交互变得更容易。有类似的API｜支持其他数据库，例如PostgreSQL、MySQL以及Oracle，目前在众多种类中，它们中很少能成熟用于Python 3。

任何遵循DBAPI2规范的数据库API｜都会有一个能返回Connection对象的connect阐数(对于不同的数据库连接这个函数可能会有不同的参数)。通过使用execute

方法在这个连接里执行查询。通常情况下，还会提供一些额外的方法来简化查询或者返回一个命名的元组作为结果；然而，这都不是DBAP12规范所要求的。

然而，DBAP丨2是非常底层并且难以使用的。在面向对象编程中，通常使用一个Object-Relationa丨Manager或者叫ORM来与数据库交互。ORM允许我们使用熟悉的称为对象的抽象，这种抽象我们一直在本书中使用，比如当把它们的属性连接到关系数据库范例里。在Python中最受欢迎的ORM是SQLAlchemy, 这也是第一个被移植到Python 3的ORM。

引入 SQLAlchemy

SQLAlchemy 可以从 http://www. sqlalchemy • org/下载。只有 0.6 或者更高的版本支持Python3, 并且在写这本书的时候，唯一支持的数据库是SQLite和PostgreSQL。

作为SQLAIchemy提供了一个基于数据库AP丨的抽象，这不是一个巨大的交易，（理论上）它可以写一个能在一种数据库系统上工作的SQLAIchemy代码，然后同样的代码（或者稍微修改）用在其他的数据库上。所以如果你要寻找MySQL的支持，你可以开始写你自己的代码，这个代码首先使用SQLAIchemy并把SQLite作为后台，当最终后台支持了，再把它移植到MySQL。

SQLAIchemy是一个非常大和健壮的库；它几乎允许我们做可以想到的任何和数据库相关的事。在本节，我们只会涉及一些基础。

SQLAIchemy以及0RM背后的想法通常是，在后台更新数据库的表并且与正在自动修改的对象交互。SQLAIchemy提供了多种把对象映射到表的方法；我们将会使用现代基于继承的解决方案。如果你需要连接到一个遗产数据库，SQLAIchemy提供了一个替代的方法来允许任意对象类可以显式地映射到数据库表，但是这里我们没有时间去讲这些内容。

我们需要做的第一件事情就是连接到数据库。sqlalchemy. create_engine函数提供了一个用于连接数据库的单一访问点。它需要大M的参数去定制化或者调整访问。最重要的是一个URL字符串，它定义了要连接的后台数据库种类，后台要连接的具体数据库，数据库的名字，运行数据库的主机，还有一个用户名密码用于身份验证。URL的基本形式类似于 Web URL: driver: //user:password0host/dbname〇

如果我们要使用一个简单的，不需要用户名、密码或者主机的SQLite数据库，我们可以简单地指定数据库的文件名，我们会在下一个例子中看到。

然后我们需要创建一个类，来允许对象把它们的数据存到数据库里，同时有选择地为这个对象提供一些方法。每一个对象的实例都会存到数据库的一个单独的行里，由其主键

标识（把这个主键做成一个单独的整数标识符通常是一个好主意，但是SQLAlchemy不需要这么做）。

数据库里的每一个衣通常都由一个单独的类代表，并且每一个类里的特殊属性都会映射到这个表的列。当我们要访问一个对象的属性时，我们会得到数据库值，并且当我们更新并保存这个对象时，数据库会被修改。这里有一个我们宠物数据库的简单例子：

```
import sqlalchemy as sqa
from sqlalchemy.ext.declarative import declarative_base

Base == declarative base ()

class Pet(Base):

    __tablename__ = "pet"
    id = sqa.Column(sqa.Integer, primary_key=True)
    type = sqa.Column(sqa.String(16))
    breed = sqa.Column(sqa.String(32))
    gender = sqa.Column(sqa.Enum("male", "female"))
    name = sqa.Column(sqa.String(64))

engine = sqa.create_engine('sqlite:///mydata.db')
Base.metadata.create_all(engine)
```

SQLAlchemy首先要求我们通过调用一个叫declarative_base的函数来建立一个Base类。这个函数会返回一个类，我们可以用这个类来扩展我们的声明。子类需要一个叫__tablename__的特别属性来指明数据库中表的名字。

紧跟着的是一些列的声明。我们添加了一个Column对象，它的第一个参数是一个类咽对象（例如Integer或者String ），后续的参数会取决于这个类型。所有这些类型对象都是sqlalchemy包提供的。我通常在引人这个包的时候会带一个叫sqa的别名，这样会让引用这个包里的其他类变得容易。有些人建议使用from sqlalchemy import *这个语法，这样所有的对象都是可用，但是就像我们在第2章讨论的，这样会让代码维护比较混乱。

在定义了一个或者多个扩展Base对象的映射类以后，我们通过使用create_engine函数连接到一个特定数据库（在目前的情况，是一个SQLite文件）。Base.metadata .create all的调用确保了 Base类的所有表都是存在的。这通常会在

数据库底层执行某种CREATE TABLE的调用。

添加和查询对象

就像一个正常对象一样，我们可以创建我们自己的表对象的实例。Base类默认的构造闲数不接收任何参数。给我们的子类添加一个_init_方法通常会很有用，它可以初始化这个对象的部分或者全部变量。我们也可以给这个类添加任意我们喜欢的方法。下面我们可以看到是如何实例化一个新的pet对象并且设置一些参数的：

```
pet = Pet ()
pet.id = 1
pet.type = "dog"
pet.breed = "spaniel"
pet.gender = "female"
pet.name = "Esme"
```

这个对象可以像其他Python对象一样使用，但是这个对象并没有通过任何方式连接到数据库。在我们可以把这个对象和数据库的表联系起来之前，我们需要创建一个SQLAlchemy的Session（会话）对象。会话就像是数据库和对象之间的预备区域。我们可以给这个会话添加多个对象. 以及使用会话来记录更改、删除以及其他数据库操作。当我们已经准备好把这个集合的变更保存到数据库的时候，我们可以通过commit（>来提交它们，或者，如果出现问题，我们可以调用session.rollbackO让所有变更消失。

下面看一下如果把我们新创建的pet对象添加到数据库并保存它：

```
Session = sqa.orm.sessionmaker(bind=engine)
session = Session()


session.add(pet)
session.commit()
```

首先，通过调用sessionmaker函数我们得到了一个特殊的Session类；这个函数需要知道要连的数据库引擎是哪个。然后每马我们想要一个会话的时候，我们就实例化这个Session类。在提交数据库的修改之前，每一个会话都部分独立于其他会话。接下来，它们基本依靠数据库处理，采用熟悉的规则，并且根据底层数据库的不同，规则可能也不同。

