



计 算 机 科 学 从 书

WILEY

数据结构与算法

Python语言实现

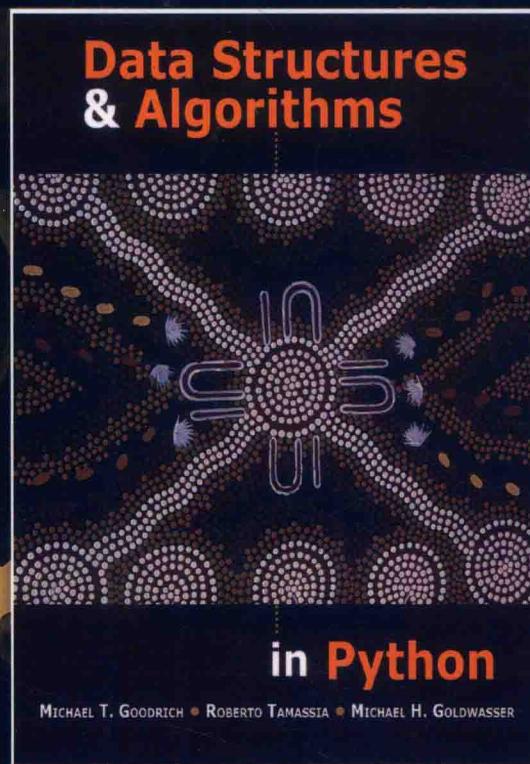
迈克尔·T. 古德里奇 (Michael T. Goodrich)

[美] 罗伯托·塔马西亚 (Roberto Tamassia) 著

迈克尔·H. 戈德瓦瑟 (Michael H. Goldwasser)

张晓 赵晓南 等译

Data Structures and Algorithms in Python



机械工业出版社
China Machine Press

数据结构与算法 Python语言实现

Data Structures and Algorithms in Python

继C、C++、Java之后，采用Python语言讲解数据结构与算法成为国内外高校的新选择。相比于同类教材，本书并非简单地替换了代码部分，而是根据Python的特点重新规划篇章结构，从而更加符合教学需求。本书要求读者具备一定的高级语言基础，快速入门Python后便进入核心思想——面向对象编程，接着重点讲解栈、队列、表、树和图等内容，同时涵盖文本处理、DNA序列比对和搜索引擎索引等大量实例。

本书特色

- 强调面向对象思想，关注抽象数据类型及其算法实现策略，通过理论方法和实验方法分析算法性能，学会比较和权衡不同策略。
- 基于Python3标准，对于书中讨论的数据结构均给出了完整的Python实现代码而非伪代码，并提供全部源代码的免费下载。
- 包含约500幅精心设计的插图，易于读者理解抽象概念；以及超过750道练习题，便于读者巩固知识、发散思维或开展项目实践。

作者简介

迈克尔·T. 古德里奇 (Michael T. Goodrich) 加州大学欧文分校计算机科学系教授，之前是约翰·霍普金斯大学教授。他是AAAS、ACM和IEEE会士，曾荣获IEEE计算机协会技术成就奖和ACM卓越服务奖等。

罗伯托·塔马西亚 (Roberto Tamassia) 布朗大学计算机科学系教授及系主任，兼任几何计算中心主任。他是AAAS、ACM和IEEE会士，曾荣获IEEE计算机协会技术成就奖。

迈克尔·H. 戈德瓦瑟 (Michael H. Goldwasser) 圣路易斯大学数学系和计算机科学系教授，兼任计算机科学项目主任，之前曾在芝加哥罗耀拉大学任教。

WILEY
www.wiley.com

Copies of this book sold without
a Wiley sticker on the cover are
unauthorized and illegal



华章教育服务微信号



上架指导：计算机/数据结构

ISBN 978-7-111-60660-4



9 787111 606604 >

定价：109.00元

投稿热线：(010) 88379604

客服热线：(010) 88378991 88361066

购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com

网上购书：www.china-pub.com

数字阅读：www.hzmedia.com.cn

计

算

书



数据结构与算法

Python语言实现

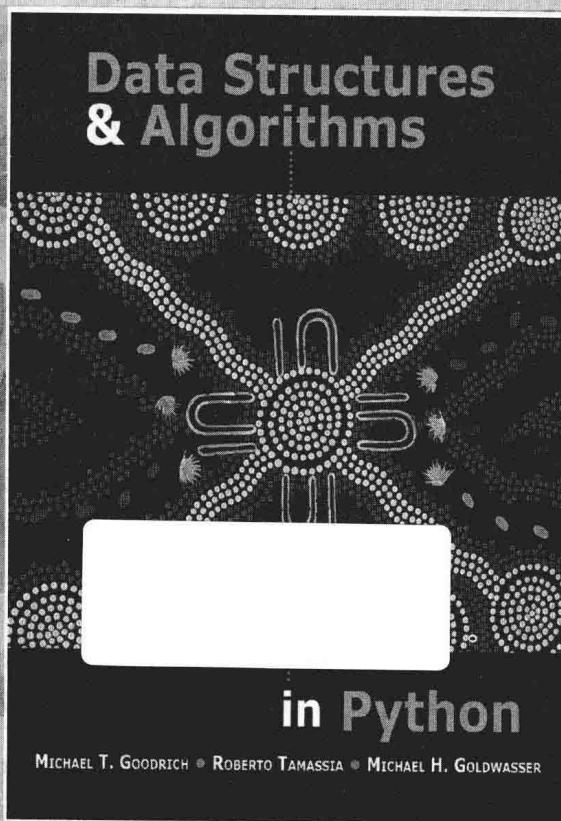
迈克尔·T. 古德里奇 (Michael T. Goodrich)

[美] 罗伯托·塔马西亚 (Roberto Tamassia) 著

迈克尔·H. 戈德瓦瑟 (Michael H. Goldwasser)

张晓 赵晓南 等译

Data Structures and Algorithms in Python



机械工业出版社
China Machine Press

图书在版编目(CIP)数据

数据结构与算法：Python语言实现 / (美) 迈克尔·T. 古德里奇 (Michael T. Goodrich) 等著；张晓等译。—北京：机械工业出版社，2018.9
(计算机科学丛书)

书名原文：Data Structures and Algorithms in Python

ISBN 978-7-111-60660-4

I. 数… II. ①迈… ②张… III. ①数据结构 ②算法分析 ③软件工具－程序设计
IV. ①TP311.12 ②TP311.561

中国版本图书馆 CIP 数据核字 (2018) 第 183790 号

本书版权登记号：图字 01-2016-6251

Copyright © 2013 John Wiley & Sons, Inc.

All rights reserved. This translation published under license. Authorized translation from the English language edition, entitled Data Structures and Algorithms in Python, ISBN 9781118290279, by Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser, Published by John Wiley & Sons. No part of this book may be reproduced in any form without the written permission of the original copyrights holder.

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

本书封底贴有 Wiley 防伪标签，无标签者不得销售。

本书采用 Python 语言讨论数据结构和算法，详细讲解其设计、分析与实现过程，是一本内容全面且特色鲜明的教材。书中将面向对象视角贯穿始终，充分利用 Python 语言优美而简洁的特点，强调代码的健壮性和可重用性，关注各种抽象数据类型以及不同算法实现策略的权衡。

本书适合作为高等院校初级数据结构或中级算法导论课程的教材，也适合相关工程技术人员阅读参考。

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：曲 煜

责任校对：李秋荣

印 刷：北京市荣盛彩色印刷有限公司

版 次：2018 年 9 月第 1 版第 1 次印刷

开 本：185mm×260mm 1/16

印 张：30.75

书 号：ISBN 978-7-111-60660-4

定 价：109.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88378991 88361066

投稿热线：(010) 88379604

购书热线：(010) 68326294 88379649 68995259

读者信箱：hzjsj@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

文艺复兴以来，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的优势，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自 1998 年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与 Pearson, McGraw-Hill, Elsevier, MIT, John Wiley & Sons, Cengage 等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出 Andrew S. Tanenbaum, Bjarne Stroustrup, Brian W. Kernighan, Dennis Ritchie, Jim Gray, Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, Abraham Silberschatz, William Stallings, Donald E. Knuth, John L. Hennessy, Larry L. Peterson 等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专门为本书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com

电子邮件：hzjsj@hzbook.com

联系电话：(010) 88379604

联系地址：北京市西城区百万庄南街 1 号

邮政编码：100037



华章教育

华章科技图书出版中心

译者序 |

Data Structures and Algorithms in Python

数据结构是计算机科学与技术专业的核心课程，是程序设计、编译原理、数据库等课程的基础。对于从事计算机应用尤其是软件开发的工程技术人员而言，掌握数据结构的相关知识和常用算法，对提高开发效率和编程质量都有着非常重要的作用。国内外有很多优秀的数据结构教材，而且有基于 C、C++、Java 等多种程序设计语言编写的版本，但是采用 Python 语言描述的并不多见。

Python 是一种面向对象的直译式计算机程序设计语言，语法简洁清晰，类库丰富强大。由于代码的平台无关性以及极简的编程思想，Python 近年来成为国内外各大科研院校和 IT 企业在教学活动、科学研究以及应用软件开发中频繁使用的程序设计语言。例如，卡内基·梅隆大学的编程基础课程、麻省理工学院的计算机科学及编程导论课程就使用 Python 语言讲授。我们西北工业大学也开设了 Python 程序设计的选修课，受到学生的热烈欢迎。在实践领域，NumPy、SciPy 等是利用 Python 语言开发的用于科学计算的工具包，著名的计算机视觉库 OpenCV、三维可视化库 VTK 等也是使用 Python 开发的。在 TIOBE 编程语言排行榜上，Python 排名第五，排在 Java、C、C++ 和 C# 之后。Python 语言也在大数据分析、网络爬虫、量化投资等新兴热点领域广泛使用。

对于计算机专业的学生和计算机应用行业的从业人员而言，从 Python 开始学习程序设计和数据结构入门门槛低，学习曲线平缓。国内已有大量介绍 Python 程序设计的书籍，但多局限在 Python 语法和特定软件包的使用方面。本书是难得的系统讲解如何使用 Python 语言设计并实现数据结构和基本算法的书籍。

本书作者 Goodrich 教授、Tamassia 教授等人先后撰写了《Data Structures and Algorithms in Java》和《Data Structures and Algorithms in C++》等书籍，对数据结构和常用算法的理解非常透彻。但本书并不是简单地将这些书籍中的代码描述部分替换成 Python 语言，而是充分利用 Python 语言的优势，以完整代码的方式实现了各种算法和数据结构。在此基础上，还大量介绍了 Python 语言内建的数据类型和一些常用基本库及接口的相关知识。

书中介绍的数据结构和算法包括完整的设计、分析和实现过程，非常适合作为初级数据结构课程的教材，同时也可作为中级算法导论课程的教材，或作为计算机基础知识有限的工程技术人员的参考书。通过学习本书，读者能够更加灵活、高效地运用 Python 语言编写满足自己需求的程序。

非常荣幸有机会翻译这样一本优秀的教材。对于书中的专业术语，我们尽量沿用了现有的习惯翻译。不过由于时间和水平所限，难免出现错误和不当之处，恳切希望广大读者不吝批评指正。

最后，感谢作者为我们呈现了一本优秀的教材；感谢出版社的信任，将这项有趣而又有意义的工作交给我们完成；还要感谢所有参与本书翻译和校对工作的教师和研究生，他们是赵晓南、王蕾、赵楠、陈震、卜海龙、柳春懿、孔兰昕、朱顺意等。

张晓

2018 年 4 月

于启真湖畔

高效数据结构的设计与分析，长期以来一直被认为是计算领域的一个重要主题，同时也是计算机科学与计算机工程本科教学中的核心课程。本书介绍数据结构和算法，包括其设计、分析和实现，可在初级数据结构或中级算法导论课程中使用。我们随后会更详细地讨论如何在这些课程中使用本书。

为了提高软件开发的健壮性和可重用性，我们在本书中采取一致的面向对象的视角。面向对象方法的一个主要思想是数据应该被封装，然后提供访问和修改它们的方法。我们不能简单地将数据看作字节和地址的集合，数据对象是抽象数据类型（Abstract Data Type, ADT）的实例，其中包括可在这种类型的数据对象上执行的操作方法的集合。我们强调的是对于一个特定的 ADT 可能有几种不同的实现策略，并探讨这些选择的优点和缺点。我们几乎为书中的所有数据结构和算法都提供了完整的 Python 实现，并介绍了将这些实现组织为可重用的组件所必需的重要的面向对象设计模式。

通过阅读本书，读者可以：

- 对常见数据集合的抽象有一定了解（如栈、队列、表、树、图）。
- 理解生成常用数据结构的高效实现的算法策略。
- 通过理论方法和实验方法分析算法的性能，并了解竞争策略之间的权衡。
- 明智地利用编程语言库中已有的数据结构和算法。
- 拥有大多数基础数据结构和算法的具体实现经验。
- 应用数据结构和算法来解决复杂的问题。

为了达到最后一个目标，我们在书中提供了数据结构的很多应用实例，包括：文本处理系统，结构化格式（如 HTML）的标签匹配，简单的密码技术，文字频率分析，自动几何布局，霍夫曼编码，DNA 序列比对，以及搜索引擎索引。

本书特色

本书主要基于由 Goodrich 和 Tamassia 所著的《Data Structures and Algorithms in Java》，以及由 Goodrich、Tamassia 和 Mount 所著的《Data Structures and Algorithms in C++》编写而成。然而，我们并不是简单地用 Python 语言实现以上书籍的内容。为了充实内容，我们重新设计了本书：

- 对全部代码进行了重新设计，以充分利用 Python 的优势，如使用生成器迭代集合的元素。
- 在 Java 和 C++ 版本中，我们提供了很多伪代码，而本书则提供了 Python 实现的完整代码。
- 在一般情况下，ADT 被定义为与 Python 内建数据类型和 Python 的 collections 模块具有一致的接口。
- 第 5 章深入探讨了 Python 中基于动态数组的内置数据结构，如 list、tuple 和 str 类。新增的附录 A 提供了关于 str 类功能的进一步讲解。
- 重新绘制或修改了超过 450 幅插图。
- 经过新增和修订，练习的总数达到 750 个。

在线资源[⊖]

本书提供一系列丰富的在线资源，可访问以下网站获取：

www.wiley.com/college/goodrich

鼓励学生在学习本书时使用这个网址，以更有效地进行练习并提高对所学知识的认识。也欢迎教师使用本网站来帮助规划、组织和展示他们的课程材料。对于教师和学生而言，网站中包含一系列与本书主题相关的教学资源，由于它们是有附加价值的，所以一些网上资源受密码保护。

对于所有的读者，尤其是学生，我们有以下资源：

- 书中所有 Python 程序的源代码。
- 提供给教师的 PDF 讲义版 PPT（每页四张）。
- 保存所有练习提示的数据库，以练习的编号为索引。

对于使用本书的教师，我们有以下额外的教学辅助资源：

- 本书练习的答案。
- 书中所有图形和插图的彩色版本。
- PPT 和 PDF 版本的幻灯片，其中 PDF 版本为每页一张。

PPT 是完全可编辑的，教师可根据自己的课程需求进行修改。在教师使用本书作为教材时，所有的在线资源不收取额外费用。

内容和组织

书中各章节的内容循序渐进，适于教学。从 Python 编程和面向对象设计的基础开始，然后逐渐增加如算法分析和递归之类的基础技术。在本书的主体部分中，我们展示了基本的数据结构和算法，并且包括对内存管理的讨论（也是数据结构的架构基础）。本书的章节安排如下：

- 第 1 章 Python 入门
- 第 2 章 面向对象编程
- 第 3 章 算法分析
- 第 4 章 递归
- 第 5 章 基于数组的序列
- 第 6 章 栈、队列和双端队列
- 第 7 章 链表
- 第 8 章 树
- 第 9 章 优先级队列
- 第 10 章 映射、哈希表和跳跃表
- 第 11 章 搜索树
- 第 12 章 排序与选择
- 第 13 章 文本处理
- 第 14 章 图算法
- 第 15 章 内存管理和 B 树

[⊖] 关于本书教辅资源，只有使用本书作为教材的教师才可以申请。需要的读者可访问华章网站 www.hzbook.com 下载 PPT、练习答案和源代码。如果需要其他资源，可向约翰·威立出版公司北京代表处申请，电话 010-84187869，电子邮件 sliang@wiley.com。——编辑注

附录 A Python 中的字符串

附录 B 有用的数学定理

预备知识

我们假设读者至少接触过一种高级语言，如 C、C++、Python 或 Java，可以理解相关高级语言的主要概念，包括：

- 变量和表达式。
- 决策结构（if 语句和 switch 语句）。
- 迭代结构（for 循环和 while 循环）。
- 函数（无论是过程式方法还是面向对象方法）。

对于已经熟悉这些概念但还不清楚如何在 Python 中应用的读者，我们建议将第 1 章作为 Python 语言的入门。这本书主要讨论数据结构，而不是讲解 Python，因此并没有详尽介绍 Python。

直到第 2 章才开始使用 Python 中的面向对象编程，这一章对于那些 Python 新手以及熟悉 Python 但不熟悉面向对象编程的人都是有用的。

就数学背景而言，我们假定读者多少熟悉些高中数学知识。即便如此，在第 3 章中，我们先讨论了算法分析的 7 个最重要的功能。若所涉及的内容超出了这 7 个功能，则作为可选章节，用星号 (*) 标记。附录 B 对其他有用的数学定理做了总结，包括初等概率等。

计算机科学课程的设计

为了帮助教师在 IEEE/ACM 2013 的框架下设计教学课程，下表描述了本书涵盖的知识要点。

知识要点	相关章节
AL/ 基本分析	第 3 章, 4.2 节, 12.2.4 节
AL/ 算法策略	12.2.1 节, 13.2.1 节, 13.3 节, 13.4.2 节
AL/ 基本数据结构与算法	4.1.3 节, 5.5.2 节, 9.4.1 节, 9.3 节, 10.2 节, 11.1 节, 13.2 节, 第 12 章, 第 14 章的大部分内容
AL/ 高级数据结构	5.3 节, 10.4 节, 11.2 ~ 11.6 节, 12.3.1 节, 13.5 节, 14.5 节, 15.3 节
AR/ 内存系统组织和架构	第 15 章
DS/ 集合、关系和功能	10.5.1 节, 10.5.2 节, 9.4 节
DS/ 证明技巧	3.4 节, 4.2 节, 5.3.2 节, 9.3.6 节, 12.4.1 节
DS/ 基础计数	2.4.2 节, 6.2.2 节, 12.2.4 节, 8.2.2 节, 附录 B
DS/ 图和树	第 8 章和第 14 章的大部分内容
DS/ 离散概率	1.11 节, 10.2 节, 10.4.2 节, 12.3.1 节
PL/ 面向对象编程	本书的大部分内容，特别是第 2 章以及 7.4 节、9.5.1 节、10.1.3 节和 11.2 节
PL/ 函数式编程	1.10 节
SDF/ 算法和设计	2.1 节, 3.3 节, 12.2.1 节
SDF/ 基本编程概念	第 1 章, 第 4 章
SDF/ 基本数据结构	第 6 章, 第 7 章, 附录 A, 1.2.1 节, 5.2 节, 5.4 节, 9.1 节, 10.1 节
SDF/ 开发方法	1.7 节, 2.2 节
SE/ 软件设计	2.1 节, 2.1.3 节

致 谢 |

Data Structures and Algorithms in Python

许多人帮助我们完成了本书。首先要感谢的是 Wiley 这个优秀的团队，感谢我们的编辑 Beth Golub 从始至终对这个项目的热情支持。从最初阶段的提议到通过广泛同行评审的过程中，Elizabeth Mills 和 Katherine Willis 的努力是推动项目持续前进的关键动力。我们非常感谢专注于细节的 Julie Kennedy，她也是本书的文字编辑。最后，非常感谢 Joyce Poh 对于最后几个月的生产过程的管理。

真心感谢评审人员和广大读者，他们丰富的评论、邮件和具有建设性的批评对我们写作本书价值很大。我们要感谢以下评审人员：Claude Anderson (Rose Hulman Institute of Technology)，Alistair Campbell (Hamilton College)，Barry Cohen (New Jersey Institute of Technology)，Robert Franks (Central College)，Andrew Harrington (Loyola University Chicago)，Dave Musicant (Carleton College)，Victor Norman (Calvin College)。特别感谢 Claude 非常负责地给我们提供了 400 条详细的建议。

感谢 David Mount (University of Maryland) 慷慨地分享了他从 C++ 版本中获得的经验。感谢 Erin Chambers 和 David Letscher (Saint Louis University) 在多年数据结构教学中的默默奉献，以及基于本书早期 Python 代码版本的评论。感谢 David Zampino (Loyola University Chicago 的学生)，他在使用本书草稿独立学习后反馈了有益的建议，还要感谢 Andrew Harrington 一直督促着 David 完成学习。

很多同行和助教为本书先前的 C++ 和 Java 版本提供了帮助，那些贡献同样对本书有益，再次感谢他们。

最后，我们要由衷地感谢 Susan Goldwasser、Isabel Cruz、Karen Goodrich、Giuseppe Di Battista、Franco Preparata、Ioannis Tollis 以及我们的父母，他们在本书的不同准备阶段给予我们建议、鼓励和支持。我们还要感谢 Calista 和 Maya Goldwasser 关于许多插图的建议，这些建议提升了图片的艺术水准。更重要的是，有些人不断提醒着我们——生活中不止写书这一件有意义的事。没错，谢谢他们。

Michael T. Goodrich

Roberto Tamassia

Michael H. Goldwasser

Michael Goodrich 于 1987 年从普渡大学获得计算机科学博士学位，目前是加州大学欧文分校计算机科学系校长讲席教授。他之前是约翰·霍普金斯大学的教授。他是富布莱特学者，美国科学促进会（AAAS）、计算机协会（ACM）以及电气和电子工程师学会（IEEE）的会士。他还是 IEEE 计算机协会技术成就奖、ACM 卓越服务奖以及 Pond 本科教学优秀奖的获得者。

Roberto Tamassia 于 1988 年从伊利诺伊大学厄巴纳 - 香槟分校获得电子与计算机工程博士学位，目前是布朗大学计算机科学系 Plastech 教授，并担任系主任，同时兼任布朗大学几何计算中心主任。他的研究方向涵盖信息安全、密码学、统计学、算法的设计和实现、图形绘制以及计算几何学。他是 AAAS、ACM 和 IEEE 的会士。他也是 IEEE 计算机协会技术成就奖的获得者。

Michael Goldwasser 于 1997 年从斯坦福大学获得计算机科学博士学位，目前是圣路易斯大学数学和计算机科学系教授，同时兼任计算机科学项目主任。之前，他在芝加哥罗耀拉大学计算机科学系任教。他的研究方向为算法的设计与实现以及计算几何学，同时他还活跃在各种计算机科学的教育社区。

这些作者的其他著作

- M.T. Goodrich and R. Tamassia, *Data Structures and Algorithms in Java*, Wiley.
- M.T. Goodrich, R. Tamassia, and D.M. Mount, *Data Structures and Algorithms in C++*, Wiley.
- M.T. Goodrich and R. Tamassia, *Algorithm Design: Foundations, Analysis, and Internet Examples*, Wiley.
- M.T. Goodrich and R. Tamassia, *Introduction to Computer Security*, Addison-Wesley.
- M.H. Goldwasser and D. Letscher, *Object-Oriented Programming in Python*, Prentice Hall.