我们也可以使用会话对象来查询数据库。SQLAkhemy查询是结合Python函数以及原

始的SQL语法写成的。我们使用session.query ()方法来获取一个Query对象。这个方法接收代表要查询的表或者列的参数。然后这个对象的方法可以获得一连串的结果集。这些方法包括：

- all ()，返回表里的所有条目。
- first ()，返回第一个条目。
- one ()，返回唯一的条目。如果没有条目或者发现多个条目，它会抛出一个异常。
- get (primary_key)，接收一个主键值然后返回匹配这个键值的对象。
- group_by ()、order_by ()，以及having ()，把相关的SQL语句添加到查询当中。
- filter_by()，使用关键字参数来查询这个会话。
- filter ()，使用更高级的SQL表达式（我们一会会讨论）查询。

那个filter_by方法允许我们通过使用关键字参数来搜索条目。例如，我们可以说：

```
session.query(Pet).filter_by(name="Esme").one()
```

这个filter_by参数试图使用一个特殊字符串来匹配名字。这会返回一个新的查询对象，在这个对象里，我们可以使用one ()方法得到一个单一值（因为在我们的示例数据库里只有一个数值，并且和我们的标准匹配，它将返回结果）。如果我们调用all ()方法，它将会返回一个包含条目的列表，在现在的情况下，它只会有一个条目。

SQL表达式语言

和接收关键字参数的filter_by不同，filter方法接收SQLAlchemy的SQL表达式语言中的值。这是一种更加强大的查询方式，它适用于对对象列的不同操作。它是一个有趣的重载特殊操作方法的应用。

举个例子，如果我们使用 session.query (Pet) .filter (Pet. name=="Esme")，包含在filter查询里的表达式不是一个计算布尔值的典型相等比较。相反，它构造了一个合适的SQL语句，过滤方法会使用这个语句进行数据库查询。这是通过重写Pet. name列对象W_eq_方法实现的。所以我们需要显式地声明用于相等比较的Pet. name对象。我们不能直接简单指定name，因为如果它是一个关键字参数，这样会导致一个错误发生。

SQL表达式语言使用许多相关的操作来构造查询。一些比较常见的有：

- !=不相等。
- <小于。

- >大于。
- <=小于等于
- >=大于等于。
- &使用AND查询组合语句。
- | 使 用 OR查询组合语句。
- ~使用NOT来把一个查询变成相反的查询。

SQLAlchemy表达式语言几乎允许任何SQL语句通过Python构造出来，包括创建连接、汇总语句以及使用SQL函数。然而，我们有很多话题需要讲，所以你需要通过去看其他的资料来发现如何使用它们，整本书已经写了在Python中的SQL、SQLAlchemy和数据库，所以这个简单的介绍只不过可以引发你的兴趣。

漂亮的用户界面

这本书的所有例子都是通过命令行运行的。对于系统管理员，喜欢鼓捣Linux的人，以及这个时代祖父级的程序员来讲这非常好. 但它不允许我们去写那种今大每个人都在用的现代桌面程序。事实上，有人可能会争论认为即使是桌面应用也足古老的东西，Web应用（我们很快会讨论）以及手机应用程序更加时尚。

我们还没有看到图形应用程序的原因是. 它们总是依赖设计模式所提供的如此高水平的底层抽象，以至于很难看到对象的模式。这对于学习面向对象编程并不是十分有用。但是现在我们已经知道了面向对象的来龙去脉，我们可以简要地看一下图形用户接口，或者简单称之为GUI的世界。

从头开始设计图形界面接口是可以的，需要跟屏幕的像素打交道来产生视觉效果。但是没有人这么做。相反，我们会使用某种窗H部件工具包，它为我们提供了一些常用的图形元素，比如按钮、文本框、复选框、工具栏、选项卡、日历，以及当我们看台式电脑时，不管是什么操作系统都能看到的那些更多的东西。

我们将会简要地（不幸的是，非常简要）讨论其中两个可以在Python 3中运行的窗口T具包。但是首先，让我们先讨论一下少ft的理论。图形化程序总是使用事件驱动的体系结构。这通常意味着它们严重依赖我们在第9章讨论的命令模式。当和用户交互的时候，我们永远不可能精确地知道什么时候他们将要按下一个键、移动鼠标，或者点缶一个对象，也不会知道他们想要在什么时间执行哪些活动。所以只苻当它们发生了，我们写的代码才会响应。这段代码应该快速无痛地响应，这样程序可以返回并且等待用户的下一个输人。

简而言之，这就是事件驱动编程的世界。起初这可能会很闲扰你，但是它非常适合我们一直在讨论的面向对象原则。

Tklnter

Python标准库里附带了一个名叫tkinter的内置图形库。它伴随Python预装到了大多数的操作系统上，尽管它需要安装TCL/TK解释器以及图形T. 具包。

一个Tklnter应用程序最基本的配置是创建一个Frame对象，给这个窗口添加一些桌面小部件对象，然后让tkinter的mainloop接管它们。mainloop负责等待串[1]件发生并且把我们写的相应代码分派给它们执行。这是一个非常基本的Tklnter应用程序，在所创建的窗口里不显示任何东西：

```
import tkinter


class EmptyFrame(tkinter.Frame):
      pass


root = tkinter.Tk()
EmptyFrame(master=root).mainloop()
```

首先我们创建一个类，扩展了 tkinter. Frame类；对于其他窗口部件，这是一个基本的容器。我们创建了一个Tk（>对象提供一个窗口来装载这个框架，然后调用mainloop运行这个对象。如果我们运行这个程序，它会显示一个小的空的窗口。不要太激动。

让我们看看这个真正可以交互的例子：

```
import tkinter
import random


class DiceFrame(tkinter.Frame):
      def _____init_____(self, master):
            super().—init_(master)

            die = tkinter.Button(self,
                        text = "Roll!",
                        command=self.roll)
            die.pack()
            self.roll_result = tkinter.StringVar()
```

```
            label = tkinter.Label(self,
                                  textvariable=self.roll_result)
            label.pack()
            self.pack()

        def roll(self):
            self.roll_result.set(random.randint(1, 6))


root = tkinter.Tk()
DiceFrame(master=root).mainloop()
```

这里发生了一些事情，几乎所有的东西都在一个重写的_init_方法里。初始化超类之后，我们创建了一个新的Button对象，代表你经常看到的“确认”和“取消”按钮，你可能从没有想过它们是对象！所有的Tkinter窗口部件都会把父窗口部件作为它们的第一个参数=这里我们通过传入self来让这个新的按钮属于这个框架。我们还提供了一个text参数，代表了显示在按钮上的字符串，还有command参数，该参数是一个当按钮被点击时要调用的函数a在我们这个情况，这个函数是同一个类里的方法。还记得命令模式吗？这里就是！

然后对我们新的按钮调用了 pack方法，这是一个基本的格式，简单地设置窗口和按钮的默认大小及位置。如果我们不调用这个方法，这个按钮会不可见。

之后，我们用相似的代码创建和封装了一个Label对象。标签携带一个有Tkinter提供的特殊的结构化的StringVar对象。这个类的好处是，任何时候只要调用set ()方法就可以更新一个新的字符串，和这个对象相关的任何窗口部件都会自动更新它们显示的内容为新的字符串。每一次我们单击“Roll”按钮的时候，我们通过这个特性来随机更新标签显示的内容。当我们运行这个程序的时候，会给我们展示一个非常简单的电子模型:

所以图形化编程都是关于构建窗口部件，并且把命令关联到它们身上，一旦特定的事件发生，就会调用相应的命令。最复杂的部分，通常是让显示“看起来正确”；也就是说，让所有的窗口部件以一个美观的方式布局，同时达到易用和容易理解$_3$我们可以通过pack方法进行定制。这个方法基本上允许窗口部件布局在任意行或者列。如果我们需要使用列的行或者行的列，我们可以把多个对象报到独立的框架里（例如，框架里包含很多行），然

后把这些独立的框架打包到它们的父框架里（例如，这个打包用到多列）。当打包一个小部件时，我们可以传给它下面一些额外的参数，来控制在它的父窗口里如何摆放。

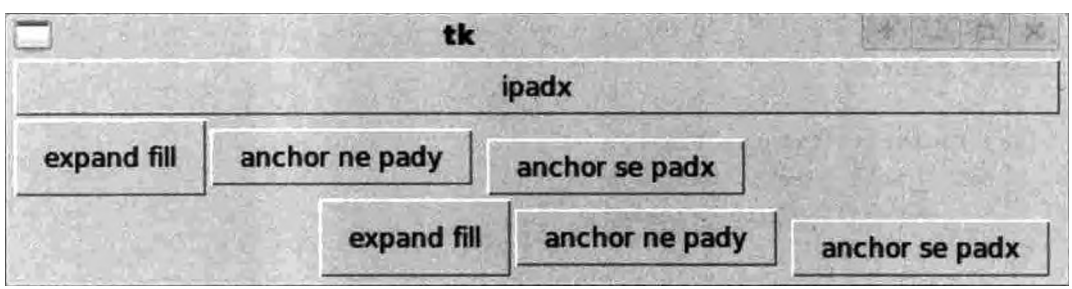
- expand：一个布尔值，决定如果父窗口缩放大小，是否窗口部件也要跟着缩放，即使超出了其预期大小。如果多个窗口部件设置了 expand，那么会在它们之间划分额外的空间。
- fill：可以设置none、x、y或者both的字符串值，用来指导窗口部件填满所有在特定方向分配给它的可用空间。
- anchor：如果窗口部件没有设置填满它的空间，它可以在这个空间里固定位置。默认的是center，即确保到各个方向距离相等，其他像n、e、s、w这些值可以用来指示方向，把窗口部件在可用空间里居上、居右、居底，或者居左放置，并且ne、se、sw以及nw可以用来设置东北、东南、西南和西北4个角。
- ipadx和ipady：这些整数值提供了在窗口部件里面填充上下左右边缘大小的功能。它有增大窗口部件大小的效果。
- padx和pady：这些提供了填充窗口部件的边缘和它可用空间的值。它有在窗口部件和它的相邻部件之间放置空间。
- side：使用left、right、top bottom 4个方向之一，沿着这个特定的方向放置窗口部件。通常情况下，在一个容器里的所有窗口部件都会被放到同一侧，混合它们的话会带来无法预料的效果。如果你需要不止一行或者一列，你可以把一个框架放置到另一个框架里。

下面是这些特性的一个示例：

```
import tkinter

class PackFrame(tkinter.Frame):
      def _____init_____(self, master):
            super()._init_(master)

                        buttonl = tkinter.Button(self,
                              text = "expand fill")
            buttonl .pack (expand=True, f ill=··both", side="left")
            button2 = tkinter•Button(self,
                        text = "anchor ne pady")
            button2 .pack <anchor=' ' ne", pady=5, side="left")
            button3 = tkinter.Button(self,
```

```
                text = "anchor se padx",f)
            button3.pack(anchor="se", padx=5, side="left")

class TwoPackFrames(tkinter.Frame):
    def —init (self, master):
        super().—init—(master)
        button1 = tkinter.Button(self,
                text='ipadx")
        button1.pack(ipadx=215)
        packFrame1 = PackFrame(self)
        packFrame1 .pack (side="bottom", anchor="e")
        packFrame2 = PackFrame(self)
        packFrame2 .pack (side="bottom",f, anchor="lw")
        self.pack()

root = tkinter.Tk()
TwoPackFrames(master=root).mainloop()
```

这个例子创建了网个PackFrame的实例，每一个实例包含了 3个水平放置的按钮（通过指定side="left"）。每一个按钮放置的位置都不同。这些框架又被垂直地放到了另一个有非常大的按钮的框架里，固定在框架的左右两边。下面是这些代码渲染出的样子。它离好看还差得很远，但是它说明了在一个单独窗口里的大部分概念：



当设汁复杂的接U,放置窗口框架时会比较单调。如果你试阉这么做，你可能需要研究一下Tkinter的网格窗口部件的布局风格。

我们这里没存时间来讨论网格布局或者一些广泛可用的Tldnter窗口部件，但是和SQLAIchemy 一样，希望你有一个体验，能知道你的选择是什么，并且如果你想构建一个GUI的应用程序，你应该首先准备仔细想想。图形界面接口并没有比命令行更复杂。它们

只是依赖不同的设计模式来完成它们的工作。

PyQt

另一个Python 3支持的主要图形工具包叫PyQt。它是一个很流行的跨平台Qt库的Python绑定，这个库在开源和商业许可上都是4用的。PyQt可以从http://www/riverbankcomputing.co.uk/software/pyqt/download 下载○ PyQt 是一个比较高级的库，它支持很多通常不被认为是GUI工具包一部分的特性。在某些方面，PyQt是一个桌面应用程序框架，扩展支持了从Web浏览器到多媒体数据库的所有东西。