推荐阅读



数据结构与算法分析：Java语言描述（原书第3版）

作者：Mark Allen Weiss ISBN：978-7-111-52839-5 定价：69.00元

本书是国外数据结构与算法分析方面的经典教材，使用卓越的Java编程语言作为实现工具，讨论数据结构（组织大量数据的方法）和算法分析（对算法运行时间的估计）。

随着计算机速度的不断增加和功能的日益强大，人们对有效编程和算法分析的要求也不断增长。本书将算法分析与最有效率的Java程序的开发有机结合起来，深入分析每种算法，并细致讲解精心构造程序的方法，内容全面，缜密严格。

算法设计与应用

作者：Michael T. Goodrich等 ISBN：978-7-111-58277-9 定价：139.00元

这是一本非常棒的著作，既有算法的经典内容，也有现代专题。我期待着在我的算法课程试用此教材。我尤其喜欢内容的广度和问题的难度。

——Robert Tarjan，普林斯顿大学

Goodrich和Tamassia编写了一本内容十分广泛而且方法具有创新性的著作。贯穿本书的应用和练习为各个领域学习计算的学生提供了极佳的参考。本书涵盖了超出一学期课程可以讲授的内容，这给教师提供了很大的选择余地，同时也给学生提供了很好的自学材料。

——Michael Mitzenmacher，哈佛大学

目 录 |

Data Structures and Algorithms in Python

出版者的话
译者序
前言
致谢
作者简介

第 1 章 Python 入门	1
1.1 Python 概述	1
1.1.1 Python 解释器	1
1.1.2 Python 程序预览	1
1.2 Python 对象	2
1.2.1 标识符、对象和赋值语句	2
1.2.2 创建和使用对象	4
1.2.3 Python 的内置类	4
1.3 表达式、运算符和优先级	8
1.4 控制流程	12
1.4.1 条件语句	12
1.4.2 循环语句	14
1.5 函数	16
1.5.1 信息传递	17
1.5.2 Python 的内置函数	19
1.6 简单的输入和输出	20
1.6.1 控制台输入和输出	21
1.6.2 文件	21
1.7 异常处理	22
1.7.1 抛出异常	23
1.7.2 捕捉异常	24
1.8 迭代器和生成器	26
1.9 Python 的其他便利特点	28
1.9.1 条件表达式	29
1.9.2 解析语法	29
1.9.3 序列类型的打包和解包	30
1.10 作用域和命名空间	31
1.11 模块和 import 语句	32
1.12 练习	34
扩展阅读	36

第 2 章 面向对象编程	37
2.1 目标、原则和模式	37
2.1.1 面向对象的设计目标	37
2.1.2 面向对象的设计原则	38
2.1.3 设计模式	39
2.2 软件开发	40
2.2.1 设计	40
2.2.2 假代码	41
2.2.3 编码风格和文档	42
2.2.4 测试和调试	43
2.3 类定义	44
2.3.1 例子：CreditCard 类	45
2.3.2 运算符重载和 Python 的特殊方法	48
2.3.3 例子：多维向量类	50
2.3.4 迭代器	51
2.3.5 例子：Range 类	52
2.4 继承	53
2.4.1 扩展 CreditCard 类	54
2.4.2 数列的层次图	57
2.4.3 抽象基类	60
2.5 命名空间和面向对象	62
2.5.1 实例和类命名空间	62
2.5.2 名称解析和动态调度	65
2.6 深拷贝和浅拷贝	65
2.7 练习	67
扩展阅读	70
第 3 章 算法分析	71
3.1 实验研究	71
3.2 本书使用的 7 种函数	74
3.2.1 常数函数	74
3.2.2 对数函数	74
3.2.3 线性函数	75
3.2.4 $n \log n$ 函数	75
3.2.5 二次函数	76

3.2.6 三次函数和其他多项式	77	5.4 Python 序列类型的效率	130
3.2.7 指数函数	77	5.4.1 Python 的列表和元组类	130
3.2.8 比较增长率	79	5.4.2 Python 的字符串类	134
3.3 渐近分析	79	5.5 使用基于数组的序列	136
3.3.1 大 O 符号	80	5.5.1 为游戏存储高分	136
3.3.2 比较分析	82	5.5.2 为序列排序	138
3.3.3 算法分析示例	84	5.5.3 简单密码技术	140
3.4 简单的证明技术	89	5.6 多维数据集	142
3.4.1 示例	89	5.7 练习	145
3.4.2 反证法	89	扩展阅读	147
3.4.3 归纳和循环不变量	90		
3.5 练习	91		
扩展阅读	95		
第 4 章 递归	96		
4.1 说明性的例子	96		
4.1.1 阶乘函数	96		
4.1.2 绘制英式标尺	97		
4.1.3 二分查找	99		
4.1.4 文件系统	101		
4.2 分析递归算法	104		
4.3 递归算法的不足	106		
4.4 递归的其他例子	109		
4.4.1 线性递归	109		
4.4.2 二路递归	112		
4.4.3 多重递归	113		
4.5 设计递归算法	114		
4.6 消除尾递归	115		
4.7 练习	116		
扩展阅读	118		
第 5 章 基于数组的序列	119		
5.1 Python 序列类型	119		
5.2 低层次数组	119		
5.2.1 引用数组	121		
5.2.2 Python 中的紧凑数组	122		
5.3 动态数组和摊销	124		
5.3.1 实现动态数组	126		
5.3.2 动态数组的摊销分析	127		
5.3.3 Python 列表类	130		
第 6 章 栈、队列和双端队列	148		
6.1 栈	148		
6.1.1 栈的抽象数据类型	148		
6.1.2 简单的基于数组的栈	149		
实现	149		
6.1.3 使用栈实现数据的逆置	152		
6.1.4 括号和 HTML 标记匹配	152		
6.2 队列	155		
6.2.1 队列的抽象数据类型	155		
6.2.2 基于数组的队列实现	156		
6.3 双端队列	160		
6.3.1 双端队列的抽象数据类型	160		
6.3.2 使用环形数组实现双端	161		
队列	161		
6.3.3 Python collections 模块中的	162		
双端队列	162		
6.4 练习	163		
扩展阅读	165		
第 7 章 链表	166		
7.1 单向链表	166		
7.1.1 用单向链表实现栈	169		
7.1.2 用单向链表实现队列	171		
7.2 循环链表	173		
7.2.1 轮转调度	173		
7.2.2 用循环链表实现队列	174		
7.3 双向链表	175		
7.3.1 双向链表的基本实现	177		
7.3.2 用双向链表实现双端队列	179		

7.4 位置列表的抽象数据类型	180	9.2 优先级队列的实现	237
7.4.1 含位置信息的列表抽象数据 类型	182	9.2.1 组合设计模式	237
7.4.2 双向链表实现	183	9.2.2 使用未排序列表实现优先级 队列	238
7.5 位置列表的排序	186	9.2.3 使用排序列表实现优先级 队列	239
7.6 案例研究：维护访问频率	186	9.3 堆	241
7.6.1 使用有序表	187	9.3.1 堆的数据结构	241
7.6.2 启发式动态调整列表	188	9.3.2 使用堆实现优先级队列	242
7.7 基于链接的序列与基于数组的 序列	190	9.3.3 基于数组的完全二叉树表示	244
7.8 练习	192	9.3.4 Python 的堆实现	246
扩展阅读	195	9.3.5 基于堆的优先级队列的分析	248
第 8 章 树	196	9.3.6 自底向上构建堆 *	248
8.1 树的基本概念	196	9.3.7 Python 的 heapq 模块	251
8.1.1 树的定义和属性	196	9.4 使用优先级队列排序	252
8.1.2 树的抽象数据类型	199	9.4.1 选择排序和插入排序	253
8.1.3 计算深度和高度	201	9.4.2 堆排序	254
8.2 二叉树	203	9.5 适应性优先级队列	255
8.2.1 二叉树的抽象数据类型	204	9.5.1 定位器	256
8.2.2 二叉树的属性	206	9.5.2 适应性优先级队列的实现	256
8.3 树的实现	207	9.6 练习	259
8.3.1 二叉树的链式存储结构	207	扩展阅读	263
8.3.2 基于数组表示的二叉树	212	第 10 章 映射、哈希表和跳跃表	264
8.3.3 一般树的链式存储结构	214	10.1 映射和字典	264
8.4 树的遍历算法	214	10.1.1 映射的抽象数据类型	264
8.4.1 树的先序和后序遍历	214	10.1.2 应用：单词频率统计	266
8.4.2 树的广度优先遍历	216	10.1.3 Python 的 MutableMapping 抽象基类	267
8.4.3 二叉树的中序遍历	216	10.1.4 我们的 MapBase 类	267
8.4.4 用 Python 实现树遍历	217	10.1.5 简单的非有序映射实现	268
8.4.5 树遍历的应用	220	10.2 哈希表	269
8.4.6 欧拉图和模板方法模式 *	223	10.2.1 哈希函数	270
8.5 案例研究：表达式树	227	10.2.2 哈希码	271
8.6 练习	230	10.2.3 压缩函数	274
扩展阅读	235	10.2.4 冲突处理方案	274
第 9 章 优先级队列	236	10.2.5 负载因子、重新哈希和 效率	276
9.1 优先级队列的抽象数据类型	236	10.2.6 Python 哈希表的实现	278
9.1.1 优先级	236	10.3 有序映射	281
9.1.2 优先级队列的抽象数据类型的 实现	236	10.3.1 排序检索表	282

10.3.2 有序映射的两种应用	286	12.2.2 基于数组的归并排序的实现	351
10.4 跳跃表	288	12.2.3 归并排序的运行时间	353
10.4.1 跳跃表中的查找和更新操作	289	12.2.4 归并排序与递归方程 *	354
10.4.2 跳跃表的概率分析 *	292	12.2.5 归并排序的可选实现	355
10.5 集合、多集和多映射	294	12.3 快速排序	357
10.5.1 集合的抽象数据类型	294	12.3.1 随机快速排序	361
10.5.2 Python 的 MutableSet 抽象基类	295	12.3.2 快速排序的额外优化	362
10.5.3 集合、多集和多映射的实现	297	12.4 再论排序：算法视角	364
10.6 练习	298	12.4.1 排序下界	365
扩展阅读	302	12.4.2 线性时间排序：桶排序和基数排序	366
第 11 章 搜索树	303	12.5 排序算法的比较	367
11.1 二叉搜索树	303	12.6 Python 的内置排序函数	369
11.1.1 遍历二叉搜索树	303	12.7 选择	370
11.1.2 搜索	305	12.7.1 剪枝搜索	370
11.1.3 插入和删除	306	12.7.2 随机快速选择	371
11.1.4 Python 实现	307	12.7.3 随机快速选择分析	371
11.1.5 二叉搜索树的性能	311	12.8 练习	372
11.2 平衡搜索树	312	扩展阅读	376
11.3 AVL 树	316		
11.3.1 更新操作	318		
11.3.2 Python 实现	320		
11.4 伸展树	322		
11.4.1 伸展	322		
11.4.2 何时进行伸展	323		
11.4.3 Python 实现	324		
11.4.4 伸展树的摊销分析 *	325		
11.5 (2, 4) 树	328		
11.5.1 多路搜索树	328		
11.5.2 (2, 4) 树的操作	330		
11.6 红黑树	334		
11.6.1 红黑树的操作	335		
11.6.2 Python 实现	341		
11.7 练习	343		
扩展阅读	348		
第 12 章 排序与选择	349		
12.1 为什么要学习排序算法	349		
12.2 归并排序	349		
12.2.1 分治法	349		
12.2.2 基于数组的归并排序的实现	351		
12.2.3 归并排序的运行时间	353		
12.2.4 归并排序与递归方程 *	354		
12.2.5 归并排序的可选实现	355		
12.3 快速排序	357		
12.3.1 随机快速排序	361		
12.3.2 快速排序的额外优化	362		
12.4 再论排序：算法视角	364		
12.4.1 排序下界	365		
12.4.2 线性时间排序：桶排序和基数排序	366		
12.5 排序算法的比较	367		
12.6 Python 的内置排序函数	369		
12.7 选择	370		
12.7.1 剪枝搜索	370		
12.7.2 随机快速选择	371		
12.7.3 随机快速选择分析	371		
12.8 练习	372		
扩展阅读	376		
第 13 章 文本处理	377		
13.1 数字化文本的多样性	377		
13.2 模式匹配算法	378		
13.2.1 穷举	378		
13.2.2 Boyer-Moore 算法	379		
13.2.3 Knuth-Morris-Pratt 算法	382		
13.3 动态规划	385		
13.3.1 矩阵链乘积	385		
13.3.2 DNA 和文本序列比对	386		
13.4 文本压缩和贪心算法	389		
13.4.1 霍夫曼编码算法	390		
13.4.2 贪心算法	391		
13.5 字典树	391		
13.5.1 标准字典树	391		
13.5.2 压缩字典树	394		
13.5.3 后缀字典树	395		
13.5.4 搜索引擎索引	396		
13.6 练习	397		
扩展阅读	400		

第 14 章 图算法	401
14.1 图	401
14.2 图的数据结构	405
14.2.1 边列表结构	406
14.2.2 邻接列表结构	407
14.2.3 邻接图结构	408
14.2.4 邻接矩阵结构	409
14.2.5 Python 实现	409
14.3 图遍历	412
14.3.1 深度优先搜索	413
14.3.2 深度优先搜索的实现和 扩展	416
14.3.3 广度优先搜索	419
14.4 传递闭包	421
14.5 有向非循环图	424
14.6 最短路径	426
14.6.1 加权图	427
14.6.2 Dijkstra 算法	428
14.7 最小生成树	434
14.7.1 Prim-Jarník 算法	435
14.7.2 Kruskal 算法	438
14.7.3 不相交分区和联合查找 结构	442
14.8 练习	445
拓展阅读	451
第 15 章 内存管理和 B 树	452
15.1 内存管理	452
15.1.1 内存分配	452
15.1.2 垃圾回收	453
15.1.3 Python 解释器使用的额外 内存	455
15.2 存储器层次结构和缓存	456
15.2.1 存储器系统	456
15.2.2 高速缓存策略	456
15.3 外部搜索和 B 树	460
15.3.1 (a, b) 树	460
15.3.2 B 树	462
15.4 外部存储器中的排序	462
15.5 练习	464
拓展阅读	465
附录 A Python 中的字符串	466
附录 B 有用的数学定理	469
参考文献	474

Python 入门

1.1 Python 概述

构建数据结构和算法需要我们了解计算机中详细的指令。一种很好的方法是使用高级计算机语言描述这个了解的过程，如 Python。Python 编程语言最初是由 Guido van Rossum 于 20 世纪 90 年代初开发的，并已在工业和教育领域成为一门十分重要的语言。Python 语言的第二个主要版本 Python 2 于 2000 年发布，而第三个主要版本 Python 3 于 2008 年发布。Python 2 和 Python 3 之间不兼容。本书是基于 Python 3 编写的（更具体地说，基于 Python 3.1 或更高版本）。Python 语言的最新版本及其文档和教程可在 www.python.org 免费获取。

在本章中，我们对 Python 编程语言进行了概述，并将在下一章继续讨论面向对象原则。我们假设这本书的读者已有一定的编程经验，但不一定局限于 Python。本书不提供 Python 语言的完整描述（有许多语言可以参考用于实现这一目的），但它的确介绍了使用的代码片段里所用语言的方方面面。

1.1.1 Python 解释器

Python 是一种解释语言。命令通常在被称为 Python 解释器的软件中执行。Python 解释器接收到一条命令，然后评估该命令，最后返回该命令的结果。解释器可以交互使用（尤其是在调试时），程序员通常提前定义一系列命令，然后把这些命令保存为纯文本文件，这些程序被称为源代码或脚本。对于 Python，源代码通常存储在一个扩展名为 .py 的文件中（例如 demo.py）。

在大多数操作系统中，Python 解释器可以通过在命令行中输入“python”启动。在默认情况下，解释器在交互模式下使用新的工作空间启动。执行命令时，从保存在文件中的一个预定义脚本（例如 demo.py）中把文件名作为调用解释器执行的一个参数（例如 python demo.py），或使用一个额外的 -i 标志来执行脚本，然后进入交互模式（例如 python -i demo.py）。

许多集成开发环境（Integrated Development Environments, IDE）为 Python 提供了更加丰富的软件开发平台，包括一个拥有标准 Python 发行版的 IDLE。IDLE 提供了一个嵌入式的文本编辑器（可显示和编辑 Python 代码），以及一个基本调试器（允许逐步执行程序，以便检查关键变量的值）。

1.1.2 Python 程序预览

下面的代码段 1-1 是一个 Python 程序，用户输入字母表示学生的成绩等级，而后程序由输入数据计算学生平均绩点（Grade-Point Average, GPA）。这个例子所采用的许多技术将在本章的其余部分讨论。这时，我们注意到一些高层次的问题，尤其是对于那些初次接触 Python 这门编程语言的读者。

代码段 1-1 计算学生平均绩点 (GPA) 的 Python 代码

```

print('Welcome to the GPA calculator.')
print('Please enter all your letter grades, one per line.')
print('Enter a blank line to designate the end.')
# map from letter grade to point value
points = {'A+':4.0, 'A':4.0, 'A-':3.67, 'B+':3.33, 'B':3.0, 'B-':2.67,
          'C+':2.33, 'C':2.0, 'C-':1.67, 'D+':1.33, 'D':1.0, 'F':0.0}
num_courses = 0
total_points = 0
done = False
while not done:
    grade = input()                      # read line from user
    if grade == '':                       # empty line was entered
        done = True
    elif grade not in points:             # unrecognized grade entered
        print("Unknown grade '{0}' being ignored".format(grade))
    else:
        num_courses += 1
        total_points += points[grade]
if num_courses > 0:                      # avoid division by zero
    print('Your GPA is {0:.3}'.format(total_points / num_courses))

```

Python 的语法在很大程度上依赖于缩进。典型的写法是将一条语句写在一行，当然，也可以将一条命令写在多行，如利用反斜杠字符 (\)，或者使用“开”分隔符，比如定义值映射 (value-map) 的 { 字符。

在 Python 划定控制结构的主体时，可以用空白字符进行缩进。具体来说，代码块缩进到其指定的控制体结构内，嵌套控制结构使用空白缩进保持代码整洁。在代码段 1-1 中，while 循环主体之后的 8 行，包括嵌套的条件结构都使用空白进行了缩进。

Python 解释器会忽略代码中的注释。在 Python 中，注释是以 # 字符标识的，# 表示该行的剩余部分是注释。

1.2 Python 对象

Python 是一种面向对象的语言，类则是所有数据类型的基础。在本节中，我们将介绍 Python 对象模型的重要方面，并介绍 Python 的内置类，如对于整数的 int 类、浮点数的 float 类以及字符串的 str 类。有关面向对象更加深入的介绍将着重在第 2 章进行。

1.2.1 标识符、对象和赋值语句

在 Python 语言的所有语句中，最重要的就是赋值语句，例如

```
temperature = 98.6
```

这条语句规定 temperature 作为标识符（也称为名称）与等号右边表示的对象相关联，在这一示例中浮点对象的值为 98.6。图 1-1 描述了这种赋值操作的结果。



图 1-1 标识符 temperature 引用了 float 类的一个值为 98.6 的实例

标识符

在 Python 中，标识符是大小写敏感的，所以 temperature 和 Temperature 是不同的标识

符。标识符几乎可以由任意字母、数字和下划线字符（或更一般的 Unicode 字符）组成。主要的限制是标识符不能以数字开头（因此 9lives 是非法的名字），并且有 33 个特别的保留字不能用作标识符，见表 1-1。

表 1-1 Python 中的保留字，这些名字不能用作标识符

保留字									
False	as	continue	else	from	in	not	return	yield	
None	assert	def	except	global	is	or	try		
True	break	del	finally	if	lambda	pass	while		
and	class	elif	for	import	nonlocal	raise	with		

对于熟悉其他编程语言的读者来说，Python 标识符的语义非常类似于 Java 中的引用变量或 C++ 中的指针变量。每个标识符与其所引用的对象的内存地址隐式相关联。Python 标识符可以分配给一个名为 None 的特殊对象，这与 Java 或 C++ 中空引用的目的是相似的。

与 Java 和 C++ 不同，Python 是一种动态类型语言，标识符的数据类型并不需要事先声明。标识符可以与任何类型的对象相关联，并且它可以在以后重新分配给相同（或不同）类型的另一个对象。虽然标识符没有被声明为确切的类型，但它所引用的对象有一个明确的类型。在第一个示例中，字符 98.6 被认为是一个浮点类型，因此标识符 temperature 与具有该值的 float 类的实例相关联。

程序员可以通过向现有对象指定第二个标识符建立一个别名。继续前面的例子，图 1-2 描绘了一个赋值操作 original = temperature 的结果。

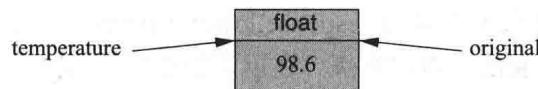


图 1-2 标识符 temperature 和 original 是同一个对象的别名

一旦建立了别名，两个名称都可用来访问底层对象。如果该对象支持影响其状态的行为，当使用一个别名而通过另一个别名更改对象的行为，其结果是显而易见的（因为它们指的是相同的对象）。然而，如果对象的一个别名被赋值语句重新赋予了新的值，那么这并不影响已存在的对象，而是给别名重新分配了存储对象。继续之前的示例，我们考虑下面的语句：

```
temperature = temperature + 5.0
```

这条语句的执行首先从 = 操作符右边的表达式开始。表达式 temperature + 5.0 是基于已存在的对象名 temperature 进行运算，因此，结果的值为 103.6，即 98.6 + 5。该结果被作为新的浮点实例存储，如果赋值语句左边的名称是 temperature，那么（重新）分配存储对象。随后的配置如图 1-3 所示。特别值得注意的是，后面这条语句对标识符 original 继续引用现有的浮点型实例的值没有任何影响。