然而，我们只能看看PyQt作为一个图形库的基本用法。之前，让我们通过显示一个空窗口开始：

```
from PyQt4 import QtGui

app = QtGui.QApplication([])

class EmptyWidget(QtGui.QWidget):
      pass

window = EmptyWidget()
window.show()
app.exec_()
```

和tkinter版本并没有太大的不同。在我们创建任何窗口之前，我们需要构建一个QApplication对象，因为构造它可以在内部初始化QU然后我们构建一个空的窗口，然后调用app里的mainloop函数，这通过调用Qt里的exec_完成。

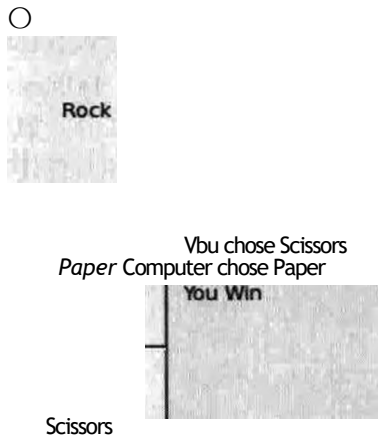
我们将构建一个基本的岩石、纸、剪刀的窗口，而不是一个滚动骰子的应用程序：

```
from PyQt4 import QtGui
import random

app = QtGui.QApplication([])

choices = ["Rock", "Paper", "Scissors"]

class RockPaperScissorsWidget(QtGui.QWidget):
```

```
        def —init—(self):
            super () . —init—()
            rock = RPSButton("Rock", self)
            paper = RPSButton("Paper", self)
            scissors = RPSButton("Scissors", self)
            for button in (rock, paper, scissors):
                button•resize(100, 100)
            rock.move(0,0)
            paper.move(0,100)
            scissors.move(0,200)
            self.response = QtGui.QLabel("", self)
            self.response.setGeometry(110, 0, 200, 300)

    class RPSButton(QtGui.QPushButton):
        def mousePressEvent(self, event):
        computer_choice = random.choice(choices)
        user_choice = self.text()

        comp_idx = choices.index(computer_choice)
        user idx = choices.index(user_choice)

        message = {
                  0:  •Tied',
                  1:  'Computer Wins',
                  2:  •You Win,}[(comp—idx      user idx H h 3) % 3]

        self.parent().response.setText("You chose {0}<br />"
                    "Computer chose {1}<br />••
                    •• {2 }"• format (user_choice, computer_choice, message))

    window = RockPaperScissorsWidget()
    window.show()
    app.exec_()
```

这个程序给一个窗口里添加了 4个窗口部件（3个按钮和1个结果标签)ˆ每一个按钮都是我们设计的QPushButton子类的实例。这个继承的唯一目的就是重写了mousePressEvent方法，这个方法会在鼠标单击按钮的时候调用。创建按钮之后，我们

使用了绝对定位功能（resize、move和setGeometry ）把窗口部件放置到我们希望的位置。下面是它们看起来的样子：



在我们重写的mousePressEvent方法内部，我们随机给电脑玩家一个选择，然后计算赢家。这个计算是通过一个简单的取模的数学函数达到的，这个函数能评估是0、1或者2,依据是谁获胜了（为了让这个值是对的，我不得不做一些实验;我的数学知识比较匮乏！）。然后我们通过这个结果去一个字典里查找相关的信息。最后，通过调用setText方法，我们在标签上显示结果。每一次按下按钮，计算机会做一个新的选择，然后更新标签内容。

PyQt提供了很多窗口部件，从按钮、复选框到能输人字段的单选按钮，从组合框、滑块部件到像日历、视频播放器和语法高亮显示的文本编辑器等高级窗口部件。一般来讲，任何你见过的窗口部件，PyQt都能提供。如果你需要的东西它没有，可以去创建你自己的窗口部件。PyQt也像Tklnter一样提供了一些高级的布局调整方案，这样在一个可变大小的窗口中定位窗口部件就不是一件痛苦的事了。（我们例子里使用的绝对定位方式并不是可以调整大小的。）

选择一个GUI工具包

PyQt和Tklnter都是目前Python 3可用的GUT工具包。此外，还有两个Python 2工具的包非常流行：PyGTK和wxPython。前者正处于往Python 3移植的晚期，可能你在读这篇文章的时候，它已经可用了。后者是一个非常先进的跨平台系统，有基于当前运行的操

作系统平台显示本地窗口部件的优势，这允许wxPython程序“无缝”地融入操作系统的感觉和外观。

但是对于一个给定的任务你应该选择哪个？这真的取决于个人爱好。你需要每一个都尝试一段时间，来决定哪一个提供给你最舒服的编程范式。它还取决于你的特定需要。如果你要给一个小的脚本或昔程序开发一个简单的接口，Tklnter可能是你最好的选择，因为它是和Python绑定的，因此不需要做额外的安装部署工作。如果你计划开发一个带有复杂窗II部件的以GUI为中心的应用程序，你最好不要使用其他任何的库。选一个你感兴趣的窗口部件开始工作。PyQt可能是最容易学和使用的，但是如果你有在其他语言中使用wxWidgets或者GTK的经验. 你可能会发现Python或者pyGTK■是更合适的选择

XML

很多Python程序员认为处理XML是一件非常痛苦的事情。但是，XML是受到极其广泛的欢迎的。Python程序经常不得不和它去交互；作为消费者把从其他数据源来的数据解释成XML, 以及作为生产者需要给其他程序或者计算机创建XML数据让它们去解析。

在Python的标准库里有3个文档较好的库可以用于和XML文档交互。有两个是基于传统的XML解析技术的，第3个是一个不错的Python接口。

SAX（ Simple AP丨for XML ）库是一个事件驱动的系统，当在字节流里遇到特定的对象时就调用特定的函数去处理：打开和关闭标签、属性或者内容。使用它可能不是那么灵活，但是它的优势在于“动态地”解析XML文档而无须把整个流加载到内存。这对于大型文档很有帮助。

DOM (Document Object Mode) 库采用了不同的方式。它允许在任何时间访问文约的任何部分，并把文档当成一个把点连起来的树。每一个点代表一个文档中的元素、属性或者文本。元素可以有子元素，并且可以随机访问它们。它允许读写XML文档并且动态地给这个树修改添加新的节点。

这些工具都有它们的用途，但是Python中最常见的XML任务是解析，使用第3个叫ElementTree的内置库或者一个在此基础上更高级的叫lxml的库可以减轻你的痛苦。
这两个库都允许把XML文档当作Python对象处理，使它们容易阅读、组合、交互和修改。