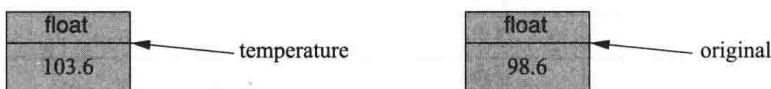


图 1-3 temperature 标识符已分配了新的值，而 original 继续引用以前已有的值

1.2.2 创建和使用对象

实例化

创建一个类的新实例的过程被称为实例化。一般来说，通过调用类的构造函数来实例化对象。例如，如果有一个名为 `Widget` 的类，假设这个构造函数不需要任何参数，我们可以使用如 `w = Widget()` 这样的语句来创建这个类的实例。如果构造函数需要参数，我们可以使用诸如 `Widget(a, b, c)` 的语句来构造一个新的实例。

许多 Python 的内置类（在 1.2.3 节中讨论）都支持所谓的字面形式指定新的实例。例如，语句 `temperature = 98.6` 的结果是创建 `float` 类的新实例。在该表达式中，`98.6` 这个词是字面形式。我们将在接下来的部分进一步讨论 Python 的字面形式。

从程序员的角度来看，另一种间接创建类的实例的方法是调用一个函数来创建和返回这样一个实例。例如，Python 有一个内置的函数名为 `Sorted`（见 1.5.2 节），它以一系列可比较的元素作为参数，并返回包含这些已排序元素的 `list` 类的一个新实例。

调用方法

Python 支持传统函数调用（见 1.5 节），调用函数的形式如 `sorted(data)`。在这种情况下，`data` 作为一个参数传递给函数。Python 的类也可以定义一个或多个方法（也称为成员函数），类的特定实例上的方法可以使用点操作符（“.”）来调用。例如，Python 的 `list` 类有一个名为 `sort` 的方法，那么可以使用 `data.sort()` 这样的形式调用。这个特殊的方法对列表中的内容进行重排，从而使其有序。

点左侧的表达式用于确认被方法调用的对象。通常，这将是一个标识符（例如 `data`），但我们可以根据其他操作的返回结果使用点操作符来调用一个方法。例如，如果 `response` 标识一个字符串实例，那么可以采用 `response.lower().startswith('y')` 的形式调用函数，其中 `response.lower()` 返回一个新的字符串实例，在返回的中间字符串的基础上调用 `startswith('y')` 方法。

当使用一个类的方法时，了解它的行为是很重要的。一些方法返回一个对象的状态信息，但是并不改变该状态。这些方法被称为访问器。其他方法，如 `list` 类的 `sort` 方法，会改变一个对象的状态。这些方法被称为应用程序或更新方法。

1.2.3 Python 的内置类

表 1-2 给出了 Python 中常用的内置类。我们要特别注意可变的类和不可变的类。如果类的每个对象在实例化时有一个固定的值，并且在随后的操作中不会被改变，那么就是不可变的类。例如，`float` 类是不可改变的。一旦一个实例被创建，它的值不能被改变（虽然一个标识符引用的对象被赋予了一个不同的值）。

表 1-2 Python 中常用的内置类

类	描述	不可变
<code>bool</code>	布尔值	✓
<code>int</code>	整数（任意大小）	✓
<code>float</code>	浮点数	✓
<code>list</code>	对象的可变序列	
<code>tuple</code>	对象的不可变序列	✓

(续)

类	描述	不可变
str	字符串	✓
set	不同对象的无序集合	
frozenset	集合类的不可改变的形式	✓
dict	关联映射(字典)	

在这一节中，我们对这些类进行了介绍，讨论了它们的目的，并且对创建类的实例提出了几种方法。大多数内置类都存在字面形式（如 98.6），所有类都支持传统构造函数形式创建基于一个或多个现有值的实例。这些类支持的操作在 1.3 节中描述。更多关于这些类的详细信息可以在如下章节中找到：列表和元组（第 5 章）；字符串（第 5 章、第 13 章和附录 A）；集合和字典（第 10 章）。

布尔类

布尔（bool）类用于处理逻辑（布尔）值，该类表示的实例只有两个值——True 和 False。默认构造函数 bool() 返回 False，但是与其使用这种语法，还不如采用更直接的表现形式。Python 允许采用 bool(foo) 语法用非布尔值类型为值 foo 创造一个布尔类型。结果取决于参数的类型。就数字而言，如果为零就为 False，否则就为 True。对于序列和其他容器类型，如字符串和列表，如果是空为 False，如果非空则为 True。这种方式的一个重要应用是可以使用非布尔类型的值作为控制结构的条件。

整型类

整型（int）和浮点（float）类是 Python 的主要数值类型。int 类被设计成可以表示任意大小的整型值。不像 Java 和 C++ 支持不同精度的不同整数类型（如 int、short、long），Python 会根据其整数的大小自动选择内部表示的方式。对于整数，典型的形式包括 0、137 和 -23。在某些情况下，可以很方便地使用二进制、八进制或十六进制表示一个整型值。可以使用 0 这个前缀和一个字符来描述这些进制形式。这样的例子分别有 0b1011、0o52 和 0x7F。

整数的构造函数 int() 返回一个默认为 0 的值。该构造函数可用于构造基于另一类型的值的整数值。例如，如果 f 是一个浮点值，表达式 int(f) 得到 f 的整数部分。例如，int(3.14) 和 int(3.99) 得到的结果都是 3，而 int(-3.9) 得到的结果是 -3。构造函数也可以用来分析一个字符串（例如用户输入的一个字符串），该字符串被假定为表示整型值。如果 s 是一个字符串，那么 int(s) 得到这个字符串代表的整数值。例如，表达式 int('137') 产生整数值 137。如果一个无效的字符串作了参数，如 int('hello')，那么就会产生一个 ValueError（见 1.7 节讨论的 Python 异常）。默认情况下，该字符串必须使用十进制。如果需要从不同的进制中转换，那么需要把进制表示为第二个可选参数。例如，表达式 int('7f', 16) 计算结果为整数 127。

浮点类

浮点（float）类是 Python 中唯一的浮点类型，使用固定精度表示。其精度更像是 Java 或 C++ 中的 double 型，而不是 Java 或 C++ 中的 float 型。我们已经讨论了一个典型的形式——98.6。我们注意到，整数的等价浮点形式可以直接表达成 2.0。从技术上讲，数字末尾的零是可选的，所以有些程序员可以使用表达式 2. 来表示数字 2.0 的浮点形式。浮点型数据的另一种表达形式是采用科学计数法。例如，表达式 6.022e23 代表数学上的

6.022×10^{23} 。

float() 构造函数的返回值是 0.0。当给定一个参数时，float() 构造函数尝试返回等价的浮点值。例如，调用函数 float(2) 返回浮点值 2.0。如果构造函数的参数是一个字符串，如 float('3.14')，它试图将字符串解析为浮点值，那么将会产生 ValueError 的异常。

序列类型：列表、元组和 str 类

list、tuple 和 str 类是 Python 中的序列类型，代表许多值的集合，集合中值的顺序很重要。list 类是最常用的，表示任意对象的序列（类似于其他语言中的“数组”）。tuple 类是 list 类的一个不可改变的版本，可以看作列表类一种简化的内部表示。str 类表示文本字符不可变的序列。我们注意到，Python 没有为字符设计一个单独的类，可以将其看作长度为 1 的字符串。

列表类

列表（list）实例存储对象序列。列表是一个参考结构，因为它在技术上存储其元素的引用序列（见图 1-4）。列表的元素可以是任意的对象（包括 None 对象）。列表是基于数组的序列，采用零索引，因此一个长度为 n 的列表包含索引号从 0 到 $n - 1$ 的元素。列表也许是 Python 中最常用的容器类型，对数据结构和算法的研究极其重要。它们有很多有用的操作，还具备随着需求动态扩展和收缩存储容量的能力。在本章中，我们将讨论列表最基本的性质。第 5 章将重点审视 Python 中所有序列类型的内部工作。

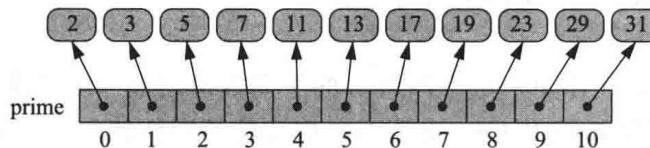


图 1-4 Python 中整数列表的内部表示，实例化为 prime = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]，元素的隐式索引显示在每一个条目的下方

Python 使用字符 [] 作为列表的分隔符，[] 本身表示一个空列表。作为另一个示例，['red', 'green', 'blue'] 是含有三个字符串实例的列表。列表中的内容并不需要在字面上表达出来。如果标识符 a 和 b 已经声明，则语法 [a, b] 是合法的。

list() 构造函数默认产生一个空的列表。然而，构造函数可以接受任何可迭代类型的参数。我们将在 1.8 节进一步讨论迭代，但迭代器类型的例子包括所有的标准容器类型（如字符串、列表、元组、集合、字典）。例如：list('hello') 产生一个单个字符的列表，['h', 'e', 'l', 'l', 'o']。因为现有列表本身可迭代，语法 backup = list(data) 可用于构造一个新的列表实例，该列表实例引用与 data 相同的内容作为原始列表元素。

元组类

元组（tuple）类是序列的一个不可改变的版本，它的实例中有一个比列表更为精简的内部表示。Python 使用 [] 符号表示列表，而使用圆括号表示元组，() 代表一个空的元组。这里有一个重要的细节——为了表示只有一个元素的元组，该元素之后必须有一个逗号并且在圆括号之内。例如，(17,) 是一元元组。之所以这么做，是因为如果没有添加后面的逗号，那么表达式 (17) 会被看作一个简单的带括号的数值表达式。

str 类

Python 的 str 类专门用来有效地代表一种不变的字符序列，它基于 Unicode 国际字符

集。相较于引用列表和元组，字符串有更为紧凑的内部表示，如图 1-5 所示。

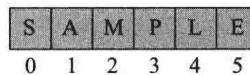


图 1-5 一个 Python 字符串，它是字符的一个索引序列

字符串可以用单引号括起来，如 'hello'，或双引号括起来，如 "hello"。这种选择很方便，特别是在序列中使用另一个引号字符作为一个实际字符时，如 "Don't worry"。另外，引号的分隔作用可以用反斜杠来实现，即所谓的转义字符，如 'Don't worry'。因为反斜杠可以实现这个目的，它在字符串中正常使用时也应该遵循这个用法，如 'C:\\Python\\'，它实际所要表达的字符串是 C:\Python\。其他常用的转义字符有 \n（表示换行）和 \t（表示制表符）。Unicode 字符也包括在内，如 '20 \u20AC' 表示字符串 20 €。

Python 也支持在字符串的首尾使用分割符 "" 或者 """。这样使用三重引号字符的优点是换行符可以在字符串中自然出现（而不是使用转义字符 \n）。这可以大大提高源代码中长字符串的可读性。例如，在代码段 1-1 的开始，相较于使用单独的输出语句逐行输出介绍词，我们可以使用一个输出语句，如下：

```
print("""Welcome to the GPA calculator.
Please enter all your letter grades, one per line.
Enter a blank line to designate the end.""")
```

set 和 frozenset 类

Python 的 set 类代表一个集合的数学概念，即许多元素的集合，集合中没有重复的元素，而且这些元素没有内在的联系。与列表恰恰相反，使用集合的主要优点是它有一个高度优化的方法来检查特定元素是否包含在集合内。这基于一个名为散列表的数据结构（这将是第 10 章的主题）。然而，这里有两个由算法基础产生的重要限制。一是该集合不保存任何有特定顺序的元素集。二是只有不可变类型的实例才可以被添加到一个 Python 集合。因此，如整数、浮点数和字符串类型的对象才有资格成为集合中的元素。有可能出现元组的集合，但不会有列表组成的集合或集合组成的集合，因为列表和集合是可变的。frozenset 类是集合类型的一种不可变的形式，所以由 frozensets 类型组成的集合是合法的。

Python 使用花括号 { 和 } 作为集合的分隔符，例如，{17} 或 {'red', 'green', 'blue'}。这个规则的特例是 {} 并不代表一个空的集合；由于历史的原因，{} 代表一个空的字典（见下文）。除此之外，构造函数 set() 会产生一个空集合。如果给构造函数提供可迭代的参数，那么就会产生不同元素组成的集合。例如，set('hello') 产生集合 {'h', 'e', 'l', 'o'}。

字典类

Python 的 dict 类代表一个字典或者映射，即从一组不同的键中找到对应的值。例如，字典可以把学生的唯一的学号信息和大量的学生记录（如学生的姓名、地址和课程成绩）进行一一映射。Python 实现 dict 类与实现集合类采用的方法几乎相同，只不过实现字典类时会同时存储键对应的值。

字典的表达形式也使用花括号，因为在 Python 中字典类型是早于集合类型出现的，字面符号 {} 产生一个空的字典。一个非空字典的表示是用逗号分隔一系列的键值对。例如，字典 {'ga': 'Irish', 'de': 'German'} 表示 'ga' 到 'Irish' 和 'de' 到 'German' 的一一映射。

dict 类的构造函数接受一个现有的映射作为参数，在这种情况下，它创造了一个与原有

字典具有相同联系的新字典。另外，构造函数接受一系列键值对作为参数，如 `dict(pairs)` 中的 `pairs = [('ga', 'Irish'), ('de', 'German')]`。

1.3 表达式、运算符和优先级

在前面的小节中，我们演示了如何使用标识符来标识现有的对象，以及如何使用文字和构造函数创建内部类的实例。在使用运算符（即各种特殊符号和关键词）的情况下，现有的值可以组合成较大的语法表达式。运算符（或称操作符）的语义取决于其操作数的类型。例如，当 `a` 和 `b` 是数字，语句 `a + b` 表示相加；如果 `a` 和 `b` 是字符串，那么运算符就表示字符串的连接。本节中，我们在内置类型的不同上下文语义中描述 Python 的运算符。

我们将在稍后讨论复合表达式，例如 `a + b * c`，表达式的结果取决于两个或更多的运算符运算的结果。复合表达式的运算顺序可以影响表达式的整体结果。为此，Python 定义运算符的优先级顺序，但允许程序员通过使用明确的括号对表达式中运算符的优先级进行调整。

逻辑运算符

Python 支持以下关键字作为运算符，其结果为布尔值：

<code>not</code>	逻辑非
<code>and</code>	逻辑与
<code>or</code>	逻辑或

`and` 和 `or` 运算符是短路保护的，也就是说，如果其结果可以根据第一个操作数的值来确定，那么它们不会对第二个操作数进行运算。这个功能在构造布尔表达式时很有用。我们首先测试某些条件成立（如一个引用不是 `None`），然后测试另一个条件，否则可能产生一个之前的测试没有成功的错误条件。

相等运算符

Python 支持以下运算符去测试两个概念的相等性：

<code>is</code>	同一实体
<code>is not</code>	不同的实体
<code>==</code>	等价
<code>!=</code>	不等价

当标识符 `a` 和 `b` 是同一个对象的别名时，表达式 `a is b` 的结果为真。表达式 `a == b` 测试一个更一般的等价概念。如果标识符 `a` 和 `b` 指向同一个对象，那么表达式 `a == b` 为真。如果标识符指向不同的对象，但这些对象的值被认为是等价的，那么 `a == b` 的结果也为真。精确的等价概念取决于数据类型。例如，对于两个字符串，如果它们的每个字符都对应相同，那么它们可以看作是等价的。两个集合包含相同的元素，而不考虑其顺序，那么这两个集合可以看作是等价的。在大多数编程情况下，`==` 和 `!=` 运算符适用于检验表达式是否相等。`is` 和 `is not` 在有必要检验真正的混叠时是适用的。

比较运算符

数据类型可以通过以下运算符定义一个自然次序：

<code><</code>	小于
<code><=</code>	小于等于
<code>></code>	大于
<code>>=</code>	大于等于

这些运算符对于数值类型、定义好的字典类型和有大小写之分的字符串有可预期的结果。如果操作数的类型不匹配，例如 `5 < 'hello'`，那么就会产生异常。

算术运算符

Python 支持以下算术运算符：

<code>+</code>	加
<code>-</code>	减
<code>*</code>	乘
<code>/</code>	真正的除
<code>//</code>	整数除法
<code>%</code>	模运算符

加法、减法和乘法的用法是很简单的，需要注意的是，如果两个操作数都是整型，那么其结果也是整型；如果有一个是浮点型，或两个操作数都是浮点型，那么其结果也是浮点型。

Python 对于除法有更多的考虑。我们首先考虑两个操作数都是整型的情况，例如，27 除以 4。在数学中， $27 \div 4 = 6 \frac{3}{4} = 6.75$ 。在 Python 中，`/` 运算符表示真正的除，运算返回一个浮点型的计算结果。因此，`27 / 4` 得到一个浮点型的值 6.75。Python 支持 `//` 和 `%` 运算符进行整数运算，表达式 `27 // 4` 运算的值是整型的 6（数学概念中的商），表达式 `27 % 4` 运算的值是整型的 3，整数除法的余数。我们注意到 C、C++ 和 Java 等语言不支持 `//` 运算符。另外，当两个操作数都是整型时，`/` 运算符返回不大于商的最大整数；当至少有一个操作数是浮点类型时，其结果是真正除法的结果。

在操作数有一个或两个是负数的情况下，Python 谨慎地扩展了 `//` 和 `%` 的语义。由于符号的缘故，我们假设变量 n 和 m 分别代表商式 $\frac{n}{m}$ 的被除数和除数， $q = n // m$ 和 $r = n \% m$ 。Python 保证 $q * m + r$ 等于 n 。我们已经看到操作数为正数这一情况的实例，如 $6 * 4 + 3 = 27$ 。当除数 m 为正数时，Python 进一步保证 $0 \leq r < m$ 。因此，我们发现 $-27 // 4$ 运算的值为 -7 并且 $27 \% 4$ 运算的值为 1，满足算式 $(-7) * 4 + 1 = -27$ 。当除数为负数时，Python 保证 $m < r \leq 0$ 。作为示例， $27 // -4$ 运算的值为 -7 并且 $27 \% -4$ 运算的值为 -1，满足算式 $27 = (-7) * (-4) + (-1)$ 。

`//` 和 `%` 运算符的使用甚至扩展到浮点型操作数，表达式 $q = n // m$ 的值是不大于商的最大整数，表达式 $r = n \% m$ 表示 r 是余数，确保 $q * m + r$ 等于 n 。例如， $8.2 // 3.14$ 运算的结果为 2.0， $8.2 \% 3.14$ 运算的结果为 1.92，满足算式 $2.0 * 3.14 + 1.92 = 8.2$ 。

位运算符

Python 为整数提供了以下位运算符：

<code>~</code>	取反（前缀一元运算符）
<code>&</code>	按位与
<code> </code>	按位或
<code>^</code>	按位异或
<code><<</code>	左移位，用零填充
<code>>></code>	右移位，按符号位填充

序列运算符

Python 每个内置类型的序列 (str、tuple 和 list) 都支持以下操作符语法：

<code>s[j]</code>	索引下标为 j 的元素
<code>s[start:stop]</code>	切片操作得到索引为 [start, stop) 的序列
<code>s[start:stop:step]</code>	切片操作，新的序列包含索引为 start, start + step, start + 2 * step, …, 直到序列结束
<code>s + t</code>	序列的连接
<code>k * s</code>	序列 s 连接即 $s + s + s + \dots$ (k 次)
<code>val in s</code>	检查元素 val 在序列 s 中
<code>val not in s</code>	检查元素 val 不在序列 s 中

Python 使用序列的零索引，因此一个长度为 n 的序列的元素的索引是从 0 到 $n - 1$ 。Python 还支持使用负索引，表示离序列尾部的距离；索引 -1 表示序列的最后一个元素，索引 -2 表示序列的倒数第二个元素，以此类推。Python 使用切片标记法来描述一个序列的子序列。切片被描述为一种半开放的状态，即开始索引的元素包含在内，结束索引的元素排除在外。例如，语句 `data[3:8]` 产生一个子序列，子序列包含 5 个索引值：3, 4, 5, 6, 7。一个可选的“step”值，有可能是负数，可以当作切片的第三个参数。如果在切片表达式中省略了一个起始索引或结束索引，则假设起始或结束对应的是原始序列的头或尾。

因为列表是可变的，语法 `s[j] = val` 可以替换给定索引的元素。列表还支持语法 `del s[j]`，即从列表中删除指定的元素。切片标记法也可以用来取代或删除子列表。

表达式 `val in s` 可以用在任何序列中检验其中是否有元素与 val 的值相等。对字符串来说，这个语法可以用来匹配其中的一个字符或一个较大的子串，如 `'amp' in 'example'`。

所有序列规定的比较操作都是基于字典顺序，即一个元素接一个元素地比较，直至找到第一个不同的元素。例如，`[5, 6, 9] < [5, 7]`，因为第一个序列中索引为 1 的元素小。因此，下面的操作由序列类型支持：

<code>s == t</code>	相等 (每一个元素对应相等)
<code>s != t</code>	不相等
<code>s < t</code>	字典序地小于
<code>s <= t</code>	字典序地小于或等于
<code>s > t</code>	字典序地大于
<code>s >= t</code>	字典序地大于或等于

集合和字典的运算符

`set` 和 `frozenset` 支持以下操作：

<code>key in s</code>	检查 <code>key</code> 是 s 的成员
<code>key not in s</code>	检查 <code>key</code> 不是 s 的成员
<code>s1 == s2</code>	$s1$ 等价 $s2$
<code>s1 != s2</code>	$s1$ 不等价 $s2$
<code>s1 <= s2</code>	$s1$ 是 $s2$ 的子集
<code>s1 < s2</code>	$s1$ 是 $s2$ 的真子集
<code>s1 >= s2</code>	$s1$ 是 $s2$ 的超集
<code>s1 > s2</code>	$s1$ 是 $s2$ 的真超集 ($s1$ 不等于 $s2$)

$s1 s2$	$s1$ 与 $s2$ 的并集
$s1 \& s2$	$s1$ 与 $s2$ 的交集
$s1 - s2$	$s1$ 与 $s2$ 的差集
$s1 ^ s2$	对称差分 (该集合中的元素在 $s1$ 与 $s2$ 的其中之一)

需要注意的是，集合并不保证它们内部元素以特定的顺序排列，所以比较运算符（如 $<$ ）不是以字典顺序进行比较的；相反，它们是基于子集的数学概念的。所以，比较运算符定义一个部分的顺序，但不是一个总体的顺序，因为不相交的集合彼此不是“小于”“等于”或“大于”的关系。集合通过命名方法（例如添加、删除）支持许多基本的行为，我们将在第 10 章更充分地探讨其功能。