ElementTree

xml. etree . ElementTree包里包含一些操作XML文档的类和函数。它们之中最重要的是Element和ElementTree两个类。一个ElementTree类本质上代表了内存中一个完整的XML文档；它使用了我们在第10章讨论的组合模式来构造一个树形的Element对象。它有一个单一的指针指向根节点，这个根节点包含了一些子节点，子节点又包含了更多的子节点，以此类推。

任意一个Element对象包含了 XML文档中开始标签和关闭标签之间的所有东西。它引用了标签的名字、在开始标签中的任意属性、元素内部的文本，以及递归的子元素的列表（嵌套的开始标签和关闭标签）。如果文本中包含描述的文字以及标签元素（就像在一个HTML文档中）,任何在一个关闭标签和后续标签之间的文本(不管它是一个新打开的标签,还是一个之前关闭的标签），添加到元素里的一个tail属性里。

作为一个例子，让我们从一个简单的HTML文档开始：

```
<!DOCTYPE html>
<html>
    <head>
        <title>This is a web page</title>
        clink rel="stylesheet" href="styles . css,f />
    </head>
    <body>
        <hl>Welcome To My Site</hl>
         article class^'simple'')
            This is <em>my</em> site. It is rather lame.
            But it's still <strong>mine</strong>. Mine,
            I say.
        </article>
   </body>
</html>
```

> 如果你感兴趣，这个文档是一个有效的HTML 5源码。如果你对于XHTML或者HTML4熟悉的话，看到这个新版本可读性更好的时候，你就放心了。在一些标准的开发团队里一定有些Python程序员，一直提醒大家，可读性是很重要的！

现在，下面的程序会把这个HTML文档加载到内存中，并说明各种元素是如何关联的：

```
from xml.etree.ElementTree import fromstring

with open (nhtml_document .html*') as file:
    root = fromstring(file.read())

print ("ROOT NODE*')
print ("\ttag, root. tag)
print("\tnumber of children:", len(root))
print("\tchildren:[c.tag for c in root.getchildren()])
head = root[0]
print("HEAD NODE")
print("\tfirst child:", head[0].tag)
print(M\tsecond_child:", head[1].tag)
print("\tlink attributes:", head[1].attrib)
article = root[1][1]
print (,fARTICLE NODE'*)
print ("\ttag,article. tag)
print (,f \ttext:, article. text)
print("\tfirst child:", article[0]•tag, article[0].text)
print("\t\tem_s tail:", article[0]•tail)
print("\tsecond child:", article[1].tag, article[1].text)
print("\tstrong1s tail:", article[1].tail)
```

如果我们运行这个程序，我们可以看到子列表是如何通过使用一个元素类来访问tag、text attrib 和 tail 属性的：

```
ROOT NODE
    tag: html
    number of children: 2
    children: [* head1,'body']
HEAD NODE
    first child: title
    second_child: link
    link attributes: {'href1: 'styles.css',  'rel1: * stylesheet1}
ARTICLE NODE
    tag: article
    text:
            This is
```

```
    first child: em my
          em's tail: site. It is rather lame.
           But it's still
second child: strong mine
strong's tail: . Mine,
         I say.
```

值得注意的一件重要的事情是，一个元素的子元素总是另一个具有类似接口的元素列表查找支持对节点的迭代（就像for child in element ）和分片（就像element [1: 5]),所以很容易把一个元素当成一个普通Python序列看待。

一件需要小心的事情是，当检测子元素的时候，ElementTree有点模糊。不要使用if element:语句来决定一个元素是否存在，因为如果这个元素存在但不包含子元素，它可能会返回False。如果你想检查一个元素是否含有子元素，使用if len（element）i吾句。相反，如果你想检查一个兀素是两存在，使用if element is not None语句。

如果我们想解析或者处理一个从文件读来或者互联网提供的元素树，这个简短的介绍几乎已经是足够了。通常我们读取或者接收一个XML文件时，我们需要做一两件事情：

- 一个节点一个节点地解析它并且把它转换成其他结构。
- 找到特定的元素或者属性并且查找它们的值。

第1个任务可以通过递归遍历节点以及查看它们的属性、文本以及尾部来实现。第2个任务通常意味着某种搜索机制。Element类提供了一些方法来帮助匹配元索。有3个这样的方法。它们返回不同的值，但都接收一个叫pattern的参数。这个参数支持一个非标准版本的XPath选择语言。不幸的是，不支持完整的XPath语言。但是一些基本的特性，像选择一个标签、递归地选择一个标签，以及从当前T作的节点构建一条路径，尽管：

```
print('search for child tag:', root.find('head'))
print('search children recursively:*, root.findall(*.//em'))
print('build path:', root.findtext('./body/article/em'))
```

如果运行这段代码会得到输出：

```
search for child tag:  Element head at 961f7ac>
search children recursively: [ Element em at 961fb2c>]
build path: my
```

这个例子还说明了 3种不同的搜索方法，每一种都接收同样的路径。find方法会返回第1个叫配的元索 f indall方法会返回一个匹配元素的列表Df indtext方法略有不同：

它找到第一个匹配的子元素（就像find方法），但是之后它会返回这个元素的文本属性，而不是元素本身〇因此，e. findtext（path）和e. find.（path）. text是相同的。

构造XML文档

ElementTree不仅可以很好地用于解析和搜索XML文档。它还提供了一个直观接口使用标准的列表和对象访问特性来构造XML文档。我们对一个元素可以使用append函数给它添加子元素，或者帮助函数，可以少打点字，就是SubElement。我们可以通过使用字典语法来设置属性，通过访问对象厲性来设置文本以及尾巴。下面的代码通过使用这些特性来构造一个简单的HTML文档：

```
from xml.etree.ElementTree import (Element, SubElement,
        tostring)


root = Element("html")
head = Element("head")
root.append(head)
title = SubElement(head, "title"}
title.text = "my page"
link = SubElement(head, "link")
link.attrib['rel1] = "stylesheet"
link.attrib['href'] = "styles•css"
body = Element("body")
body.text = "This is my website."
root.append(body)

print(tostring(root))
```

首先我们创建了一个根元素，然后按照顺序添加一些子元素。SubElement函数做了同样的事情给XML头添加一个标题。当我们创建一个链接元素，我们也是通过更新属性字典来给它设置属性。最后，我们通过使用tostring方法来把一个元素转换成一个XML字符串，看起来像这样：

```
<htmlxheadxtitle>my page</title><link href="styles .css'*
rel=*'stylesheet" /></head><body>This is my website.</bodyx/html>
```

ElementTree有到目前为止我们提到的更多的东西，但是不像那些可替代的XML库，

基本的东西会让你花很长时间。

Lxml

lxml是一个高级的XML解析库，使用非常快的libxml2库做底层困难的丁.作。它可以从Ixm丨的站点下载http://codespeak.net/lxml。它是一个第三方库，过去，在一些操作系统上很难安装，虽然使用最新版本应该不会这样。

如果你的需求是一些基本的并且可以通过我们刚刚讨论到的ElementTree API丨转化的话，那么就通过一切方法，使用ElementTree。但是如果你需要解析无效的XML文档，高级的XPath搜索，或者CSS选择器，lxml就是你的工具。

lxml有一个非常类似ElementTree的接口，实际上，对于基本的用法，lxml可以用作ElementTree的一个方便的替代。这将一定会使你的解析代码加速，但兼容ElementTree并不是lxml的伟大之处。lxml是更为先进的并且提供了大量超过ElementTree的特性。

前面解析和搜索XML文件的例子只需要改一个地方就可以用于lxml;把导入改为读取from lxml.etree import fromstring并且代码不用修改就可以运行〇

我最喜欢的lxml高级特性，是它支持高级的用于搜索整个XML文档的XPath以及CSS选择器。这远比基本的ElementTree搜索有用。这里有一些例子：

```
print('xpath attribute:', root.xpath('//link[0href]*))
print('xpath text filter:*, root•xpath('//*[contains(em, "my")]'))
print(* xpath first child:', root.xpath(f/html/body/article/em[1]'))
from lxml.cssselect import CSSSelector
print(1 css class selector:*, CSSSelector('.simple1) (root))
print(1 css tag selector:*, CSSSelector(•em*) (root})
```

lxml将支持任意XPath选择器，这是通过底层的libxml2库实现的。这包括了整个XPath语言定义，虽然一些最奇特的选择器可能不成熟。

对于任何使用jQuery JavaScript库或者类似库的人来讲，CSS选择器是非常合适的。CSS选择器在内部编译，相当于XPath选择器在选择之前就运行了。XPath和CSS选择器的功能都是返回一个所有匹配元素的列表，这和ElementTree的findall方法类似。

除了这些高级的搜索功能，lxml还提供了：

•一个可以解析错误HTML格式的解析器。

. 一个可以把元素看成对象的独特的库，这样如果子标签是对象的属性，你就可以访问它们。

- 一个完整的XML验证T. 具，它可以使用DTDs XMLSchema以及RELAX NG模式。

这甲. 我们没有时间去讨论这些，但是当你处理XML或者HTML解析时，如果你有任何高级或者复杂的需求的话，Ixml 一定是你想要的T. 具。

CherryPy

CherryPy 3. 2版是第一个在Python 3平台可用的主要的Web应用程序服务器。可以从http://cherrypy.org下载到。它不像很流行的Django、TurboGears或者Zope库那样是个全栈的框架。这些框架额外提供了对数据存储、模板、身份认证以及其他Web操作的支持。这些特性在CherryPy中是没有的，你需要自己去寻找或者实现它们。

CherryPy是一个通过使用简单的设计来构建Web应用程序的强大的Web服务器。我们直接通过一个简单的例子来处理我们在前面开发的HTML文件：

```
import cherrypy


class SimplePage:
    @cherrypy.expose
    def index(self):
        with open("html_document•html") as file:
            return file.read()


cherrypy.quickstart(SimplePage())
```

如果我们运行这个程序，我们可以通过在Web浏览器里访问http://localhost:8080来实际看一下这个Web页面。我们这里所做的是创建了一个类并把它传给了quickstart函数。这个闲数会启动一个Web服务器并处理从类里传进来的页面。在这个我们创建的类里所有方法都被标记成了 exposed, 这样通过HTTP使用带有这个方法名字的URL就可以访问这个方法ˆ任何没有明确标为曝漏的方法可以在内部作为帮助方法使用，但是不能通过任何URL访问到。

这个方法本身只是简单打开了一个文件，并返回该文件的内容。最终，我们编写一个能处理单个HTML 5页面的Web应用程序。

当然，一个只有一个页面的网站是非常无聊的$_3$让我们看一个能触碰到更多精彩内容

的例子：

```
import cherrypy

template = """<! DOCTYPE html<>
<html>
    <body>
        {message}
    </body>


class AboutPage:
    @cherrypy.expose
    def index(self):
        return template•format(message="' "'
        I'm not a very interesting person.

@cherrypy.expose
def contactPage(self>:
    print(self)
    return template.format(
            message="I can't be contacted anywhere.")

class MainPage:
    about = AboutPage()
    contact = contactPage
    @cherrypy.expose
    def index(self):
        return template . format (message=····u
        This is the main page.
        <a href=·,'/about/·f>About Me</a>
        <a href ="/contact/*'>Contact Me</a>
        <a href=··/links/··>Some Links</a>
        """)

    @cherrypy.expose
    def links(self):
```

```
            return template.format(
                        message=*'No Links Yet")


cherrypy.quickstart(MainPage())
```

这个例子展示了 3种给网站添加页面的方式。最明显的一个是添加一个曝漏的方法，就像上面的类里面的link方法。但是我们也可以通过其他方式添加曝漏的对象。

- 就像在contactPage里那样，通过定义一个独立的函数并且在类定义里包含它的属性来实现。
- 就像我们在aboutPage里那样，通过定义一个独立的类并且在类定义里包含一个实例来实现。
- 就像app• some_page = AnExposedClass ()那样，在初始化类之后，通过给这个对象添加一个曝漏的方法来实现。

你可能已经发现,index方法是一个特殊的方法。它并没有映射到/index这个URL;相反，它会在斜线后面没有添加路径时调用。

我们也可以接收HTML表单作为参数。让我们创建一个真的联系页面：

```
import cherrypy


class ContactPage:
      @cherrypy.expose
      def index(self, message=None):
            if message:
                  print("The user submitted:\n{0}".format(
                        message))
                  return "Thank you!"
            return ""n<form>
                  <textarea name="message"x/textarea>
                  <input type="submit" />
                  </form>"""


cherrypy.quickstart(ContactPage())
```

这个页面根据当前关键字参数message会显示不同的结果。如果没有提供这个参数，会呈现给访问者一个可以输人消息的表单。如果提供了参数，消息的内容会打印到控制台（正常情况下，对于这个值我们会做一些有用的事情，比如通过电子邮件把它发送到某个

地方或者存储到数据库或者文件以供后继检索)。然后会返回给客户端一个谢谢你的消息。

因此，message参数该如何设置？通常来说，当提交一个页面时，一个表单里任何命名的输入（在这种情况，指的是消息textarea)都会映射到一个关键字参数。就是这么简单！