字典像集合一样，它们的元素没有一个明确定义的顺序。此外，对于字典，子集的概念并没有太大的意义，所以 dict 类并不支持形如 $<$ 的运算符。字典支持等价的概念，如果两个字典包含相同的键-值对集合，那么 $d1 == d2$ 。字典最广泛使用的操作是访问一个值，这个值与特定的索引语法为 $d[k]$ 的键 k 相关联。支持的操作如下：

$d[key]$	给定键 key 所关联的值
$d[key] == value$	设置（或重置）与给定的键相关联的值
$del d[key]$	从字典中删除键及其关联的值
$key in d$	检查 key 是 d 的成员
$key not in d$	检查 key 不是 d 的成员
$d1 == d2$	$d1$ 等价于 $d2$
$d1 != d2$	$d1$ 不等价于 $d2$

字典通过命名方法支持许多有用的行为，我们将在第 10 章更充分地探讨其功能。

扩展赋值运算符

Python 支持对大多数二元运算符进行扩展赋值运算，例如，允许形如 $count += 5$ 的语法表达式。默认情况下，这是更繁琐的表达式 $count = count + 5$ 的一种简约表述。对于不可变类型，如数字或字符串，不应该认为该语法改变现有对象的值，而是它将对新构造的值重新分配标识符（见图 1-3）。然而，对于一种类型，它可通过重新定义语法规则去改变对象的行为，如对列表类进行 $+=$ 操作。

```
alpha = [1, 2, 3]
beta = alpha          # an alias for alpha
beta += [4, 5]         # extends the original list with two more elements
beta = beta + [6, 7]    # reassigns beta to a new list [1, 2, 3, 4, 5, 6, 7]
print(alpha)           # will be [1, 2, 3, 4, 5]
```

这个例子展现了语句 $beta += foo$ 与 $beta = beta + foo$ 在列表语义方面的微妙差异。

复合表达式和运算符优先级

编程语言对复合表达式的执行顺序必须有明确的规则，如计算 $5 + 2 * 3$ 。Python 中运算符正式的优先级顺序在表 1-3 中给出。在同一个级别中，优先级高的运算符将会比优先级低的运算符先执行，除非表达式中有括号。因此，我们看到 Python 中乘法的优先级高于加法，因此表达式 $5 + 2 * 3$ 是作为 $5 + (2 * 3)$ 计算的，值为 11，但是加了括号后的表达式 $(5 + 2) * 3$ 计算的值为 21。同一个级别的运算符是从左到右计算的，因此 $5 - 2 + 3$ 的值为 6。此规则的例外情况是一元运算符和求幂运算是从右至左运算的。

表 1-3 Python 运算符的优先级，同类别中从最高级别到最低级别排序。我们使用 expr 来表示文字、标识符，或表达式的运算结果。所有没有明确提及的 expr 的运算符都是二元运算符，其语法形式如 expr1 operator expr2

运算符优先级		
	类 型	符 号
1	成员访问	expr.member
2	函数 / 方法调用	expr(...)
	容器下标 / 切片	expr[...]
3	幂运算	**
4	一元运算符	+ expr, - expr, ~ expr
5	乘法, 除法	*, /, //, %
6	加法, 减法	+, -
7	按位移位	<<, >>
8	按位与	&
9	按位异或	^
10	按位或	
11	比较 包含	is, is not, ==, !=, <, <=, >, >=, in, not in
12	逻辑非	not expr
13	逻辑与	and
14	逻辑或	or
15	条件判断	val1 if cond else val2
16	赋值	=, +=, -=, *= 等

Python 支持多级赋值，如 `x = y = 0`，将最右边的值赋值给指定的多个标识符。Python 还支持链接比较运算符。例如，表达式 `1 <= x + y <= 10` 等价于复合表达式 `(1 <= x + y) and (x + y <= 10)`，这样可以不用将中间值 `x + y` 计算两次。

1.4 控制流程

在本节中，我们将回顾 Python 中最基本的控制结构：条件语句和循环语句。在 Python 中，控制结构中常见的是使用语法来定义代码块。冒号字符用于标识代码块的开始，代码块作为控制结构的结构体。如果结构体可以被表述为一个可执行语句，则它可以与冒号置于同一行上，且在冒号的右边。然而，结构体通常从冒号的下一行起整齐缩进。Python 依赖于缩进级别或嵌套结构来指定代码块。同样的原则适用于指定一个函数体（见 1.5 节）和一个类的主体（见 2.3 节）。

1.4.1 条件语句

条件结构（也称为 if 语句）提供了一种方法，用以执行基于一个或多个布尔表达式的运行结果而选择的代码块。在 Python 中，条件语句一般的形式如下：

```
if first_condition:  
    first_body  
elif second_condition:  
    second_body  
elif third_condition:  
    third_body  
else:  
    fourth_body
```

每个条件都是布尔表达式，并且每个主体包含一个或多个在满足条件时才执行的命令。如果满足第一个条件，那么将执行第一个结构体，而其他条件或结构体不会执行。如果不满足第一个条件，那么这个流程以相似的方式评估第二个条件，并将继续下去。整体构造的执行将决定必有一个结构体会被执行。这里可能有任意数量的 `elif` 语句（包括零个），最后一条 `else` 语句是可选的。就像前面提到的，非布尔类型可以被评估为具有直观含义的布尔值。例如，如果 `response` 是由用户输入的一个字符串，我们会以这是一个非空字符串为条件，写为

```
if response:
```

可以看作下列等价表达式的简写：

```
if response != '':
```

作为一个简单的例子，一个机器人控制器可能有以下逻辑：

```
if door_is_closed:  
    open_door()  
    advance()
```

注意：最后的命令 `advance()` 没有缩进，因此不是条件结构体的一部分。它将会被无条件地执行（尽管它在打开一个关着的门之后）。

我们可以在一个控制结构中嵌套另一个控制结构，基于缩进可以明确不同结构体的范围。重新审视机器人的例子，这里有一个更复杂的控制，是在紧闭的门上增加开锁的条件。

```
if door_is_closed:  
    if door_is_locked:  
        unlock_door()  
    open_door()  
    advance()
```

这个例子表示的逻辑可以描绘为一种传统的流程图，如图 1-6 所示。

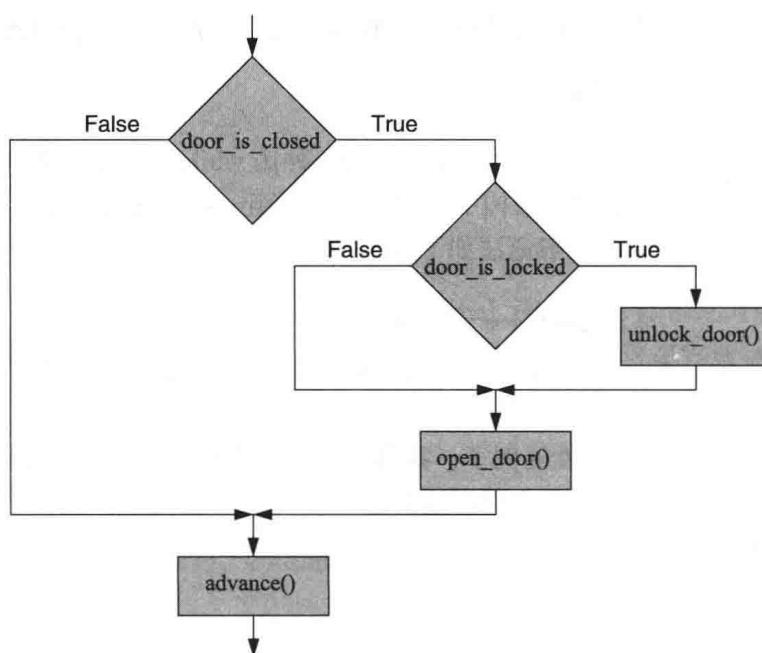


图 1-6 描述逻辑嵌套条件语句的流程图

1.4.2 循环语句

Python 提供了两种不同的循环结构。`while` 循环允许以布尔条件的重复测试为基础的一般重复。`for` 循环对定义序列的值提供了适当的迭代（如字符串中的字符、列表中的元素或一定范围内的数字）。

`while` 循环

Python 中的 `while` 循环的语法如下：

```
while condition:  
    body
```

就一个 `if` 语句来说，`condition` 可以是任意布尔表达式，结构体可以是任意代码块（包括嵌套控制结构）。执行 `while` 循环时首先测试布尔条件。如果条件的结果为 `True`，执行循环的主体。每次执行结构体后，重新测试循环条件，如果测试条件的结果为 `True`，那么开始执行结构体的另一轮迭代。如果测试条件的结果为 `False`（假设曾经出现过），那么循环退出，并且控制流在循环的主体之外继续。

作为示例，这里给出一个循环，通过字符序列的索引，找到一个输入值为 '`X`' 的值或直接到达序列的尾部。

```
j = 0  
while j < len(data) and data[j] != 'X':  
    j += 1
```

我们将在 1.5.2 节讨论 `len` 函数，它返回一个序列（如列表或字符串）的长度。这个循环的正确性依赖于 `and` 运算符的短路效应。在访问元素 `data[j]` 之前，首先测试 `j < len(data)` 以确保 `j` 是一个有效的索引。如果我们以相反的顺序写成复合条件，当 '`X`' 不存在时，`data[j]` 的结果将最终会抛出 `IndexError` 异常（见 1.7 节讨论的异常情况）。

如上所述，当这个循环结束时，如果 '`X`' 存在，变量 `j` 的值是出现在最左边的 '`X`' 的索引，否则就是序列的长度（这将会被作为一个预示着搜索失败的无效索引）。值得注意的是这段代码本身的正确性，甚至在特殊情况下，如当列表为空时，条件 `j < len(data)` 一开始即执行失败，而循环体永远不会被执行。

`for` 循环

在迭代一系列的元素时，Python 的 `for` 循环是一种比 `while` 循环更为便利的选择。`for` 循环的语法可以用在任何类型的迭代结构中，如列表、元组、`str`、集合、字典或文件（我们将在 1.8 节正式讨论迭代器）。其一般语法如下：

```
for element in iterable:  
    body          # body may refer to 'element' as an identifier
```

对于熟悉 Java 的读者，Python 的 `for` 循环的语法与 Java 1.5 中介绍的 “`for each`” 循环的风格很相似。

作为 `for` 循环一个很有启发性的例子，我们考虑的是计算列表中元素数值的总和（当然，Python 中有一个内置的函数 `sum`，也可以达到这一目的）。我们在 `for` 循环中执行如下计算，假设 `data` 代表列表：

```
total = 0  
for val in data:  
    total += val          # note use of the loop variable, val
```

标识符 `val` 从 `for` 循环指定的元素开始遍历，对于 `data` 序列中的每一个元素，循环体都

会执行一次。值得注意的是，`val` 被视为一个标准标识符。如果原始 `data` 中的元素是可变的，可以使用 `val` 标识符调用它的方法。但是给标识符 `val` 重新赋一个新的值并不影响原始 `data`，也不影响下一次的迭代循环。

第二个经典的例子，我们考虑在一个列表的元素中寻找最大值（Python 的内置函数 `max` 已经提供了这种功能）。我们可以假设 `data` 列表至少有一个元素，那么可以实现这个任务：

```
biggest = data[0]                      # as we assume nonempty list
for val in data:
    if val > biggest:
        biggest = val
```

虽然我们可以用 `while` 循环来完成上述任务，但 `for` 循环的优点是简洁，即不需要管理列表的明确索引及构造布尔循环条件。此外，我们可以在 `while` 循环不适用的情况下使用 `for` 循环，例如遍历一个集合 `set`，但它不支持任何直接形式的索引。

基于索引的 `for` 循环

标准的 `for` 循环用于遍历一个列表的元素时是很简洁的，但这种形式的一个限制是我们不知道元素在这个序列的哪一个位置。在某些应用程序中，我们需要知道序列中元素的索引。例如，假设我们想知道列表中最大元素所在的位置。

在这种情况下，我们宁愿遍历列表中所有可能的索引，而不是直接在列表的元素上循环。为此，Python 提供了一个名为 `range` 的内置类，它可以生成整数序列。（我们将在 1.8 节讨论生成器。）在最简单的形式中，语法 `range(n)` 生成具有 `n` 个值的序列，下标从 0 到 `n - 1`。很明显，这一系列有效的索引构成的序列的长度为 `n`。因此，标准的 Python 语言对数据序列的一系列索引应用 `for` 循环时，使用以下语法：

```
for j in range(len(data)):
```

在这种情况下，标识符 `j` 并不是 `data` 中的元素，它是一个整数。而表达式 `data[j]` 可以用来检索序列中相应的元素。例如，我们可以找到列表中最大元素的索引，如下：

```
big_index = 0
for j in range(len(data)):
    if data[j] > data[big_index]:
        big_index = j
```

`break` 和 `continue` 语句

Python 支持 `break` 语句，当在循环体内执行 `break` 语句时，`while` 或 `for` 循环就会立即终止。更正式地说，如果在嵌套的控制结构中使用 `break` 语句，它会导致内层循环立刻终止。一个典型的例子如下面的代码所示，它是确定一个目标值是否出现在数据集中：

```
found = False
for item in data:
    if item == target:
        found = True
        break
```

Python 也支持 `continue` 语句，`continue` 语句会使得循环体的当前迭代停止，但循环过程的后续迭代会正常进行。

我们建议慎用 `break` 和 `continue` 语句。然而，在有些情况下，可以有效地使用这些命令，以免引入过于复杂的逻辑条件。

1.5 函数

在这一节中，我们探讨 Python 中函数的创建和使用。正如我们在 1.2.2 节讨论的，应明确函数和方法之间的区别。我们用一般的术语——函数来描述一个传统的、无状态的函数，该函数被调用而不需要了解特定类的内容或该类的实例，例如 `sorted(data)`。我们使用更具体的术语——方法来描述一个成员函数，在调用特定对象时使用面向对象的消息传递语法，如 `data.sort()`。在这一节中，我们只考虑纯函数；在第 2 章中，我们使用更广泛的面向对象原则来探讨方法。

我们从一个例子开始说明在 Python 中定义函数的语法。在任何形式的可迭代数据集中，下面的函数计算给定目标值出现的次数。

```
def count(data, target):
    n = 0
    for item in data:
        if item == target:           # found a match
            n += 1
    return n
```

以关键字 `def` 开始的第一行作为函数的签名。这个标志建立了一个新的标识符作为函数的名称（在这个示例中是 `count`），并且设立了期望的参数个数，以及标识这些参数的名称（在这个示例中是 `data` 和 `target`）。与 Java 和 C++ 不同，Python 是一种动态类型语言，因此 Python 签名不指定这些参数的类型，也不指定返回值的类型（如果有的话）。这些参数的使用在函数的说明文档中描述（见 2.2.3 节），并且在函数体中执行，但是对于函数的错误使用只有在运行时才被检测到。

函数定义的其余部分称为函数的主体。和 Python 中控制结构的情况一样，函数体通常以缩进的代码块的形式表示。每次调用函数时，Python 会创建一个专用的活动记录用来存储与当前调用相关的信息。这个活动记录包括了命名空间（见 1.10 节）。命名空间用以管理当前调用中局部作用域内的所有标识符。命名空间包含该函数的参数以及在函数体内定义的其他本地标识符。函数调用者局部作用域内的标识符与调用者作用域内的其他相同名称的标识符没有关系（虽然在不同的作用域的标识符可能是同一对象的别名）。在第一个例子中，标识符 `n` 的范围是局部函数调用。作为标识符项，它被作为循环变量使用。

return 语句

`return` 语句一般用在函数体内，用来表示该函数应立即停止执行，并将所得到的值返回给调用者。如果 `return` 语句在执行之后没有明确的返回值，则 `None` 值会自动返回给调用者。同样，如果控制流在没有执行 `return` 语句的情况下到达过函数体的末端，那么 `None` 值会被返回。通常，`return` 语句会是函数体的最后一条命令，如前面所示 `count` 函数例子中。然而，如果命令执行受条件逻辑控制，那么在同一函数中可以有多个 `return` 语句。作为一个深入的例子，下面考虑这样一个函数——测试序列中是否有一个这样的值。

```
def contains(data, target):
    for item in data:
        if item == target:           # found a match
            return True
    return False
```

如果满足循环体内的条件，那么 `return True` 语句就会执行，然后函数就会立即结束。`True` 表示目标值已经被找到。相反，如果 `for` 循环到达结尾仍然没有找到匹配值，那么最后

的 return False 语句将被执行。

1.5.1 信息传递

要成为一个优秀的程序员，你必须对编程语言如何从函数中传递信息的机制有一个清晰的理解。在函数签名的上下文中，用来描述预期参数的标识符被称为形式参数，调用者调用函数时发送的对象是实际参数。在 Python 中，参数传递遵循标准赋值语句的语法。当调用一个函数时，在函数的局部范围内，每个标识符将作为一个形式参数被赋值给该函数的调用方提供的相应的实际参数。

例如，考虑以下来自前面 count 函数的调用：

```
prizes = count(grades, 'A')
```

在执行函数体之前，实际参数 grades 和 'A' 已经被隐式分配给了形式参数 data 和 target。代码如下：

```
data = grades
target = 'A'
```

这些赋值语句将标识符 data 作为 grades 的别名，并将 target 作为字符串 'A' 的别名，如图 1-7 所示。

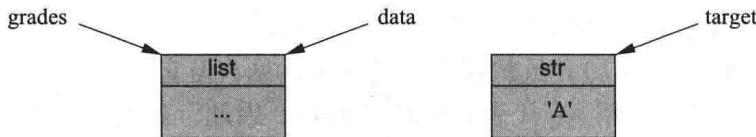


图 1-7 Python 中对于函数调用 count(grades, 'A') 参数传递的描述。标识符 data 和 target 是 count 函数定义的局部范围内的形式参数

函数的返回值传递给调用者这一实现类似于赋值。因此，我们的示例调用 prizes = count(grades, 'A')，在调用者作用域内的标识符 prizes 赋值给了对象，此对象就是函数体中返回语句确定的 n。

对于从一个函数中传递信息来说，Python 机制的优点是不用复制对象。即使在一个参数或返回值是一个复杂对象的情况下，这也确保了函数的调用是有效的。

可变参数

当一个参数是可变对象时，Python 的参数传递模式有其他作用。因为形参是实际参数的一个别名，函数体与对象的交互或许会改变它的状态。再一次考虑对于示例 count 函数的调用，如果函数体执行 data.append('F') 这条命令，新的条目被添加到函数中 data 列表的末尾，这改变了被调用者所知的相同的列表，比如 grades。另外，我们注意到在函数体内给形式参数重新赋予新值，形如设置 data = []，并不改变实际参数——这种重新赋值的方式只是改变了别名。

我们假设的 count 方法的例子是给列表追加新的元素，这是缺乏常识的。没有理由期待这样的行为，对参数有这样一个意想不到的影响将是相当糟糕的设计。然而，在许多合法的情况下，一个函数可以被设计（和清楚地记录）用以修改参数的状态。作为一个具体的例子，我们提出实现一个名为 scale 的方法，主要的目的是给数据集中的所有数都乘上一个给定的因子。

```
def scale(data, factor):
    for j in range(len(data)):
        data[j] *= factor
```

默认参数值

Python 提供了支持多个可能的调用函数签名的方法。这样的函数被视为多态的（在希腊语是“许多形式”的意思）。最值得注意的是，函数可以为参数声明一个或多个默认值，从而允许调用方用不同个数的实际参数调用函数。例如，如果一个函数用下列签名来声明

```
def foo(a, b=15, c=27):
```

这里有三个参数，其中最后两个提供默认值。调用方可以提供三个实际参数，如 `foo(4, 12, 8)`。在这种情况下，默认值是没有用的。换言之，如果调用方只能提供一个参数 `foo(4)`，该函数将以参数值 $a = 4$ 、 $b = 15$ 、 $c = 27$ 执行。如果调用方提供两个参数，那么这两个参数被假定赋给形式参数的前两位，形式参数的第三位还是取默认值。因此，`foo(8, 20)` 将以参数值 $a = 8$ 、 $b = 20$ 、 $c = 27$ 执行。然而，形如 `bar(a, b = 15, c)` 的签名，其中 b 具有默认值而后续的 c 没有默认值，使用这样的签名定义函数是不合法的。如果一个默认的参数具有参数值，那么它后面的参数也必须具有默认值。

作为一个使用默认参数的更加深入的例子，我们重新计算一个学生平均绩点（GPA）的任务（见代码段 1-1）。不是假设与控制台进行直接的输入和输出，我们希望设计一个函数，这个函数用于计算并返回一个 GPA。最初的实现是使用一个固定的映射，每个字母等级（如 B-）对应相应的值（如 2.67）。虽然这在系统中很常见，但它并不是所有学校使用的系统（例如，有些学校可能会使用 A+ 代表分值高于 4.0）。因此，我们设计了一个 `compute_gpa` 函数，如代码段 1-2 所示，它允许调用者指定自定义的等级到值的映射，同时提供标准的系统默认值。

代码段 1-2 一个计算学生平均绩点的函数，这个函数的特点是可以定制可选的参数

```
def compute_gpa(grades, points={'A+':4.0, 'A':4.0, 'A-':3.67, 'B+':3.33,
                                'B':3.0, 'B-':2.67, 'C+':2.33, 'C':2.0,
                                'C-':1.67, 'D+':1.33, 'D':1.0, 'F':0.0}):
    num_courses = 0
    total_points = 0
    for g in grades:
        if g in points:                      # a recognizable grade
            num_courses += 1
            total_points += points[g]
    return total_points / num_courses
```

作为有趣的多态函数的另外一个示例，我们考虑 Python 对 `range` 的支持。（从技术上讲，这是一个 `range` 类的构造函数，但是为了讨论这个问题，我们可以把它当作一个纯函数来对待。）Python 对于 `range` 支持三种调用语法：单参数的形式，如 `range(n)`，产生一个从 0 到 n 但不包含 n 的整数序列；两个参数的形式，如 `range(start, stop)`，生成从 `start` 开始到 `stop` 结束但不包含 `stop` 的整数序列；三个参数的形式，如 `range(start, stop, step)`，生成一个类似于 `range(start, stop)` 的序列，但序列增量的大小是 `step` 而不是 1。

这种形式的组合似乎违反了默认参数的规则。特别是当只有单参数时，如 `range(n)`，它作为一个 `stop` 值（这是第二个参数）。在这种情况下，`start` 的有效值是 0。然而，这种效果可以用一些手法来实现，代码如下：

```
def range(start, stop=None, step=1):
    if stop is None:
        stop = start
        start = 0
    ...

```

从技术角度来看，当 `range(n)` 被调用时，实际参数 `n` 将被赋值给形式参数 `start`。在函数体内，如果只接收到一个参数，`start` 和 `stop` 的值将会重新被赋值以提供所需的语义。

关键字参数

把调用者的实际参数匹配给由函数签名声明的形式参数，传统机制是基于位置参数的概念。例如，签名 `foo(a = 10, b = 20, c = 30)`，调用者按照给定的顺序把实际参数匹配给形式参数。`foo(5)` 的调用表示 `a = 5`，而 `b` 和 `c` 的值是指定的默认值。

Python 支持另一种将关键字参数传递给函数的机制。关键字参数是通过显式地按照名称将实际参数赋值给形式参数来指定的。例如，使用上述定义的 `foo` 函数，调用 `foo(c = 5)` 将以参数 `a = 10`、`b = 20`、`c = 5` 的形式执行。

一个函数的作者可以获取某些只能通过关键字参数语法传递的参数。我们在自己的函数定义中从来没有这样的限制，但在 Python 标准库中会看到唯一关键字参数的几个重要的用法。例如，内置的 `max` 函数接收一个名为 `key` 的关键字参数，可以用来改变使用的“最大”的概念。

默认情况下，`max` 运算符是根据 `<` 操作符对元素的自然顺序进行操作的。但是最大的数可以通过比较元素的其他方面得出。这可以通过提供一个辅助函数实现——为了比较将自然元素转换为其他值来完成。例如，如果有兴趣寻找数值最大的一个数（即考虑 `-35` 要大于 `20`），我们可以调用语法 `max(a, b, key = abs)`。在这种情况下，内置的 `abs` 函数本身就是传递与关键字参数 `key` 相关联的值（在 Python 中函数是第一类对象，参见 1.10 节）。在这种方式下调用 `max` 函数，它会比较 `abs(a)` 和 `abs(b)`，而不是 `a` 和 `b`。在 `max` 函数的情景中，关键字语法作为位置参数的替代的是很重要的。这个函数在参数的个数方面是多态的，允许形如 `max(a, b, c, d)` 的调用，因此，它不可能指定一个关键函数作为传统的位置元素。在 Python 中，排序函数为了表示非标准的序列也支持类似的 `key` 参数（当讨论排序算法时，我们在 9.4 节和 12.6 节对此做进一步探讨）。

1.5.2 Python 的内置函数

表 1-4 列出了 Python 中自动可用的常见函数，包括前面讨论的 `abs`、`max` 和 `range`。当选择参数的名称时，我们使用标识符 `x`、`y`、`z` 表示任意数值类型，`k` 代表整数，`a`、`b` 和 `c` 表示任意可比较类型。我们使用标识符 `iterable` 代表任何可迭代类型的一个实例（如 `str`、`list`、`tuple`、`set`、`dict`）。我们将在 1.8 节讨论迭代器和可迭代数据类型。序列代表可索引类的一个更窄的范畴，包含 `str`、列表和元组，但不包含集合和字典。表 1-4 根据功能将函数分为如下几类：

- 输入 / 输出：`print`、`input` 和 `open` 函数，细节参见 1.6 节的内容。
- 字符编码：`ord` 和 `chr` 将字符和其对应的整型编码关联起来。如 `ord('A')` 的值是 65，`chr(65)` 的值是 'A'。
- 数学：`abs`、`divmod`、`pow`、`round` 和 `sum` 提供了通用的数学功能。1.11 节介绍了一个额外的数学模块。
- 排序：`max` 和 `min` 适用于支持比较概念的任何数据类型或这些值的任何集合。同样，

从已存在的集合中抽取元素，可以使用 sorted 生成排序的列表。

- 集合 / 迭代：range 产生一个新的数字数列；len 得到任何现有集合的长度；函数 reversed、all、any 和 map 操作任意的迭代类型；iter 和 next 通过集合中的元素对迭代提供一个总体框架，参见 1.8 节相关内容。

表 1-4 常见的内置函数

调用语法	描述
abs(x)	返回数字的绝对值
all(iterable)	对于每一个元素 e，如果 bool(e) 为 True，那么返回 True
any(iterable)	至少存在一个元素 e，使 bool(e) 为 True，那么返回 True
chr(integer)	返回给定 Unicode 编码的字符
divmod(x, y)	如果 x 和 y 都是整数，返回元组 (x//y, x%y)
hash(obj)	对于对象 obj 返回一个整数的散列值（见第 10 章）
id(obj)	返回作为对象身份标识的唯一整数
input(prompt)	返回标准输入的字符串，prompt 是可选的
isinstance(obj, cls)	确定对象是类的一个实例（或子类）
iter(iterable)	为参数返回一个新的迭代对象（见 1.8 节）
len(iterable)	返回给定迭代对象的元素个数
map(f, iter1, iter2, ...)	返回迭代器产生的函数调用 f(e1, e2, ...) 的结果，其中元素 e1 ∈ iter1, e2 ∈ iter2, ...
max(iterable)	返回给定迭代对象中最大的元素
max(a, b, c, ...)	返回给定参数中最大的元素
min(iterable)	返回给定迭代对象中最小的元素
min(a, b, c, ...)	返回给定参数中最小的元素
next(iterator)	通过迭代器返回下一个元素（见 1.8 节）
open(filename, mode)	通过给定的名字和存取模式打开文件
ord(char)	返回给定字符的 Unicode 编码值
pow(x, y)	返回 x^y 的值（当 x 和 y 为整型时值为整型）；等价于 $x**y$
pow(x, y, z)	返回整型值 ($x^y \bmod z$)
print(obj1, obj2, ...)	打印参数，参数之间以空格分隔，打印完毕后换行
range(stop)	构造关于值 0, 1, ..., stop - 1 的迭代
range(start, stop)	构造关于值 start, start + 1, ..., stop - 1 的迭代
range(start, stop, step)	构造关于值 start, start + step, start + 2*step, ... 的迭代
reversed(sequence)	返回逆置序列的迭代
round(x)	返回最接近的 int 型值（省去小数点后的数向偶数值靠近）
round(x, k)	返回最接近 10^{-k} 的近似值（返回类型匹配 x）
sorted(iterable)	返回一个列表，它包含的元素是以顺序排序的 iterable 中的元素
sum(iterable)	返回 iterable 中元素的和（必须是数字）
type(obj)	返回实例 obj 所属的类

1.6 简单的输入和输出

在本节中，我们会谈到 Python 语言中输入和输出的基本知识，并描述通过用户控制台来实现标准输入和输出，以及对读写文本文件的支持。

1.6.1 控制台输入和输出

print 函数

`print` 函数（Python 语言中的内置函数）用来生成标准输出到控制台。在其最简单的形式中，它可以打印任意序列的参数。多个参数之间以空格作为分隔，末尾有一个换行符。例如，命令 `print('maroon', 5)` 就是输出字符串 'maroon 5\n'。注意：这些参数也可以不是字符串实例，一个非字符串参数 `x` 也将以 `str(x)` 的形式显示。要是没有任何参数，命令 `print()` 输出的就是单个的换行符。

`print` 函数可以使用下列关键字参数进行自定义（参照 1.5 节对关键字参数的讨论）：

- 默认情况下，`print` 函数在输出时会在每对参数间插入空格作为分隔，其实可以通过关键字参数 `sep` 自定义想要的分隔符以分隔字符串。例如，用冒号分隔可以使用 `print(a, b, c, sep = ':')`。分隔字符串不需要一定用单个字符，它可以是一个长的字符串，当然，它也可以是一个空串，如 `sep = ''`，这样可使得这些参数直接相连。
- 默认情况下，在最后一个参数后会输出换行符。使用关键字参数 `end` 可以指定一个可选择的结尾字符串。指定空字符串 `end = ''`，这样结束后不输出任何字符。
- 默认情况下，`print` 函数会直接将输出发送到标准控制台。然而，通过使用关键字参数 `file` 指示一个输出文件流（参见 1.6.2 节），也可以直接输出到一个文件。

input 函数

`input` 是一个内置函数，它的主要功能是接收来自用户控制台的信息。如果给出一个可选参数，那么这个函数会显示提示信息，然后等待用户输入任意字符，直到按下返回键。这个函数的返回值是按下返回键之前用户所输入的字符串（即换行符不存在于返回值中）。

当读到来自用户的数值时，程序员必须使用 `input` 函数获取字符串，然后使用 `int` 或 `float` 语法来构建用字符串表示的这些数值，即如果 `response = input()`，用户输入字符串 '2013'，那么 `int(response)` 可以得到整型值 2013。将这些操作和语法结合起来是很常见的，例如，

```
year = int(input('In what year were you born? '))
```

假定用户会输入一个合适的响应（在 1.7 节中，我们会讨论这种情况下的错误处理）。

因为 `input` 函数会返回一个字符串作为结果，如附录 A 中所述，该函数的使用可以与 `string` 类的现有功能相结合。例如，如果用户在同一行上输入多个信息，则通常会对结果调用 `split` 方法，即

```
reply = input('Enter x and y, separated by spaces: ')
pieces = reply.split() # returns a list of strings, as separated by spaces
x = float(pieces[0])
y = float(pieces[1])
```

示例程序

下面有一个简单但完整的程序，展示了 `input` 和 `print` 函数的使用规范。格式化最终输出结果的工具会在附录 A 中讨论。

```
age = int(input('Enter your age in years: '))
max_heart_rate = 206.9 - (0.67 * age) # as per Med Sci Sports Exerc.
target = 0.65 * max_heart_rate
print('Your target fat-burning heart rate is', target)
```

1.6.2 文件

在 Python 中访问文件要先调用一个内置函数 `open`，它返回一个与底层文件交互的对

象。例如，命令 `fp = open('sample.txt')` 用于打开名为 `sample.txt` 的文件，返回一个对该文本文件允许只读操作的文件对象。

`open` 函数的第二个可选参数是确认对文件的访问权限，默认权限 '`r`' 是只读。其他常见权限如 '`w`' 是对文件进行写操作（会覆盖当前文件之前的内容），'`a`' 是对当前文件的尾部追加内容。尽管我们对文本文件的使用比较关注，但使用 '`rb`' 或者 '`wb`' 也可以对二进制文件进行访问。

在处理一个文件时，文件对象使用距离文件开始处的偏移量（以字节为单位）维护文件中的当前位置。当以只读权限 '`r`' 或只写权限 '`w`' 打开文件时，初始位置是 `0`；如果是以追加权限 '`a`' 打开，初始位置是在文件的末尾。`fp.close()` 会关闭与文件对象 `fp` 相关的文件，确保写入的内容已被保存。读写文件的常用方法见表 1-5。

表 1-5 文件对象 `fp` 与文件交互的常用方法

调用方法	描述
<code>fp.read()</code>	将可读文件剩下的所有内容作为一个字符串返回
<code>fp.read(k)</code>	将可读文件中接下来的 <code>k</code> 个字节作为一个字符串返回
<code>fp.readline()</code>	从文件中读取一行内容，并以此作为一个字符串返回
<code>fp.readlines()</code>	将文件中的每行内容作为一个字符串存入列表中，并返回该列表
<code>for line in fp</code>	遍历文件的每一行
<code>fp.seek(k)</code>	将当前位置定位到文件的第 <code>k</code> 个字节
<code>fp.tell()</code>	返回当前位置偏离开始处的字节数
<code>fp.write(string)</code>	在可写文件的当前位置将 <code>string</code> 的内容写入
<code>fp.writelines(seq)</code>	在可写文件的当前位置写入给定序列的每个字符串。除了那些嵌入到字符串中的换行符，这个命令不插入换行符
<code>print(..., file=fp)</code>	将 <code>print</code> 函数的输出重定向给文件（输出文件内容）

读文件

通过文件对象读取文件最基本的命令是 `read` 方法。当使用 `fp.read(k)` 命令时，将返回从文件当前位置开始后继的 `k` 个字节。如果没有参数，即形如 `fp.read()`，则返回文件当前位置后的全部内容。为了方便，文件也可以一次读取一行，使用 `readline` 方法或者 `readlines` 方法将剩余的每一行以列表方式返回。文件也支持 `for-loop` 操作，即逐行遍历（例如 `for line in fp`）。

写文件

当文件对象是可写的，例如，以写权限 '`w`' 或追加权限 '`a`' 创建一个文件时，就可以使用 `write` 方法或 `writelines` 方法。例如，如果现在定义 `fp = open('results.txt', 'w')`，执行 `fp.write('Hello World.\n')` 就是将给定字符串在文件中单独写一行。注意：在写文件时，它不会自动在尾部追加换行符。如果需要换行符，则必须将其写入字符串中。回忆一下前面提到的 `print` 方法，可以使用关键字参数将内容输出到文件中。

1.7 异常处理

异常是程序执行期间发生的突发性事件。逻辑错误或未预料到的情况都有可能造成异常。在 Python 中，异常（也被称为错误）也是执行代码时遇到突发状况所引发（或抛出）的对象。当遇到突发状况如内存溢出时，Python 解释器也可以引发异常。如果在上下文中有处理异常的代码，那么异常可能会被捕获。如果没有捕获，异常可能会导致解释器停止运行。

程序，并且向控制台发送合适的信息。在这一节，我们会学习 Python 中最常见的错误类型、捕获异常和处理异常的机制以及用户定义的代码块内引发错误的语法。

常见错误类型

Python 含有大量的异常类，它们定义了各种不同类型的异常。表 1-6 给出了一些常见的异常类。`Exception` 类是所有异常类的基类。各子类的实例都编码成已发生问题的细节。本章所介绍的异常案例就会引发一些异常。例如，在表达式中使用未定义的标识符会造成 `NameError` 异常，还有 ‘.’ 符号的错误使用，如 `foo.bar()`，如果对象 `foo` 没有 `bar` 成员，则会引发 `AttributeError` 异常。

表 1-6 Python 中的常见异常类

异常类名	描述
<code>Exception</code>	所有异常类的基类
<code>AttributeError</code>	如果对象 <code>obj</code> 没有 <code>foo</code> 成员，会由语法 <code>obj.foo</code> 引发
<code>EOFError</code>	一个“end of file”到达控制台或者文件输入引发错误
<code>IOError</code>	输入 / 输出操作（如打开文件）失败引发错误
<code>IndexError</code>	索引超出序列范围引发错误
<code>KeyError</code>	请求一个不存在的集合或字典关键字引发错误
<code>KeyboardInterrupt</code>	用户按 <code>ctrl - C</code> 中断程序引发错误
<code>NameError</code>	使用不存在的标识符引发错误
<code>StopIteration</code>	下一次遍历的元素不存在时引发错误，参照 1.8 节
<code>TypeError</code>	发送给函数的参数类型不正确引发错误
<code>ValueError</code>	函数参数值非法时引发错误（例如， <code>sqrt(-5)</code> ）
<code>ZeroDivisionError</code>	除数为 0 引发错误

向函数发送一个错误的数字、类型或参数值是引发异常的另一个常见起因。例如，调用 `abs('hello')` 就会引发 `TypeError` 异常，因为参数不是数字型的；调用 `abs(3, 5)` 也会引发 `TypeError` 异常，因为只允许一个参数。如果传递参数的类型和数目都是正确的，但对于函数来说参数值是非法的，那就会引发 `ValueError` 异常。例如，`int` 型构造函数可接收字符串，如 `int('137')`，但如果字符串代表的不是整数，如 `int('3.14')` 或 `int('hello')`，就会引发 `ValueError` 异常。

当 `data[k]` 中的 `k` 对于所给序列是一个非法的索引时，Python 的序列类型（如列表、元组和 `str` 类）会引发 `IndexError` 异常。当试图访问一个不存在的元素时，集合和字典会引发 `KeyError` 异常。

1.7.1 抛出异常

当执行到带有异常类的实例（其中以指定问题作为参数）的 `raise` 语句时，就会抛出异常。例如，计算平方根的函数传递了一个负数作为参数，就会引发有如下命令的异常：

```
raise ValueError('x cannot be negative')
```

随着这个错误信息作为构造函数的一个参数，该语法会生成一个新创建的 `ValueError` 类实例。如果这个异常在函数体内没有被捕获，函数的执行会立刻停止，并且这个异常可能会被传播到调用的上下文（甚至更远）。

检查一个函数参数的有效性，首先要验证参数类型是否正确，然后再验证参数的值的正

确性。例如，在 Python 的 math 库中 sqrt 函数有错误检测，代码如下：

```
def sqrt(x):
    if not isinstance(x, (int, float)):
        raise TypeError('x must be numeric')
    elif x < 0:
        raise ValueError('x cannot be negative')
    # do the real work here...
```

检测一个对象的类型可以在运行时使用内置函数 `isinstance` 来实现。在最简单的形式中，如果对象 `obj` 是 `cls` 类的一个实例或者是该类型的任何子类，`isinstance(obj, cls)` 会返回 `True`。在上述例子中，更常见的形式是使用以第二个参数表示的正当类型的元组。在确认该参数是数字后，函数会产生一个数字是非负的异常，否则会抛出一个 `ValueError` 异常。

要对函数执行多少次错误检测是一个有争议的问题。检查参数的类型和数值需要额外的执行时间，如果走向极端，似乎与 Python 的本质不符。例如，内置函数 `sum()` 用于计算一系列数字的总和，其错误检测的实现如下：

```
def sum(values):
    if not isinstance(values, collections.Iterable):
        raise TypeError('parameter must be an iterable type')
    total = 0
    for v in values:
        if not isinstance(v, (int, float)):
            raise TypeError('elements must be numeric')
        total = total + v
    return total
```

抽象基类 `collections.Iterable` 包括所有确保支持 `for` 循环语法的 Python 迭代容器类型（如，`list`、`tuple`、`set`）。我们在 1.8 节讨论迭代，并且在 1.11 节讨论模块（如 `collections`）的使用。在 `for` 循环的主体内部，在将每个元素加到整体之前，要确认它是数字。该函数更直接、更清晰的实现如下：

```
def sum(values):
    total = 0
    for v in values:
        total = total + v
    return total
```

有趣的是，这个简单实现完全像 Python 函数的内置版本。即使没有显式检查，适当的异常也会由代码自然抛出。特别是，如果 `values` 不是一个迭代类型，尝试使用 `for` 循环则会引发 `TypeError`，同时报告该对象是不可迭代的。在用户传递了一个包括非数字化元素的迭代类型的情况下，如 `sum([3.14, 'oops'])`，计算表达式 `total + v` 则自然会引发一个 `TypeError` 异常，然后向调用者发送错误信息

```
unsupported operand type(s) for +: 'float' and 'str'
```

可能稍微不那么明显的错误来自 `sum(['alpha', 'beta'])`。当 `total` 初始化为 0 后，由于表达式 `total + 'alpha'` 的初始计算，则会报告整数与字符串相加是一个错误的尝试。

在本书的其余部分，大多数情况下，执行最少的错误检查和清晰的演示时，我们倾向于更简单的实现。

1.7.2 捕捉异常

有一些关于写代码时如何应对可能出现的异常情况的观点。例如，在计算除法 `x/y` 时，

有一定的风险，当变量 y 为 0 时，引发 `ZeroDivisionError` 异常。在理想情况下，如果程序的逻辑可以表明 y 是非零的，那么就不用担心错误。然而，对于更复杂的代码，或在 y 的值取决于一些外部输入的程序的情况下，仍有发生错误的可能性。

处理特殊情况的第一个理念是三思而后行。想要完全避免异常发生，则要使用积极的条件测试。重温除法的例子，我们可以通过如下写法来避免异常发生：

```
if y != 0:
    ratio = x / y
else:
    ... do something else ...
```

第二个理念通常被 Python 程序员所接受，就是“请求原谅比得到许可更容易”。这句话是计算机科学的先驱 Grace Hopper 提出来的。该观点是指我们不需要花费额外的时间来维护每一个可能发生的异常，只要异常发生时，有一个处理问题的机制就可以了。在 Python 中，这一理念是使用 `try-except` 控制结构来实现的。回顾除法的例子，确保运算正确的代码如下：

```
try:
    ratio = x / y
except ZeroDivisionError:
    ... do something else ...
```

在这种结构中，`try` 块中的代码是要执行的，虽然这个例子中只有一条命令，不过更多的是一个较大块的缩进代码。`try` 块后面会跟着一个或多个 `except` 子句，如果 `try` 块中引发了指定的错误，确定的错误类型和缩进代码块都要被执行。

使用 `try-except` 结构的相对优势是，非特殊情况下高效运行，不需要多余的检查异常条件。然而，在处理异常情况时，使用 `try-except` 结构比使用一个标准的条件语句会需要更多的时间。为此，当我们有理由相信异常情况是相对不可能的，或主动评估条件来避免异常代价异常高时，最好使用 `try-except` 语句。

当用户输入时或读取文件时，异常处理是非常有用的，因为有一些情况是不可预测的。在 1.6.2 节中，我们推荐用语法 `fp = open('sample.txt')` 以读取访问权限打开文件。该命令可能因为多种原因引发 `IOError`，如一个不存在的文件，或者缺乏足够的权限打开文件等。显然，尝试输入命令然后捕捉错误结果比准确预测命令是否成功会更容易。

我们会继续演示一些其他形式的 `try-except` 语法结构，当捕获异常时，异常对象是可以检测出来的。为了能检测到，一个标识符需采用以下语法建立：

```
try:
    fp = open('sample.txt')
except IOError as e:
    print('Unable to open the file:', e)
```

在这种情况下，名称 `e` 表示抛出异常的实例，输出并显示详细的错误消息（如“文件未找到”）。

一个 `try` 语句可能处理不止一种类型的异常。例如，1.6.1 节中的命令：

```
age = int(input('Enter your age in years: '))
```

这个命令可能因为各种各样的原因而出错。如果控制台输入出错，那么调用 `input` 命令会抛出 `EOFError`。如果调用 `input` 成功完成，但是用户没有输入表示一个有效整数的字符，那么 `int` 构造函数会抛出 `ValueError`。如果想要处理两个或两个以上类型的错误，我们可以

使用一个 `except` 语句，像下面的例子：

```
age = -1                      # an initially invalid choice
while age <= 0:
    try:
        age = int(input('Enter your age in years: '))
        if age <= 0:
            print('Your age must be positive')
    except (ValueError, EOFError):
        print('Invalid response')
```

我们希望使用 `except` 语句来捕获异常，使用元组 (`ValueError`, `EOFError`) 来指定错误类型。在这个实现中，我们捕获一个错误，就会输出一个响应，并继续 `while` 循环。我们注意到，当一个错误发生在 `try` 块中时，剩下的语句会直接跳过。在这个例子中，如果在调用 `input` 中出现异常或在后续调用 `int` 构造函数时发生异常，那么 `age` 就不会被赋值，也不会输出你的年龄必须是正数的信息。因为 `age` 值没有改变，所以 `while` 循环也将继续。如果希望在不输出 '`Invalid response`' 的情况下继续 `while` 循环，我们可以写入 `except` 语句：

```
except (ValueError, EOFError):
    pass
```