一个完整的Web堆栈

正如我们所讨论的，CherryPy只是一个Web应用服务器；它不是一个Web框架。它提供了一个完整的Web服务器以及把HTTP请求映射到代码，这些代码是做出这些请求时需要执行的。它还提供了一些需要一点配置的特性，比如完整的**SSL**支持，设置或者取回cookies的能力，缓存支持，HTTP身份验证和会话。然而，它缺失了两个关键的特性，而这两个特性很多其他框架都提供：模板和数据存储。

很多网站使用数据库做数据存储，但是CherryPy并没有提供这个能力。我们真的需要它吗？我们真正需要的是数据库连接；它并不需要内置到Web框架里。的确，为什么我们不用在本章早些时候讨论的SQLAlchemy呢？事实上，这就是TruboGears框架用来访问它的数据库的。

这之后，我们仍然先不去解决模板的问题，这是CherryPy缺失的另一个框架特性。模板是基于某种上下文把静态的字符串或者文件里特定的子字符串替换成新的字符串的过程。在第10章中讨论的str. format函数是一个模板的基本例子。它允许我们用传递到函数里的变量替换掉修饰符。事实上，这是我们在之前那个简单的CherryPy应用程序中用到的模板方法。

大部分模板语言远超像条件（包括在模板里只有满足特定的条件的数据，例如两个变：«：相等，或者用户登录了）和循环（包括在模板里重复的数据，例如创建一个表格或者一个打乱顺序的列表，它的元素来自于一个Python列表）这样的能力。一些更深入的，甚至允许任意的Python代码在模板内部执行。

关于一个模板语言应该是什么样的有无数的看法，这就是为什么对于Python 2来讲，会设计出大量的不同的模板语言。这种多样性还没有扩展到Python 3,但是其中一个最强大的模板语言Jinja2 已 经 在 Python 3的平台上可用了。它可以从http://jinja.pocoo • org/下载。

作为一个案例学习，让我们使用CherryPy SQLAlchemy以及Jinja这3个工具创建一个快速和简单的博客引擎！我们将以SQLAlchemy模型开始；这些定义了那些将要存储到数据库的数据：

```
import datetime
import sqlalchemy as sqa
from sqlalchemy.ext.declarative import declarative_base

Base        =        declarative_base()

class Article(Base):
       —tablename— = "article"
       rowid = sqa.Column(sqa.Integer, primary—key=True>
       title = sqa.Column(sqa.String)
       message = sqa.Column(sqa.String)
       pub_date - sqa.Column(sqa.DateTime)

       def _init_(self, title, message):
       self.title = title
       self.message = message
       self.pub_date=datetime.datetime.now()

class Comment(Base):
       —tablename  = "comment'*
       rowid = sqa.Column(sqa.Integer, primary_key=True)
       article_id = sqa.Column(sqa.Integer,
                    sqa.ForeignKey('article.rowid *))
       article = sqa.orm.relationship(Article, backref="comments")
       name = sqa.Column(sqa.String)
       message = sqa.Column(sqa.String)

def _____init_____(self, article—id, name, message):
       self.article—id = article_id
       self.name = name
       self.message = message

engine = sqa.create_engine('sqlite:///blog.db')
Base.metadata.create_all(engine)
Session = sqa.orm.sessionmaker(bind=engine)
```

我们创建了两个带有一定字段的模型。这两个模型通过一个Comment类的

ForeignKey关系联系起来。

rowid是一个特殊的字段；在SQLite中，每一个模型都会自动地被赋予一个唯一的整数rowid值。我们不需要做任何事来填充这个值，它在数据库中是简单可用的。这对于PostgreSQL或者其他数据库引擎是不行的，而是我们需要给它们创建一个序列或者autoincrement 宇段〇

我们为每一个类添加了一方法，使之更容易构造新的实例3然后我们关联引擎. 创建表以及创建一个Session类来和数据库交互。

Jinja模板

现在，我们可以创建Jinja来为我们处理来自一个文件夹的一些模板：

```
import jinja2
templates = j inja2.Environment(loader=j inja2.FileSystemLoader(
            ' blog—templates•)>
```

这很容易。这里给了我们一个templates变量，基于一个给定的文件夹里的文件名，我们可以通过它加载模板。在我们创建Cherry Py应用服务器之前，让我们看一下这些模板。首先让我们仔细看看这个用于添加博客文章的简单模板：

```
{% extends "base.html" %}

{% block title %}New Entry{% endblock %}
{% block content %}
<form method="POST" action="/process_add/">
      Title: <input name="title" type="text" size="40" /><br />
      <textarea name="message" rows="10" cols="40M>
      </textareaxbr />
      <input type="submit" value="Publish" />
</form>
{% endblock %}
```

这种类似于普通的HTML, 但是所有这些{ %是新的。这是Jinja用于模板标签的标记（如果你用过Django的模板系统或者对它感兴趣的话，会发现它类似于Django的标记）。模板标签会指示模板系统在这个地方做些特别的事情。有两种类型的模板标签在使用：extends和block: extends标签本质上是告诉模板系统从base.html开始，但是会

换掉任何在这个文件中有名字的块。这就是block和endblock标签的作用；在父模板base. html中不管指定什么，有名字的块都会覆盖它。如果我们知道base. html长啥样可能会清楚点：

```
<!DOCTYPE html>
<html>
      <headxtitle> {% block title % } {% endblock % }</titlex/head>
      <body>
            <hl>My Blog</hl>
            <ul>
                  <li><a href=‥/ⁿ>Main</a></li>
                  <li><a hrefss″/add/‥>Add Entry</a></li>
             </ul>
             <hr />
             {% block content %}
             {% endblock %}
        <body>
  <html>
```

这看起来更像一个正常的HTML页面；它展示了在一个更大的页面中，这两个有名字的块该显示什么。

在其他的模板里扩展base. html可以让我们忽略每一个页面里相同的部分。进一步说，如果我们想给菜单添加一个链接或者修改整个网站，我们只需要在一个单独的模板文件里做就行了。

其他的模板，像index. html是更加复杂的：

```
{% extends "base.html" % >

{% block title %}My Blog{% endblock %}
{% block content %}
    {% for article in articles %}
        <h2>{{article.title}}</h2>
        <em>{{article.pub_date.strftime('%b %d %Y')}}</em>
        <p>{{article.message}}</p>
        <div style=ᴹmargin-left: 6em*'>
              <h3>Comments</h3>
              {% for comment in article.comments %}
```

```
                <em>{{conunent. name} } wrote :</em>
                <p>
                {{comment.message}}
                </p>
            {% endfor %}
            {% include " comment_form. html '' %}
        </div>
        <hr />
    {% endfor %}
{% endblock %}
```