关键词 `pass` 仅仅是一个声明，但它可以作为一种控制结构的主体。这样，我们就“悄悄”地捕获异常，从而允许 `while` 循环继续。

为了对不同类型的错误提供不同的响应，我们可以使用两个或两个以上 `except` 语句作为 `try` 结构的一部分。在前一个例子中，`EOFError` 表明不可逾越的错误不仅仅是输入了一个错误值。在这种情况下，我们希望能提供更准确的错误信息，或者是允许异常能中断循环并传达给上下文。我们可以通过以下方法实现：

```
age = -1                      # an initially invalid choice
while age <= 0:
    try:
        age = int(input('Enter your age in years: '))
        if age <= 0:
            print('Your age must be positive')
    except ValueError:
        print('That is an invalid age specification')
    except EOFError:
        print('There was an unexpected error reading input.')
        raise                      # let's re-raise this exception
```

在这个实现中，对于 `ValueError` 和 `EOFError` 情况，我们有单独的 `except` 语句。处理 `EOFError` 的语句体依赖于 Python 中的另一种技术。它使用 `raise` 语句且没有其他后续参数来重新抛出相同的目前正在处理的异常。这使我们对异常能提供自己的响应，然后中断 `while` 循环并向上传播。

最后，我们注意到 Python 中 `try-except` 结构的另外两个特征。它允许最后一个 `except` 语句不加特定的错误类型，直接使用“`except:`”来捕获一些其他异常，不过这种技术比较少用，因为对于如何处理一个未知类型的异常是比较困难的。一个 `try` 语句允许有 `finally` 子句，这个子句中的代码总是会被执行，无论是在正常情况下还是在异常情况下，甚至是未捕获异常或重复抛出异常的情况下。通常该代码块是用于清理工作的，如关闭一个文件。

1.8 迭代器和生成器

在 1.4.2 节中，我们使用了以下语句介绍 `for` 循环语法：

```
for element in iterable:
```

我们注意到，Python 中有许多类型的对象可以被定义为可迭代的。基本的容器类型，如列表、元组和集合，都可以定义为迭代类型。此外，字符串可以产生它的字符的迭代，字典可以生成它的键的迭代，文件可以产生它的行的迭代。用户自定义类型也可支持迭代。在 Python 中，迭代的机制基于以下规定：

- 迭代器是一个对象，通过一系列的值来管理迭代。如果变量 `i` 定义为一个迭代器对象，接下来每次调用内置函数 `next(i)`，都会从当前序列中产生一个后续的元素；要是没有后续元素了，则会抛出一个 `StopIteration` 异常。
- 对象 `obj` 是可迭代的，那么通过语法 `iter(obj)` 可以产生一个迭代器。

通过这些定义，`list` 的实例是可迭代的，但它本身不是一个迭代器。如 `data = [1, 2, 4, 8]`，调用 `next(data)` 是非法的。然而，通过语法 `i = iter(data)` 则可以产生一个迭代器对象，然后调用 `next(i)` 将返回列表中的元素。Python 中的 `for` 循环语法使这个过程自动化，为可迭代的对象创造了一个迭代器，然后反复调用下一个元素直至捕获 `StopIteration` 异常。

一般情况下，基于同一个可迭代对象可以创建多个迭代器，同时每个迭代器维护自身演进的状态。不过，迭代器通常通过间接引用回到初始的元素集合维护其状态。例如，对列表实例调用 `iter(data)` 会产生 `list_iterator` 类的一个实例。迭代器不存储自己列表的元素。相反，它保存原始列表的当前索引，该索引指向下一个元素。因此，如果原始列表的内容在迭代器构造之后但在迭代完成之前被修改，迭代器将报告原始列表的更新内容。

Python 还支持产生隐式迭代序列值函数和类，即无须立刻构建数据结构来存储它所有的值。例如，调用 `range(1000000)` 不是返回一个数字列表，而是返回一个可迭代的 `range` 对象。这个对象只有在需要的时候一次性产生百万个值。这样的懒惰计算法有很大的优势。在 `range` 的例子中，它允许执行“`for j in range(1000000):`”这样的循环形式，无须留出内存来存储一百万个值。同样，如果这样一个循环以某种方式被打断，也不用花时间来计算 `range` 中未使用的值。

我们发现懒惰计算在 Python 中的许多库中都用到了，例如，字典类支持方法 `keys()`、`values()` 和 `items()`，它们分别在字典中产生所有 `keys`、`values` 或 `(key, value)` 的“视图”。这些方法没有一个能产生显式的结果列表，相反，产生的视图是基于字典的实际内容的可迭代对象。从这样的迭代中而来的一个显式值的列表可以通过将迭代作为参数调用 `list` 构造器来快速构造，例如，语法 `list(range(1000))` 会生成一个值为 $0 \sim 999$ 的列表实例，然而语法 `list(d.values())` 则会生成一个其元素基于字典 `d` 的当前值生成的列表，同样，我们可以基于所给的迭代器简单地创建元组或集合实例。

生成器

在 2.3.4 节中，我们将解释如何定义一个类——其实例作为迭代器使用。然而，在 Python 中创建迭代器最方便的技术是使用生成器。生成器的语法实现类似于函数，但不返回值。为了显示序列中的每一个元素，会使用 `yield` 语句。作为一个例子，考虑确定一个正整数的所有因子。例如，数字 100 有因子 1, 2, 4, 5, 10, 20, 25, 50, 100。传统的函数可能会产生并返回一个包含所有因子的列表，实现如下：

```
def factors(n):          # traditional function that computes factors
    results = []           # store factors in a new list
    for k in range(1,n+1):
        if n % k == 0:      # divides evenly, thus k is a factor
```

```

results.append(k)      # add k to the list of factors
return results        # return the entire list

```

而生成器中计算这些因子的实现如下：

```

def factors(n):          # generator that computes factors
    for k in range(1,n+1):
        if n % k == 0:    # divides evenly, thus k is a factor
            yield k        # yield this factor as next result

```

注意：我们使用关键字 `yield` 而不是 `return` 来表示结果。这表明在 Python 中，我们正在定义一个生成器，而不是传统的函数。在同一实现中，将 `yield` 和 `return` 语句结合起来是非法的，一个没有返回参数的 `return` 语句也会导致生成器终止执行。如果一个程序员写了一个循环如“`for factor in factors(100):`”，那么会创建一个生成器的实例。在每次循环迭代中，Python 执行程序直到一个 `yield` 语句指出下一个值为止。在这一点上，该程序是暂时中断的，只有当另一个值被请求时才恢复。当控制流自然到达程序的末尾时（或碰到一个零参数的 `return` 语句），会自动抛出一个 `StopIteration` 异常。虽然这个特殊的例子在源代码中使用单一的 `yield` 语句，但生成器可以依赖不同构造中的多个 `yield` 语句，以及由控制的自然流决定的生成序列。例如，我们可以显著提高生成器的效率，在计算整数 n 的因子时，仅仅通过使测试值达到这个数的平方根，同时指出与每个 k 相关联的因子 $n//k$ （除非 $n//k$ 等于 k ）。这样的生成器可以实现如下：

```

def factors(n):          # generator that computes factors
    k = 1
    while k * k < n:      # while k < sqrt(n)
        if n % k == 0:
            yield k
            yield n // k
        k += 1
    if k * k == n:         # special case if n is perfect square
        yield k

```

我们应该注意到，这个生成器的实现与我们的第一个版本不同，因为这些因子不是以严格递增的顺序产生的。例如，`factors(100)` 产生序列 1, 100, 2, 50, 4, 25, 5, 20, 10。

总之，我们在使用生成器而不是传统的函数时，总是强调懒惰计算的好处——只计算需要的数，并且整个系列的数不需要一次性全部驻留在内存中。事实上，一个生成器可以有效地产生数值的无限序列。作为一个例子，斐波那契数列是一个经典的数学序列，初始值为 0，接着值为 1，然后每个后续的值是前两个值的总和。因此，斐波那契数列以 0, 1, 1, 2, 3, 5, 8, 13, … 开始。下面的生成器可以产生这个无穷级数。

```

def fibonacci():
    a = 0
    b = 1
    while True:          # keep going...
        yield a           # report value, a, during this pass
        future = a + b
        a = b             # this will be next value reported
        b = future         # and subsequently this

```

1.9 Python 的其他便利特点

在本节中，我们介绍 Python 的若干特性，这些特性尤其便于编写清晰、简洁的代码。这些语法提供了一些功能，这些功能可以用本章前面提到的功能实现。不过，有时候新语法会有更清晰和直接的逻辑表达。

1.9.1 条件表达式

Python 支持条件表达式的语法，可以取代一个简单的控制结构。一般语法表达式的语法规形式如下：

```
expr1 if condition else expr2
```

对于这种复合表达式，如果条件为真，则计算 *expr1*；否则，计算 *expr2*。这相当于 Java 或 C++ 中的语法“*condition ? expr1 : expr2*”。

考虑这样一个例子，将变量 *n* 的绝对值传递给一个函数（不依赖内置函数 *abs* 的功能）。若使用传统控制结构，可实现如下：

```
if n >= 0:  
    param = n  
else:  
    param = -n  
result = foo(param)      # call the function
```

在条件表达式的语法中，我们可以直接给变量 *param* 赋值，如下所示：

```
param = n if n >= 0 else -n    # pick the appropriate value  
result = foo(param)           # call the function
```

事实上，没有必要将复合表达式赋值给变量。条件表达式本身就可以作为一个函数的参数，如下所示：

```
result = foo(n if n >= 0 else -n)
```

有时，只缩短源代码是有好处的，因为它避免了更繁琐的控制结构。不过，我们建议仅当一个条件表达式能提高源代码的可读性，或者当两个选项的第一个是更“自然”的情况下，为了在语法上强调其重要性才使用。（我们希望当异常发生时可以查看变量的值。）

1.9.2 解析语法

一个很常见的编程任务是基于另一个序列的处理来产生一系列的值。通常，这个任务在 Python 中使用所谓的解析语法后实现很简单。我们先演示列表解析语法，因为这是 Python 支持的第一种形式。它的一般形式如下：

```
[ expression for value in iterable if condition ]
```

我们注意到 *expression* 和 *condition* 都取决于 *value*，而 *if* 子句是可选的。解析计算与下面的传统控制结构计算结果列表在逻辑上是等价的。

```
result = []  
for value in iterable:  
    if condition:  
        result.append(expression)
```

举一个具体的例子，数字 $1 \sim n$ 的平方的列表是 $[1, 4, 9, 16, 25, \dots, n^2]$ ，这可以通过传统方式实现如下：

```
squares = []  
for k in range(1, n+1):  
    squares.append(k*k)
```

使用列表解析，这个逻辑表达式的实现如下：

```
squares = [k*k for k in range(1, n+1)]
```

再举一个例子，1.8节介绍的求一个整数 n 的因子的列表，其使用列表解析的实现如下：

```
factors = [k for k in range(1, n+1) if n % k == 0]
```

Python 支持类似的集、生成器或字典的解析语法。我们通过“计算数字的平方”的例子来比较这些语法。

[k*k for k in range(1, n+1)]	列表解析
{ k*k for k in range(1, n+1) }	集合解析
(k*k for k in range(1, n+1))	生成器解析
{ k : k*k for k in range(1, n+1) }	字典解析

当结果不需要存储在内存中时，生成器语法特别有优势。例如，计算前 n 个数的平方和，生成器语法 total = sum(k * k for k in range(1, n + 1)) 是一种推荐的方法，该方法将列表作为参数使用。

1.9.3 序列类型的打包和解包

Python 提供了另外两个涉及元组和其他序列类型的处理的便利。第一个便利是相当明显的。如果在大的上下文中给出了一系列逗号分隔的表达式，它们将被视为一个单独的元组，即使没有提供封闭的圆括号。例如，命令

```
data = 2, 4, 6, 8
```

会使标识符 data 赋值成元组 (2, 4, 6, 8)，这种行为被称为元组的自动打包。在 Python 中，另一种常用的打包是从一个函数中返回多个值。如果函数体执行命令

```
return x, y
```

就自动返回单个对象，也就是元组 (x, y)。

作为一个对偶的打包行为，Python 也可以自动解包一个序列，允许单个标识符的一系列元素赋值给序列中的各个元素。例如，我们可以这样写

```
a, b, c, d = range(7, 11)
```

这与 $a = 7$ 、 $b = 8$ 、 $c = 9$ 和 $d = 10$ 的赋值效果一样，只要调用 range 函数，就会返回序列中的 4 个值。对于这个语法，右边的表达式可以是任何迭代类型，只要左边的变量数等于右边迭代的元素数。

这种技术可以用来解包一个函数返回的元组。例如，内置的函数 divmod(a, b)，返回这个整除相关的一对数值 $(a/b, a \% b)$ 。尽管调用者可以认为返回值是一个元组，但也可以写成以下形式：

```
quotient, remainder = divmod(a, b)
```

来分别标识返回的元组中的两个值。这个语法也可以使用在 for 循环中，当遍历迭代序列时，就像：

```
for x, y in [(7, 2), (5, 8), (6, 4)]:
```

在这个例子中，将循环执行 3 次。第一次为 $x = 7$, $y = 2$ ，然后以此类推。这种循环的风格常用于遍历由字典类的 item() 方法返回的键值对，就像：

```
for k, v in mapping.items():
```

同时分配

自动打包和解包结合起来就是同时分配技术，即我们显式地将一系列的值赋给一系列标识符，所用语法为：

```
x, y, z = 6, 2, 5
```

实际上，该赋值右边将自动打包成一个元组，然后自动解包，将它的元素分配给左边的三个标识符。

当使用同时分配技术时，所有表达式都是在对左边的变量赋值之前先计算右侧。这一点很重要，因为它提供了一种方便的方法，用来交换与两个变量相关联的值：

```
j, k = k, j
```

有了这个命令，原来的 `k` 值赋给 `j`，原来的 `j` 值赋给 `k`。如果没有同时分配技术，那么一个典型的交换需要使用一个临时变量，如：

```
temp = j
j = k
k = temp
```

有了同时分配技术，在执行交换时，代表右边打包值的未命名元组相当于隐式的临时变量。

使用同时分配技术可以大大简化代码演示。作为一个例子，我们考虑 1.8 节生成的斐波那契数列。原来的代码需要对序列开始的变量 `a` 和 `b` 初始化。在每一次循环中，其目标是给 `a` 和 `b` 分别赋予 `b` 和 `a + b` 的值。当时我们完成这个目标时使用了第三个变量。有了同时分配技术，生成器直接按以下方式实现：

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a+b
```

1.10 作用域和命名空间

当在 Python 中以 `x + y` 计算两数的和时，`x` 和 `y` 这两个名称一定要与先前作为值的对象相关联；如果没有找到相关定义，会抛出一个 `NameError` 异常。确定与标识符相关联的值的过程称为名称解析。

每当标识符分配一个值，这个定义都有特定的范围。最高级赋值通常是全局范围，对于在函数体内的赋值，其范围通常是该函数调用的局部。因此，函数体内的 `x = 5` 对外部函数标识符 `x` 没有影响。

Python 中的每一个定义域使用了一个抽象名称，称为命名空间。命名空间管理当前在给定作用域内定义的所有标识符。图 1-8 描绘了两个命名空间，一个是 1.5 节调用 `count` 函数的命名空间，另一个是在函数执行过程中本地的命名空间。

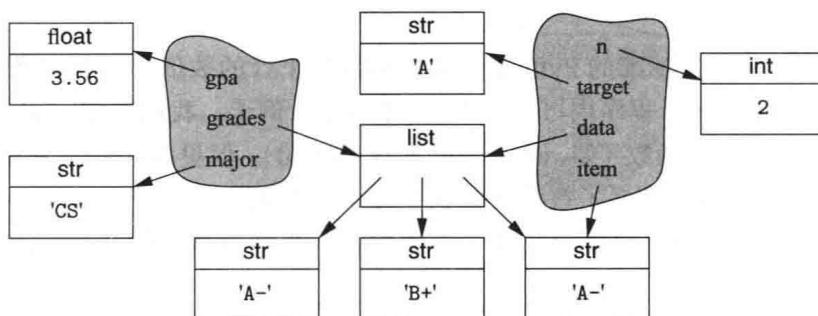


图 1-8 两个命名空间的描述都与 1.5 节定义的用户调用的 `count(grade, 'A')` 相关。左边的是调用者的命名空间，右边的是函数本地范围的命名空间

Python 实现命名空间是用自己的字典将每个标识符字符串（例如 'n'）映射到其相关的值。Python 还提供了几种方法来检查一个给定的命名空间。函数 `dir()` 报告给定命名空间中的标识符的名称（即字典的键），而函数 `var()` 返回完整的字典。默认情况下，调用 `dir()` 和 `var()` 报告的是执行过程中本地封闭的命名空间。

在命令中指示标识符时，Python 会在名称解析过程中搜索一系列的命名空间。首先，搜索的是所给名字的本地命名空间，若没找到，则搜索外一层的命名空间，然后以此类推。在 2.5 节讨论面向对象的处理时，我们还会继续讨论命名空间，我们会发现每个对象都有自己的命名空间存储其属性，每个类也都有自己的命名空间。

第一类对象

在编程语言的术语中，第一类对象是一些可以分配给一个标识符的类型的实例，可作为参数传递，或由一个函数返回。我们在 1.2.3 节介绍的所有数据类型，如 `int` 和 `list`，无疑都是 Python 中的第一类类型，函数和类也作为第一类对象处理。例如：

```
scream = print      # assign name 'scream' to the function denoted as 'print'
scream('Hello')    # call that function
```

在这个例子中，我们没有创建新的函数，只是简单地将 `scream` 定义为现有 `print` 函数的别名。使用这个例子还有一个目的，它说明了 Python 允许一个函数作为参数传递到另一个函数的机制。在 1.5.2 节，我们注意到，当计算最大值时，内置函数 `max()` 可接收一个可选的关键字参数去指定一个非默认的序列。例如，调用者可以使用语法 `max(a, b, key = abs)`，以确定哪个值有更大的绝对值。在该函数的主体中，形式参数 `key` 是将要赋值给实际参数 `abs` 的标识符。

就命名空间而言，赋值语句如 `scream = print` 将标识符 `scream` 引入当前的命名空间，其值表示的是内置函数对象 `print`。同样的机制也可以应用在用户定义的函数声明中。例如，1.5 节的 `count` 函数的声明语法：

```
def count(data, target):
    ...
```

这样一个声明将标识符 `count` 引入了命名空间，它的值是一个表示其实现的函数实例。类似的，新定义的类的名称与该类的值的表示形式相关联（我们将在下一章介绍类的定义）。

1.11 模块和 import 语句

我们已经介绍了 Python 内置命名空间定义的很多函数（例如 `max`）和类（例如 `list`）。基于 Python 的版本，我们认为大约有 130 ~ 150 种确实重要的定义包含在内置命名空间中。

除了内置的定义外，标准的 Python 分配包括数以千计的数值、函数以及被组织在附加库中的类（称为模块，一个程序内可以导入）。作为一个例子，我们考虑 `math` 模块。虽然内置命名空间包含一些数学函数（如，`abs`、`min`、`max`、`round`），但更多的是归为 `math` 模块（如，`sin`、`cos`、`sqrt`）。该模块还定义了数学常数 `pi` 和 `e` 的近似值。

Python 的 `import` 声明可以将定义从一个模块载入当前命名空间。`import` 语句的语法形式如下：

```
from math import pi, sqrt
```

这个命令将在 `math` 模块定义的 `pi` 和 `sqrt` 添加到当前的命名空间，允许直接使用标识符 `pi`、

或调用函数 `sqrt(2)`。如果有许多定义来自导入的同一模块，则可以使用 `*`，如 `from math import *`，但这种形式应谨慎使用。危险在于，模块中定义的一些名称可能与当前命名空间中的名称冲突（或与导入的另一个模块冲突），而导入的模块会产生新定义去替换原有的定义。

另一种可以用于从相同模块访问许多定义的方法是导入模块本身，使用如下语法：

```
import math
```

同时将标识符 `math` 以及作为其值的模块引入当前的命名空间（模块在 Python 中是第一类对象）。一旦引入，模块中的定义可以用一个完全限定的名称来访问，例如 `math.pi` 或者 `math.sqrt(2)`。

创建新模块

要创建一个新模块，你只需要简单地把相关的定义放在一个扩展名为 `.py` 的文件里。这些定义可以从同一工程目录下的其他 `.py` 文件中导入。例如，如果我们把计数函数的定义（见 1.5 节）放到 `utility.py` 文件中，那么可以使用语法 `from utility import count` 来导入该 `count` 函数。

值得注意的是，当第一次导入模块时，模块源代码的顶层命令会被执行，就好像这个模块是自己的脚本。在模块中，如果该模块被直接调用作为一个脚本，而不是从另一个脚本导入模块时，将执行该模块中嵌入命令的特殊构造。

这样的命令应该放在如下形式的条件语句中：

```
if __name__ == '__main__':
```

以我们假设的 `utility.py` 模块为例，如果解释器通过 Python `utility.py` 的命令启动，但 `utility.py` 模块不是从其他上下文中导入，这些命令将会执行。这种方法通常用于嵌入模块的单元测试。我们将在 2.2.4 节进一步讨论单元测试。

现有模块

表 1-7 给出了一些可用的与数据结构的研究相关的模块小结。之前我们已经简略地讨论过 `math` 模块了。在本节的其余部分，我们将重点介绍另一个对于我们在本书后面研究的一些数据结构和算法特别重要的模块。

表 1-7 一些与数据结构和算法相关的现有 Python 模块

模块名	描述
<code>array</code>	为原始类型提供了紧凑的数组存储
<code>collections</code>	定义额外的数据结构和包括对象集合的抽象基类
<code>copy</code>	定义通用函数来复制对象
<code>heapq</code>	提供基于堆的优先队列函数（参见 9.3.7 节）
<code>math</code>	定义常见的数学常数和函数
<code>os</code>	提供与操作系统交互
<code>random</code>	提供随机数生成
<code>re</code>	对处理正则表达式提供支持
<code>sys</code>	提供了与 Python 解释器交互的额外等级
<code>time</code>	对测量时间或延迟程序提供支持

伪随机数生成

Python 的 random 模块能够生成伪随机数，即数字是统计上随机的（但不一定是真正随机的）。伪随机数生成器使用一个确定的公式来根据一个或多个过去数字生成的序列来产生下一个数。事实上，一个简单而流行的伪随机数生成器选择它的下一个数字是基于最近选择的数和一些额外的参数，所使用的公式如下：