它包含了和之前模板相同的extends和block标签。此外，它还引人了 for模板标签，它会遍历所有的文章（或者文章中所有的评论）并且为它们呈现出不同的HTML。它也值染一批使用{{<variable_name>}}语法的变量。变量名会从我们的CherryPy应用程序传进来或者在上下文里分配，然后在for循环内完成循环。

在文章里渲染pub_data变量是非常有趣的，因为这个变量是一个datatime.datetime对象，并且我们可以看到，Jinja允许我们直接对这个对象调用strftime方法。

巖后，include标签允许我们在一个单独的文件里渲染部分模板，就像comment_form.html 看起来这样：

```
<form method="POST"
        action='f/process_comment/{{article.rowid} }/**>
    Name: <input name="name" type="text" size=" 30" /xbr />
    ctextarea name=Mmessage" rows="5" cols="30">
    </textarea><br />
    <input type="submit" value='f*Comment" />
</form>
```

简而言之这就是Jinja的基本语法；当然可以做的事情远比这个多，但是这些基本知识足以让你感兴趣。它们对于我们简单的博客引擎也是足够的！

CherryPy博客Web应用程序

为了了解Web应用程序是如何设计的，请注意，我并没有在写我们将要看的CherryPy应用程序之前写那些模板。相反，我迭代开发，通过同时创建模板和写代码来添加一篇文

章，紧跟其•后的代码和模板是为了显示这篇文章，最后，建立评论系统。在上一节中我把所有生成的模板放到了一起，因此我们可以重点去看Jitija的模板语法。现在，让我们关注一下CherryPy以及这些模板是如何被调用的！

首先，下面是我们带有索引方法的博客引擎：

```
import cherrypy
class Blog:
    Qcherrypy.expose
    def index(self):
        session = Session()
        articles = session.query(Article).all()
        template = templates . get_template (" index. html '*)
        content = template.render(articles=articles)
        session.close()
        return content
        cherrypy.quickstart(Blog())
```

在这里，我们开始看到3个拼图合并到了一起。当然，ChenyPy是用于页面。Jinja使用我们的模板来创建页面，并且SQLAIchemy给了 Jinja需要显示的数据。

首先，我们构造了一个session并且用它来搜索所有可用的文章。然后我们通过名字得到一个模板；来自于之前在这个模块里我们创建的templates对象。然后，我们通过传递一个关键字参数给这个模板来渲染它。关键字参数映射到模板内容里的变量；我们之前定义的这个模板会循环遍历传进这个函数的文章。然后我们返回要呈现的内容让CherryPy显示它。

让新添加的文章显示在表格的代码更加简单；我们只是渲染模板，因为它不需要任何变fit：

```
    @cherrypy.expose
    def add(self):
        template = templates.get_template("add.html")
        return template.render()
```

你可能已经注意到在我们的模板中，用来添加文章以及评论的表格有行为属性指向了process_add 以及 process_comment URL〇 这个 process_add URL 简单地从表格参数（标题和名字）构造了一篇新文章，这篇文章作为关键字参数从CherryPy呈现给我们。然后它抛出一个异常来把客户重定向到主视角，也就是将显示这篇新文章：

```
@cherrypy.expose
def process_add(self, title=None, message=None):
    session = Session()
    article = Article(title, message)
    session.add(article)
    session.commit()
    session.close()
    raise cherrypy.HTTPRedirect("/")
```

process_comment方法与process__add方法，除了前者还接收一个位置参数以外，很类似。位置参数来自于在URL里的分隔符之间，所以如果传人一个article_id等于3，下面的方法实际上会映射到/process_comment/3/：

```
@cherrypy.expose
def process_conunent(self, article_id, name=None,
        message=None):
    session = Session()
    comment = Comment(article_id, name, message)
    session.add(comment)
    session.commit()
    session.close()
    raise cher rypy. HTTPRedirect (,f/M)
```

现在我们有了一个没有身份认证以及会在几分钟之内充满垃圾邮件的完整的、简单的博客引擎，但是它可以工作！并且这一切我们都是用Python 3对象写的。

练习

在本章中我们讲解了各种各样的内容，但是我们没有讲解太多的细节。然后这些练习将会是额外的阅读。本章我们讨论的工具在它们的网站上都有非常好的文档，包括教程、API参考以及具体的例子。如果你对我们讨论过的任何一个话题特别感兴趣，那就去回顾这些库的文档。试试看，你能走多远。

获取写一个复杂的GUI程序你需要的知识，然后把它写出来。Tklnter和PyQt都试一下，然后决定你喜欢哪一个工具。找出在Python 3里SQLAlchemy所支持的数据库后台，在我写作的时候，仅有两个，但正在快速增加。同样，研究可用的Web框架，并且看一下它们和Cher^Py比较起来怎么样。Jinja是最好的可用模板工具吗？尝试一下其他产品，并

且看看你能想到什么。或者如果你有时间并且想要挑战，可以根据我们从第10章获取的一些字符串操作知识来写一个你自己的模板引擎！

试一下ElementTree和Ixml,看看你能否发现它们的异同。如果你想把lxml合并到我们上面创建的Web堆栈里并且用它创建HTML文档而不是用模板渲染它们，该怎么办？（提示：这不是一个好主意，但不管怎样值得一试！）

这里还有数百个Python库和AP丨可以用，并且每一天越来越多的它们会可以用于Python 3。如果你需要解决一个特定的问题，很可能这里已经有一个可用的支持库能帮助你解决。整本书尤其是这一章，我们在例子中接触了一些流行的库。但是除此之外还有非常多。例如，我们并没有讨论科学计算或者显示包。对一些可用的库可以做一些研究，找出什么是可用的$_3$你可能永远不会知道何时会派上用场！

总结

本章相关的各种主题都是相当特别的。我们从数据库开始，渐进到图形用户接口，分化到对XML的讨论，并且最终构建了一个小型的Web应用程序。目标是为主要的真实世界里的工作引人流行的可用的库。Python 3中可用的库的数量正在稳步增长，因为越来越多的开发者选择支持当前Python语言版本里使用的清晰语法。我们看到了一个概述：

- 用于数据库的SQLAlchemy。
- 用于围形接口的Tklnter。
- 用于不同图形接口的PyQU
- 用于 XML 解析的 ElementTree。
- 用于更好的XML解析的lxml。
- 用于Web应用程序的CherryPy。

. 在Web应用程序里用于字符串模板的Jinja

就这样结束在Python中我们通往面向对象编程世界的旅程。我真诚地希望你能够喜欢驾驭并且兴奋地测试你的新技能来创新新的编程问题。谢谢你的关注，保重。