```
next = (a*current + b) % n;
```

这里的 a 、 b 和 n 是适当选择的整数。Python 使用更先进的技术梅森旋转算法（Mersenne twister）。事实证明这些技术所产生的序列是系统统一的，对于大多数需要随机数字的应用程序（比如游戏）通常是足够的。对于应用程序，如计算机安全设置这样一个需要不可预测的随机序列的程序，就不应该使用这种公式。相反，我们需要真正随机的理想样本，如来自外太空的静态无线电。

由于伪随机数生成器中的下一个数是由前一个数决定的，这样的发生器总是需要一个开始的数字，这就是所谓的种子。一个给定的种子产生的序列将永远是相同的。要在每次程序运行时得到不同序列，一个常见的技巧是每次运行时使用不同的种子。例如，我们可以用来自某个用户输入的或当前以毫秒为单位的系统时间作为种子。

Python 的 random 模块通过定义一个 Random 类支持伪随机数生成，这个类的实例作为有独立状态的生成器（见表 1-8）。这允许一个程序的不同方面依靠自己的伪随机数生成器，因此，一个生成器的调用不影响由另一个生成器产生的数字的序列。为了方便，Random 类支持的所有方法在 random 模块中都有支持的独立函数（基本上单个的生成器实例可用于所有的顶级调用）。

表 1-8 Random 类的实例支持的方法和 random 模块的顶级函数

语 法	描 述
seed(hashable)	基于参数的散列值初始化伪随机数生成器
random()	在开区间 (0.0, 1.0) 返回一个伪随机浮点值
randint(a, b)	在闭区间 $[a, b]$ 返回一个伪随机整数
randrange(start, stop, step)	在参数指定的 Python 标准范围内返回一个伪随机整数
choice(seq)	返回一个伪随机选择的给定序列中的元素
shuffle(seq)	重新排列给定的伪随机序列中的元素

1.12 练习

请访问 www.wiley.com/college/goodrich 以获得练习帮助。

巩固

- R-1.1 编写一个 Python 函数 `is_multiple(n, m)`，用来接收两个整数值 n 和 m ，如果 n 是 m 的倍数，即存在整数 i 使得 $n = mi$ ，那么函数返回 True，否则返回 False。
- R-1.2 编写一个 Python 函数 `is_even(k)`，用来接收一个整数 k ，如果 k 是偶数返回 True，否则返回 False。但是，函数中不能使用乘法、除法或取余操作。
- R-1.3 编写一个 Python 函数 `minmax(data)`，用来在数的序列中找出最小数和最大数，并以一个长度为 2 的元组的形式返回。注意：不能通过内置函数 `min` 和 `max` 来实现。
- R-1.4 编写一个 Python 函数，用来接收正整数 n ，返回 $1 \sim n$ 的平方和。

- R-1.5 基于 Python 的解析语法和内置函数 sum，写一个单独的命令来计算练习 R-1.4 中的和。
- R-1.6 编写一个 Python 函数，用来接收正整数 n ，并返回 $1 \sim n$ 中所有奇数的平方和。
- R-1.7 基于 Python 的解析语法和内置函数 sum，写一个单独的命令来计算练习 R-1.6 中的和。
- R-1.8 Python 允许负整数作为序列的索引值，如一个长度为 n 的字符串 s ，当索引值 $-n \leq k < 0$ 时，所指的元素为 $s[k]$ ，那么求一个正整数索引值 $j \geq 0$ ，使得 $s[j]$ 指向的也是相同的元素。
- R-1.9 要生成一个值为 50, 60, 70, 80 的排列，求 range 构造函数的参数。
- R-1.10 要生成一个值为 8, 6, 4, 2, 0, -2, -4, -6, -8 的排列，求 range 构造函数中的参数。
- R-1.11 演示怎样使用 Python 列表解析语法来产生列表 [1, 2, 4, 8, 16, 32, 64, 128, 256]。
- R-1.12 Python 的 random 模块包括一个函数 choice(data)，可以从一个非空序列返回一个随机元素。Random 模块还包含一个更基本的 randrange 函数，参数化类似于内置的 range 函数，可以在给定范围内返回一个随机数。只使用 randrange 函数，实现自己的 choice 函数。

创新

- C-1.13 编写一个函数的伪代码描述，该函数用来逆置 n 个整数的列表，使这些数以相反的顺序输出，并将该方法与可以实现相同功能的 Python 函数进行比较。
- C-1.14 编写一个 Python 函数，用来接收一个整数序列，并判断该序列中是否存在一对乘积是奇数的互不相同的数。
- C-1.15 编写一个 Python 函数，用来接收一个数字序列，并判断是否所有数字都互相不同（即它们是不同的）。
- C-1.16 在 1.5.1 节 scale 函数的实现中，循环体内执行的命令 $data[j] *= factor$ 。我们已经说过这个数字类型是不可变的，操作符 *= 在这种背景下使用是创建了一个新的实例（而不是现有实例的变化）。那么 scale 函数是如何实现改变调用者发送的实际参数呢？
- C-1.17 1.5.1 节 scale 函数的实现如下。它能正常工作吗？请给出原因。

```
def scale(data, factor):
    for val in data:
        val *= factor
```

- C-1.18 演示如何使用 Python 列表解析语法来产生列表 [0, 2, 6, 12, 20, 30, 42, 56, 72, 90]。
- C-1.19 演示如何使用 Python 列表解析语法在不输入所有 26 个英文字母的情况下产生列表 ['a', 'b', 'c', ..., 'z']。
- C-1.20 Python 的 random 模块包括一个函数 shuffle(data)，它可以接收一个元素的列表和一个随机的重新排列元素，以使每个可能的序列发生概率相等。random 模块还包括一个更基本的函数 randint(a, b)，它可以返回一个从 a 到 b （包括两个端点）的随机整数。只使用 randint 函数，实现自己的 shuffle 函数。
- C-1.21 编写一个 Python 程序，反复从标准输入读取一行直到抛出 EOFError 异常，然后以相反的顺序输出这些行（用户可以通过键按 Ctrl+D 结束输入）。
- C-1.22 编写一个 Python 程序，用来接收长度为 n 的两个整型数组 a 和 b 并返回数组 a 和 b 的点积。也就是返回一个长度为 n 的数组 c ，即 $c[i] = a[i] \cdot b[i]$ ，for $i = 0, \dots, n - 1$ 。
- C-1.23 给出一个 Python 代码片段的例子，编写一个索引可能越界的元素列表。如果索引越界，程序应该捕获异常结果并打印以下错误消息：
“Don’t try buffer overflow attacks in Python!”
- C-1.24 编写一个 Python 函数，计算所给字符串中元音字母的个数。
- C-1.25 编写一个 Python 函数，接收一个表示一个句子的字符串 s ，然后返回该字符串的删除了所有标点符号的副本。例如，给定字符串 "Let’s try, Mike."，这个函数将返回 "Lets try Mike"。

- C-1.26 编写一个程序，需要从控制台输入 3 个整数 a 、 b 、 c ，并确定它们是否可以在一个正确的算术公式（在给定的顺序）下成立，如 “ $a + b = c$ ” “ $a = b - c$ ” 或 “ $a * b = c$ ”。
- C-1.27 在 1.8 节中，我们对于计算所给整数的因子时提供了 3 种不同的生成器的实现方法。1.8 节末尾处的第三种方法是最有效的，但我们注意到，它没有按递增顺序来产生因子。修改生成器，使得其按递增顺序来产生因子，同时保持其性能优势。
- C-1.28 在 n 维空间定义一个向量 $\mathbf{v} = (v_1, v_2, \dots, v_n)$ 的 p 范数，如下所示：

$$\|\mathbf{v}\| = \sqrt[p]{v_1^p + v_2^p + \dots + v_n^p}$$

对于 $p=2$ 的特殊情况，这就成了传统的欧几里得范数，表示向量的长度。例如，一个二维向量坐标为 $(4, 3)$ 的欧几里得范数为 $\sqrt{4^2 + 3^2} = \sqrt{16 + 9} = \sqrt{25} = 5$ 。编写 norm 函数，即 norm(\mathbf{v} , p)，返回向量 \mathbf{v} 的 p 范数的值，norm(\mathbf{v})，返回向量 \mathbf{v} 的欧几里得范数。你可以假定 \mathbf{v} 是一个数字列表。

项目

- P-1.29 编写一个 Python 程序，输出由字母 'c', 'a', 't', 'd', 'o', 'g' 组成的所有可能的字符串（每个字母只使用 1 次）。
- P-1.30 编写一个 Python 程序，输入一个大于 2 的正整数，求将该数反复被 2 整除直到商小于 2 为止的次数。
- P-1.31 编写一个可以“找零钱”的 Python 程序。程序应该将两个数字作为输入，一个是需要支付的钱数，另一个是你给的钱数。当你需要支付的和所给的钱数不同时，它应该返回所找的纸币和硬币的数量。纸币和硬币的值可以基于之前或现任政府的货币体系。试设计程序，以便返回尽可能少的纸币和硬币。
- P-1.32 编写一个 Python 程序来模拟一个简单的计算器，使用控制台作为输入和输出的专用设备。也就是说，计算器的每一次输入做一个单独的行，它可以输入一个数字（如 1034 或 12.34）或操作符（如 + 或 =）。每一次输入后，应该输出计算器显示的结果并将其输出到 Python 控制台。
- P-1.33 编写一个 Python 程序来模拟一个手持计算器，程序应该可以处理来自 Python 控制台（表示 push 按钮）的输入，每个操作执行完毕后将内容输出到屏幕。计算器至少应该能够处理基本的算术运算和复位 / 清除操作。
- P-1.34 一种惩罚学生的常见方法是让他们将一个句子写很多次。编写独立的 Python 程序，将以下句子“ I will never spam my friends again.” 写 100 次。程序应该对每个句子进行计数，另外，应该有 8 次不同的随机输入错误。
- P-1.35 生日悖论是说，当房间中人数 n 超过 23 时，那么该房间里有两个人生日相同的可能性是一半以上。这其实不是一个悖论，但许多人觉得不可思议。设计一个 Python 程序，可以通过一系列随机生成的生日的实验来测试这个悖论，例如可以 $n = 5, 10, 15, 20, \dots, 100$ 测试这个悖论。
- P-1.36 编写一个 Python 程序，输入一个由空格分隔的单词列表，并输出列表中的每个单词出现的次数。在这一点上，你不需要担心效率，因为这个问题会在这本书后面的部分予以解决。

扩展阅读

Python 的官方网站 (<http://www.python.org>) 有大量的资料，包括教程和内置函数、类以及标准模块的完整文档。Python 解释器本身是一个有用的参考，为交互式命令帮助 (foo) 提供了一切函数、类和 foo 标识的模块的文档。

对 Python 编程提供参考的书包括由 Campbell 等^[22]、Cedar^[25]、Dawson^[32]、Goldwasser 和 Letscher^[43]、Lutz^[72]撰写的书，更完整的 Python 参考书有 Beazley^[12] 和 Summerfield^[91]撰写的书。

面向对象编程

2.1 目标、原则和模式

顾名思义，面向对象模式中的主体被称为对象（object）。每个对象都是类（class）的实例（instance）。类呈献给外部世界的是该类实例中各对象的一种简洁、一致的概括，没有太多不必要的细节，也没有提供访问类内部工作过程的接口。类的定义通常详细规定了对象包含的实例变量（instance variable），又称数据成员（data member）；还规定了对象可以执行的方法（methods），又称成员函数（member function）。这种计算理念意在实现几个设计目标和设计原则，这就是我们在这章将要讨论的内容。

2.1.1 面向对象的设计目标

软件的实现应该达到健壮性（robustness）、适应性（adaptability）和可重用性（reusability）目标，如图 2-1 所示。

健壮性

每个优秀的程序设计者都想开发正确的软件，这就是说在应用程序中事先考虑到的所有输入都会产生一个正确的输出。除此

之外，我们希望软件变得更健壮（robust），更确切地说，希望软件能处理我们在应用程序中没有明确定义的异常输入。例如，如果一个程序需要正整数（也许是代表一件商品的价格），然而却输入一个负整数，那么这个程序需要“优雅”地从这个错误中恢复。更重要的是，不健壮的软件可能是致命的，比如在性命攸关的应用程序（life-critical application）里，软件的一个错误可能会导致健康受损甚至丧命。这一点在 20 世纪 80 年代后期的 Therac-25 意外中被发现。1985 ~ 1987 年间，一个放射医疗机器给 6 名患者使用的放射严重过量，其中有人死于由辐射过量引起的并发症。以上的 6 起事故都是软件错误导致的。

适应性

现代软件应用程序，比如网页浏览器和互联网搜索引擎，通常包含使用了多年的大型程序。软件需要随着时间不断地优化，以应对外部环境中条件的改变。于是，高质量软件的另一个重要目标是实现适应性（adaptability）（又称可进化性（evolvability））。这个概念与可移植性（portability）有关。可移植性是指软件以最少的改变运行在不同的硬件和操作系统平台上。用 Python 编写软件的一个好处是语言本身具有很好的可移植性。

可重用性

与适应性相似，我们希望软件也是可重用的，更确切地说，同样的代码可以在不同系统中的各种应用中。开发高质量软件的开销可能很昂贵，如果能把软件设计成高度可重用的，那么就会减少开发软件的开销。但是，这种可重用性应该谨慎使用，在 Therac-25 意外中，主要的软件错误之一就来源于对 Therac-20 软件的不恰当重用（Therac-20 软件不是面向对象

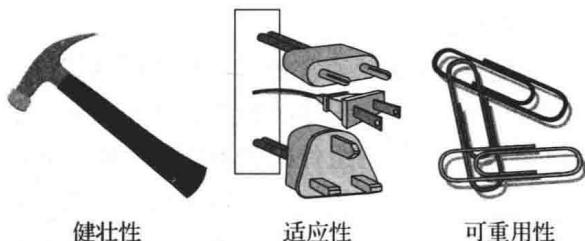


图 2-1 面向对象设计的目标

的，也不是为 Therac-25 所使用的硬件平台设计的）。

2.1.2 面向对象的设计原则

为了实现上述目标，面向对象方法的首要原则如下（见图 2-2）：

- 模块化
- 抽象化
- 封装

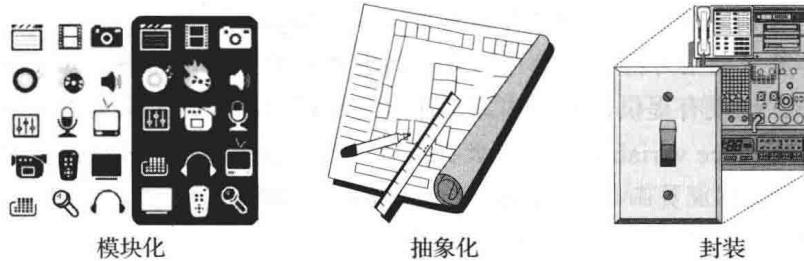


图 2-2 面向对象的设计原则

模块化

现代软件系统通常包含一些不同的组件，为了使整个系统正常工作，这些组件必须正确地合作。恰当地组织这些组件，才能保证它们合作正常。模块化指的是一种组织原则，在这个原则中，不同的组件归为不同的功能单元。

用现实世界作比，一座房子或公寓可以视为由一些不同的相互作用的单元组成，比如电力系统、加热系统、冷却系统、水暖系统和建筑结构。有些人视这些系统为一大堆杂乱的电线、通风口、管道和板材，组织架构师则不然，他们设计房子和公寓时会将它们视为单独的模块，让这些模块在恰当的方式下相互作用。这样，他就能使用模块化的思想理出清晰的思路。这种思路提供了一个从组织功能到可管理单元的自然方法。

同样，在软件系统中采用模块化还可以为实施搭建清晰而强大的组织框架。我们已经知道，在 Python 中，模块（module）是一个源代码中定义的密切相关的函数和类的集合。比如 Python 标准库包括 math 模块，该模块提供了关键的数学常量和函数的定义，还包括提供了与操作系统的交互支持的 os 模块。

模块化的使用还有助于支持 2.1.1 节中列出的目标。在形成大的软件系统之前，不同的组件是易于测试和调试的。此外，一个完整系统中的错误可能会追溯到相对独立的特定组件中。因此，健壮性被大大地提高。模块化结构还可以加强软件的重用性。如果软件模块用通用的方式来写，那么当上下文中出现相关需求时可以重用模块。这在数据结构的研究中是特别常见的，它们通常被定义得足够抽象，并且在很多应用程序中被重用。

抽象化

抽象化（abstraction）是指从一个复杂的系统中提炼出最基础的部分。通常，描述系统的各个部分涉及给这些部分命名和解释它们的功能。将抽象模式应用于数据结构的设计便产生了抽象数据类型（Abstract Data Types, ADT）。ADT 是数据结构的数学模型，它规定了数据存储的类型、支持的操作和操作参数的类型。ADT 定义每个操作要做什么（what）而不是怎么做（how）。我们通常参考 ADT 作为其公共接口（public interface）所支持的行为的集合。

Python 作为一种编程语言，提供了大量有关接口的说明。Python 使用一种被称为鸭子类型的机制应对隐式抽象类型。作为一种解释程序和动态类型的语言，在 Python 中没有“编译时”检查数据类型，并且对于抽象基类的声明没有正式的要求。相反，程序员假设对象支持一系列已知的行为，如果这些假设不成立，解释程序将出现一个运行错误。“鸭子类型”这个概念来源于诗人 James Whitcomb Riley 的一句话：“当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。”

更正式地说，Python 用一种称为抽象基类（Abstract Base Class，ABC）的机制支持抽象数据类型。一个抽象基类不能被实例化（换言之，你不能直接创建该类的实例），但它规定了一个或多个常用的方法，抽象化的所有实现都必须包括该方法。通过从一个或多个抽象类中继承的具体类（concrete classes）来实现 ABC，同时提供由 ABC 声明这些方法的实现。虽然我们为了简单起见而忽略这些声明，但 Python 的 ABC 模块为 ABC 提供正式的支持。我们将用到一些现有的来自 Python 集合模块的抽象基类，其中包括几种常用数据结构 ADT 的定义和其中一些抽象的具体实现。

封装

面向对象设计的另一个重要原则是封装（encapsulation）。软件系统的不同组件不应显示其各自实现的内部细节。封装的主要优点之一就是它给程序员实现组件细节的自由，而不用关心其他程序员写的其他依赖于这些内部代码的程序。程序设计者对于组件的唯一约束是为这些组件保持公共接口，其他程序设计者将会编写依赖于该接口的代码。封装提供了健壮性和适应性，因为它允许改变程序一部分的实现细节而不影响其他部分，因此，修复漏洞或者给组件中增加相对本地更改的新功能就变得更容易。

在这本书中，我们将遵循封装的原则，说明数据结构的哪些方面被认定为公共部分，哪些方面被认定为内部细节。也就是说，Python 为封装提供了宽泛的支持。按照惯例，以单下划线开头的类成员（数据成员和成员函数）的名称（如，`_secret`）被认定为非公开的，而且不应该被依赖。根据这些约定，自动生成文档时会忽略这些内部成员。

2.1.3 设计模式

面向对象的设计有助于实现健壮的、可适应的、可重用的软件。然而，设计好的代码不仅需要简单地理解面向对象的方法，更需要有效地利用面向对象的设计技术。

为了设计高质量的、简洁的、正确的、可重用的面向对象软件，计算研究人员和从业人员已经开发出多种组织的概念和方法。本书特别关注的是设计模式（design pattern）的概念，它描述了“典型”软件设计问题的解决方案。一种可以应用于不同情况的解决方案提供了通用模板的模式。它通过一个方式描述解决方法的主要元素，该方式是抽象的而且可以专门用于所面临的具体问题。模式包括一个名称（它标识了该模式）、一个语境（它描述应用该模式的情况）、一个模板（它描述如何应用该模式）以及一个结果（它描述和分析该模式会产生什么结果）。

在本书中，我们介绍一些设计模式，同时展示它们如何被持续地应用于数据结构和算法的实现。这些设计模式被分为两组——解决算法设计问题的模式和解决软件工程问题的模式。所讨论的算法设计模式包括以下内容：

- 递归（第 4 章）
- 摊销（5.3 节和 11.4 节）

- 分治法（12.2.1节）
- 去除法，又称减治法（12.7.1节）
- 暴力算法（13.2.1节）
- 动态规划（13.3节）
- 贪心法（13.4.2节、14.6.2和14.7节）

所讨论的软件工程模式包括以下内容：

- 迭代器（1.8节和2.3.4节）
- 适配器（6.1.2节）
- 位置（7.4节和8.1.2节）
- 合成（7.6.1节、9.2.1节和10.1.4节）
- 模板方法（2.4.3节、8.4.6节、10.1.3节、10.5.2节和11.2节）
- 定位器（9.5.1节）
- 工厂模式（11.2.1节）

然而，与其在这里解释每种设计模式的概念，不如通过不同的章节来介绍它们。对于每种模式，不论是用于设计算法还是软件工程，我们都会解释其一般用法，并且至少给出一个具体的例子来进行说明。

2.2 软件开发

传统的软件开发包括几个阶段。其中3个主要阶段如下：

- 1) 设计。
- 2) 实现。
- 3) 测试和调试。

在本节中，我们将简要讨论这些阶段所扮演的角色，介绍在利用Python编程时一些好的做法，包括编码风格、命名约定、文档和单元测试。

2.2.1 设计

对于面向对象编程，设计步骤也许是软件开发过程中最重要的阶段。因为在决定如何把程序的工作分成若干个的设计步骤中，我们决定这些类的交互方式、将要存储的数据和将要执行的功能。事实上，程序设计者刚开始面临的主要的挑战之一是决定用什么类去实现程序的功能。虽然一般的计划都很难总结，但这里有一些我们可以应用的经验规则，为确定如何设计类提供方便。

- 责任（responsibility）：把这些工作分为不同的角色（actor），它们有各自不同的责任。试着用行为动词描述责任。这些角色将形成程序的类。
- 独立（independence）：在尽可能独立于其他类的前提下规定每个类的工作。细分各个类的责任，这样每个类在程序的某个方面上就有自主权。把数据作为那些需要控制和访问这些数据的类的实例变量。
- 行为（behavior）：仔细且精确地为每个类定义行为，这样与它进行交互的其他类可以很好地理解这个由类执行的动作结果。这些行为将定义该类执行的方法，并且，类的接口（interface）是一系列类的行为，因为这些类构成了其他代码与类中对象交互的方法。

面向对象程序设计的关键是定义类和它们的实例变量及方法。随着时间的推移，一个好的程序设计者在执行这些任务时自然会探寻更好的技巧，就好像是经验在教他去注意项目需要的模式，该模式与他之前见过的模式相匹配。

CRC 卡 (Class-Responsibility-Collaborator) 是一个用于开发初始的高层次设计项目的通用工具。CRC 卡是细分程序所需工作的简单的索引卡。该工具的主要思路是每个卡代表一个组件，该组件最终将成为程序的类。我们把每个组件的名字写在索引卡的顶部。把组件的责任写在卡的左边，在卡的右边列出组件的合作者，即该组件将与之交互完成责任的其他组件。

设计过程通过行为 / 角色周期反复迭代，我们首先确定一个行为（即责任），接着决定一个最适合执行该行为的角色（即组件）。在这个过程中，使用索引卡（而不是更大的纸），我们的依据是每个组件应该有一个小的责任和合作者集合。强制遵循这个规则有助于保持单个类易于管理。

作为设计采取的形式，解释和记录设计的标准方法是使用 UML (Unified Modeling Language) 图来表达程序的组织。UML 图是一个表达面向对象软件设计的标准视觉记号。一些计算机辅助工具可以构建 UML 图。类图 (class diagram) 就是一种 UML 图。图 2-3 给出了这样一个代表消费信用卡类的图的例子。图包括三部分内容，第一部分指明类的名字，第二部分指明推荐的实例变量，第三部分指明类的方法。在 2.2.3 节中，我们将讨论命名规则。在第 2.3.1 节中，我们将提供一个完整的以该设计为依据的 Python CreditCard 类的实现方法。

类: 信用卡		
域:	_customer	_balance
	_bank	_limit
行为:	get_customer()	get_balance()
	get_bank()	get_limit()
	get_account()	charge(price)
	make_payment(amount)	

图 2-3 推荐的 CreditCard 类的类图

2.2.2 伪代码

作为在设计实现前的中间步骤，通常要求程序设计者通过一种专门为人们准备的方法来描述算法。这种描述被称为伪代码 (pseudo-code)。伪代码不是计算机程序，但是比平常文章更加结构化。伪代码是自然语言和高级编程结构的混合，用于描述隐藏在数据结构和算法实现之后的主要编程思想。因为伪代码是为读者而设计的，而不是为计算机设计的，因此我们可以交流复杂的思想，而不用担心低层具体细节的实现。同时，我们不应该注释过多的重要步骤。就像人类沟通一样，寻找正确的平衡是一种重要技能，这些技能可以在实践中积累和强化。

在本书中，我们依靠伪代码样式，并使用数学符号和字母注释的组合，使得对于 Python 程序设计者来说该伪代码风格是清晰的。例如，我们也许会用短语“indicate an error”代替正式的语句。遵循 Python 的惯例，我们依靠缩进来表示控制结构的程度，依靠

从 $A[0]$ 到 $A[n - 1]$ 的索引符号给长度为 n 的序列 A 编号。不过，在伪代码中，我们选择把注释放入大括号 {} 中，而不是用 Python 中的 # 字符。

2.2.3 编码风格和文档

程序应该被设计得易于阅读和理解。因此，好的程序设计者应该注意自己的编码风格，并且形成一种无论是对人还是计算机的交流都有好处的风格。编码风格的惯例在不同编程团体中是不同的。在网站 <http://www.python.org/dev/peps/pep-0008/> 中可得到官方的 Python 代码风格指南（Style Guide for Python Code）。

我们采取的主要原则如下：

- Python 代码块通常缩进 4 个空格。但是，为了避免代码段超过本书的边界，我们以 2 个空格作为每一级的缩进。因为不同系统中以不同的宽度显示制表符，而且 Python 解释器视制表符和空格是不同的字符，所以强烈建议避免使用制表符。许多能识别 Python 语言的编辑器会自动用适量的空格代替制表符。
- 标识符命名要有意义。试着选择大家易于理解名字，选择能反映行为、责任或其命名的数据的名字。
 - 类（不同于 Python 的内置类）应该以首字母大写的单数名词（例如，Date 而不是 date 或 Dates）作为名字。当多个单词连接起来形成一个类的名字时，它们应该遵循所谓的“骆驼拼写法”规则。即在该规则中，每个单词的首字母要大写（例如，CreditCard）。
 - 函数，包括类的成员函数，应该小写。如果将多个单词组合起来，它们就应该用下划线隔开（例如，make_payment）。函数的名字通常应该是一个描述它的作用的动词。但是，如果这个函数的唯一目的是返回一个值，那么函数名可以是一个描述返回值的名词（例如，sqrt 而不是 calculate_sqrt）。
 - 标识某个对象（例如，参数、实例变量或本地变量）的名字应该是一个小写的名词（例如，price）。有时候，当我们使用一个大写字母来表示一个数据结构的名称时，会不遵守这条规则（如 tree T）。
 - 传统上用大写字母并用下划线隔开每个单词的标识符代表一个常量值（例如，MAX_SIZE）。

回顾我们讨论的封装，在任何情况下，以单下划线开头的标识符（例如，_secret）意在表明它们只为类或模块“内部”使用，而不是公共接口的一部分。

- 用注释给程序添加说明，解释有歧义或令人困惑的结构。内嵌的行注释有助于快速理解代码有好处。在 Python 中，# 字符后的内容表示注释，如：

```
if n % 2 == 1:      # n is odd
```

多行注释块可以很好地解释更复杂的代码段。在 Python 中，有专门的多行字符串，通常用三引号（"""）表示，这种注释对程序执行没有任何影响。在下一节中，我们将讨论使用块注释作为文档。

文档

Python 使用一个称作 docstring 的机制为在源码中直接插入文档提供完整的支持。从形式上讲，任何出现在模块、类、函数（包括类的成员函数）主体中的第一个语句的字符串都

被认为是 docstring。按照惯例，这些字符串应该限定在三引号（"""）中。例如，1.5.1 节的缩放功能的版本可以有如下记录：

```
def scale(data, factor):
    """ Multiply all entries of numeric data list by the given factor."""
    for j in range(len(data)):
        data[j] *= factor
```

对于 docstring，通常用三引号字符串分隔符，即使像上面例子中的字符串仅有 1 行。更详细的 docstring 应该以概述目的一行开头，接下来是一个空白行，然后是进一步的细节描述。例如，我们可以用如下方式更清楚地记录函数 scale 的信息：

```
def scale(data, factor):
    """ Multiply all entries of numeric data list by the given factor.

    data      an instance of any mutable sequence type (such as a list)
              containing numeric elements

    factor    a number that serves as the multiplicative factor for scaling
    """
    for j in range(len(data)):
        data[j] *= factor
```

docstring 作为模块、功能或者类的声明的一个域进行存储。它可以作文档用，并且可以用多种方式检索。例如，在 Python 解释器中，用命令 help(x) 会生成与标识对象 x 关联的文档 docstring。还有一个名叫 pydoc 的外部工具，该工具是 Python 发行的，可以用于生成文本或网页格式的正式文档。可在网站 <http://www.python.org/dev/peps/pep-0257/> 中得到有用的 docstring 书写指南。

在本书中，我们将在篇幅允许的情况下加上 docstring。省略的 docstring 可以在网络版的源代码中找到。

2.2.4 测试和调试

测试是通过实验检验程序正确性的过程，调试是跟踪程序的执行并在其中发现错误的过程。在程序开发中，测试和调试通常是最耗时的一项活动。

测试

详细的测试计划是编写程序最重要的部分。用所有可能的输入检验程序的正确性通常是不可行的，所以我们应该用有代表性的输入子集来运行程序。最起码我们应该确保类的每个方法都至少被执行一次（方法覆盖）。更好的是，程序中的每个代码语句应该至少被执行一次（语句覆盖）。

在特殊情况（special cases）的输入下，程序往往会失败。需要仔细确认和测试这些情况。例如，当测试一个对整数序列排序的方法（即 sort）时，我们应该考虑以下的输入：

- 序列具有零长度（没有元素）。
- 序列有一个元素。
- 序列中的所有元素是相同的。
- 序列已排序。
- 序列已反向排序。

除了对于程序而言特殊的输入以外，我们也应该考虑使用该程序结构的特殊情况。例

如，如果用一个 Python 列表存储数据，我们应该确保诸如添加或删除列表的开头或末尾的边界情况都可以正确处理。

手工测试是必不可少的，用大量随机生成的输入测试也是有优势的。Python 中的随机模块为生成随机数或随机集合的顺序提供了几种方法。

程序类和函数之间的依赖关系形成层次结构。也就是说，在层次结构中，如果组件 A 依赖于组件 B，比如函数 A 调用函数 B，或者函数 A 依赖于一个参数，该参数是类 B 中的实例，就称组件 A 高于组件 B。这里有两种主要的测试策略，自顶向下（top-down）和自底向上（bottom-up），它们的不同之处在于测试组件的顺序不同。

自上而下的测试从层次结构的顶部向底部进行。它通常用于连接存根（stubbing），一种用桩函数（stub）代替了底层组件的启动技术，桩函数是一种模拟原函数组件的替换技术。例如，如果函数 A 调用函数 B 获取文件的第一行，当测试 A 时，我们可以用返回固定字符串的桩函数代替 B。

自下而上的测试从低级组件向更高级组件进行。例如，首先测试不调用其他函数的底层函数，其次测试只调用底层函数的函数，等等。相似的，一个不依赖于其他类的类可以在依赖前者的其他类之前被测试。常将这种测试的形式称为单元测试（unit testing），在大型软件项目的孤立状态下测试特定组件的功能。如果使用得当，这种策略能够更好地把错误的起因与被测试的组件隔离开来，因为该组件依赖的低级组件已经被充分测试过了。

Python 为自动测试提供了几种支持形式。当函数或类定义在一个模块中时，该模块的测试可以被嵌入同一个文件中。1.11 节中描述了这样做的机制。当 Python 直接调用该模块，而不是该模块用作大型软件项目的输入时，在形式的条件结构中被屏蔽的代码

```
if __name__ == '__main__':
    # perform tests...
```

将被执行。在这样一个结构中来测试该模块中函数的功能和特别规定的类是很常见的。

对于单元测试自动化，Python 的 unittest 模块提供了更强大的支持。这个框架允许将单个测试用例分组到更大的测试套件中，并为执行这些套件提供支持，并报告或分析测试结果。为了维护软件，使用回归测试（regression testing），即通过对所有先前测试的重新执行来确保对软件的更改不会在先前测试的组件中引入新的错误。

调试

最简单的调试技术包括使用打印语句（print statement）来跟踪程序执行过程中变量的值。这种方法的一个问题是，最终需要删除或注释掉打印语句，因为最终发布软件时不能执行这些语句。

一种更好的方法是用调试器（debugger）运行程序。调试器是一个专门用于控制和监视程序执行的环境。调试器提供的基本功能是在代码中插入断点（breakpoint）。当在调试器中执行时，程序在每个断点处中止。当程序中止时，可以检查变量当前的值。

标准的 Python 程序包括一个 pdb 模块，该模块直接在解释器中提供调试支持。Python 的大多数集成开发环境 IDE，比如 IDLE，用图形用户界面提供调试环境。

2.3 类定义

类是面向对象程序设计中抽象的主要方法。在 Python 中，类的实例代表了每个数据块。类以及实现它的所有实例给成员函数（也称方法（methods））提供了一系列的行为。类也是

其实例的蓝图，每个实例通过属性（attributes）（也称域（field）、实例变量或数据成员）的确定状态信息。

2.3.1 例子：CreditCard 类

作为第一个例子，我们提供了一个基于图 2-3 和 2.2.1 节中介绍的设计 CreditCard 类的实现方法。CreditCard 类定义的实例为传统的信用卡提供了一个简单的模型。实例已经确定了关于客户、银行、账户、信用额度和余额信息。该类会根据消费额度限制支付，但不收取利息或滞纳金（我们将在 2.4.1 节中再讨论这个主题）。

我们的代码开始于代码段 2-1，并在代码段 2-2 中继续。结构以关键词 class 开始，接着是类的名字和一个冒号，然后是一块作为类主体的缩进代码。主体包括所有类的方法的定义。用 1.5 节中介绍的技术把这些方法定义为函数，并用到一个名为 self 的特殊参数，该参数用来识别调用成员的特定实例。

self 标识符

在 Python 中，self 标识符扮演了一个重要的角色。在 CreditCard 类的语境下，可以有很多不同的信用卡实例，而且每个都必须保持自己的余额、信用额度等。因此，每个实例都存储自己的实例变量，以反映其当前状态。

在语句构成上，self 确定了调用方法的实例。例如，假设类的用户有一个变量 my_card，它就确定了 CreditCard 类的一个实例。当用户调用 my_card.get_balance() 时，self 标识符用 get_balance 方法引用被调用者名为 my_card 的卡。self._balance 表达式引用一个名为 _balance 的实例变量，存储为特定信用卡状态的一部分。

代码段 2-1 CreditCard 类定义的开始部分（下接代码段 2-2）

```

1 class CreditCard:
2     """A consumer credit card."""
3
4     def __init__(self, customer, bank, acnt, limit):
5         """Create a new credit card instance.
6
7         The initial balance is zero.
8
9         customer  the name of the customer (e.g., 'John Bowman')
10        bank      the name of the bank (e.g., 'California Savings')
11        acnt      the account identifier (e.g., '5391 0375 9387 5309')
12        limit      credit limit (measured in dollars)
13
14        self._customer = customer
15        self._bank = bank
16        self._account = acnt
17        self._limit = limit
18        self._balance = 0
19
20    def get_customer(self):
21        """Return name of the customer."""
22        return self._customer
23
24    def get_bank(self):
25        """Return the bank's name."""
26        return self._bank
27
28    def get_account(self):
29        """Return the card identifying number (typically stored as a string)."""

```

```

30     return self._account
31
32 def get_limit(self):
33     """Return current credit limit."""
34     return self._limit
35
36 def get_balance(self):
37     """Return current balance."""
38     return self._balance

```

代码段 2-2 CreditCard 类定义的结尾（接代码段 2-1）。方法在类定义中都是缩简的

```

39 def charge(self, price):
40     """Charge given price to the card, assuming sufficient credit limit.
41
42     Return True if charge was processed; False if charge was denied.
43     """
44     if price + self._balance > self._limit:      # if charge would exceed limit,
45         return False                            # cannot accept charge
46     else:
47         self._balance += price
48         return True
49
50 def make_payment(self, amount):
51     """Process customer payment that reduces balance."""
52     self._balance -= amount

```

我们可以看到调用者使用方法签名与类内部定义声明使用方法签名之间的差异。例如，从用户的角度来看，我们知道 `get_balance` 方法不带参数，但在类定义中，`self` 是一个明确的参数。同样，在类中声明 `charge` 方法有两个参数（`self` 和 `price`），但是这个方法调用时只使用一个参数，例如 `my_card.charge(200)`。解释器在调用这些函数时自动将调用对应函数的实例绑定为 `self` 参数。

构造函数

用户可以用类似于下面的语法创建 `CreditCard` 类的实例

```
cc = CreditCard('John Doe', '1st Bank', '5391 0375 9387 5309', 1000)
```

其中，名为 `_init_` 的方法是类的构造函数（constructor）。它最主要的责任是用适当的实例变量建立一个新创建的 `CreditCard` 类对象。就 `CreditCard` 类来说，每个对象保存 5 个实例变量，我们将其命名为 `_customer`、`_bank`、`_account`、`_limit` 和 `_balance`。这 5 个变量中前 4 个的初始值是由明确的参数提供的，这些参数是在实例化信用卡时由用户发送的，并在构造函数的主体中给这些参数赋值。比如，`self._customer = customer`，把参数 `customer` 的值赋值给实例变量 `self._customer`。注意：因为等号右侧的 `customer` 没有限定，所以它指的是本地命名空间中的参数。

封装

2.2.3 节中所描述的惯例，在数据成员名称中的前加下划线，比如 `_balance`，表明它被设计为非公有的（nonpublic）。类的用户不应该直接访问这样的成员。

通常，我们将所有数据成员视为非公有的。这使我们能够更好地对所有实例执行一致的状态。我们可以提供类似于 `get_balance` 的访问函数，以提供拥有只读访问特性的类的用户。如果希望允许用户改变状态，我们可以提供适当的更新方法。在数据机构中，封装内部表达的方式允许我们更加灵活地设计类的工作方式，这或许能提高数据结构的效率。

附加方法

我们的类中最有趣的行为是收款和付款。收款功能通常会在信用卡余额中增加所收费

用，以反映顾客提到的购买价格。然而，在收取费用前，我们的实现方法要验证新的消费不会导致余额超过信用额度。付款费用反映了客户给银行支付给定的款项，从而减少信用卡中的余额。我们注意到，在 `self._balance -= amount` 命令中 `self._balance` 的语句由 `self` 标识符做了限定，因为它代表了卡的实例变量，而没有被限定的 `amount` 表示局部参数。

错误检查

`CreditCard` 类的实现方法不是特别健壮。首先，我们注意到对于收款和付款，没有明确地检查参数的类型，也没有给构造函数任何参数。如果用户创建了一个类似于 `visa.charge('candy')` 的调用，当企图在余额中添加参数时，代码可能会崩溃。如果这个类广泛地用于图书馆中，在面对这样的滥用（见 1.7 节）时，我们可能会用更多严谨的技术来抛出 `TypeError` 错误。

除了明显的类型错误，我们的实现方法可能会受到逻辑错误的影响。例如，如果允许用户收取一个类似于 `visa.charge(-300)` 的负的价格，这将导致用户的余额变少。这是可以不通过支付来减少余额的一个漏洞。当然，如果模拟信用卡收到顾客给商家的退货时，这也会被视为合法的情况。我们将在本章末的练习中用 `CreditCard` 类讨论一些这样的问题。

测试类

在代码段 2-3 中，我们演示了 `CreditCard` 类的一些基本用法，在 `wallet` 列中插入 3 张卡。我们循环地进行收款和付款，并使用各种访问函数将结果打印到控制台。

这些测试封闭在 `if __name__ == '__main__':` 条件中，这样它们可以通过类的定义嵌入源代码中。使用 2.2.4 节中的术语，这些测试提供方法覆盖，每个方法至少被调用一次，但是这些测试不提供语句覆盖，因为有信用额度，所以这里不会有任何一种情况中的收款被拒绝。这种测试比较落后，必须手动地审核给定测试的输出结果，以确定是否该类表现得如我们所预期的一样。Python 有更正式的测试工具（见 2.2.4 节中讨论的 `unittest` 模块），这样得到的值可以与预测结果自动地比较，只有当检测到错误时才产生输出。

代码段 2-3 测试 `CreditCard` 类

```

53 if __name__ == '__main__':
54     wallet = []
55     wallet.append(CreditCard('John Bowman', 'California Savings',
56                             '5391 0375 9387 5309', 2500))
57     wallet.append(CreditCard('John Bowman', 'California Federal',
58                             '3485 0399 3395 1954', 3500))
59     wallet.append(CreditCard('John Bowman', 'California Finance',
60                             '5391 0375 9387 5309', 5000))
61
62     for val in range(1, 17):
63         wallet[0].charge(val)
64         wallet[1].charge(2*val)
65         wallet[2].charge(3*val)
66
67     for c in range(3):
68         print('Customer =', wallet[c].get_customer())
69         print('Bank =', wallet[c].get_bank())
70         print('Account =', wallet[c].get_account())
71         print('Limit =', wallet[c].get_limit())
72         print('Balance =', wallet[c].get_balance())
73         while wallet[c].get_balance() > 100:
74             wallet[c].make_payment(100)
75             print('New balance =', wallet[c].get_balance())
76             print()

```

2.3.2 运算符重载和 Python 的特殊方法

Python 的内置类为许多操作提供了自然的语义。比如，`a + b` 语句可以调用数值类型语句，也可以连接序列类型。当定义一个新类时，我们必须考虑到当 `a` 或者 `b` 是类中的实例时是否应该定义类似于 `a + b` 的语句。

默认情况下，对于新的类来说，“+”操作符是未定义的。然而，类的作者可通过操作符重载（operator overloading）技术来定义它。这个定义可通过一个特殊的命名方法来实现。特别的是，名为 `__add__` 的方法重载 + 操作符，`__add__` 用右边的操作作为参数并返回表达式的结果。也就是说，`a + b` 语句，被转换为一个调用 `a.__add__(b)` 对象的方法。类似地，特殊命名的方法存在其他操作符中。表 2-1 提供了与这一方法类似的完整列表。

表 2-1 用 Python 特殊方法实现的重载操作

常见的语法	特别方法的形式
<code>a + b</code>	<code>a.__add__(b);</code> 或者 <code>b.__radd__(a)</code>
<code>a - b</code>	<code>a.__sub__(b);</code> 或者 <code>b.__rsub__(a)</code>
<code>a * b</code>	<code>a.__mul__(b);</code> 或者 <code>b.__rmul__(a)</code>
<code>a / b</code>	<code>a.__truediv__(b);</code> 或者 <code>b.__rtruediv__(a)</code>
<code>a // b</code>	<code>a.__floordiv__(b);</code> 或者 <code>b.__rfloordiv__(a)</code>
<code>a % b</code>	<code>a.__mod__(b);</code> 或者 <code>b.__rmod__(a)</code>
<code>a ** b</code>	<code>a.__pow__(b);</code> 或者 <code>b.__rpow__(a)</code>
<code>a << b</code>	<code>a.__lshift__(b);</code> 或者 <code>b.__rlshift__(a)</code>
<code>a >> b</code>	<code>a.__rshift__(b);</code> 或者 <code>b.__rrshift__(a)</code>
<code>a & b</code>	<code>a.__and__(b);</code> 或者 <code>b.__rand__(a)</code>
<code>a ^ b</code>	<code>a.__xor__(b);</code> 或者 <code>b.__rxor__(a)</code>
<code>a b</code>	<code>a.__or__(b);</code> 或者 <code>b.__ror__(a)</code>
<code>a += b</code>	<code>a.__iadd__(b)</code>
<code>a -= b</code>	<code>a.__isub__(b)</code>
<code>a *= b</code>	<code>a.__imul__(b)</code>
...	...
<code>+ a</code>	<code>a.__pos__(b)</code>
<code>- a</code>	<code>a.__neg__(b)</code>
<code>~ a</code>	<code>a.__invert__(b)</code>
<code>abs(a)</code>	<code>a.__abs__(b)</code>
<code>a < b</code>	<code>a.__lt__(b)</code>
<code>a <= b</code>	<code>a.__le__(b)</code>
<code>a > b</code>	<code>a.__gt__(b)</code>
<code>a >= b</code>	<code>a.__ge__(b)</code>
<code>a == b</code>	<code>a.__eq__(b)</code>
<code>a != b</code>	<code>a.__ne__(b)</code>
<code>in a</code>	<code>a.__contains__(v)</code>
<code>a [k]</code>	<code>a.__getitem__(k)</code>
<code>a [k] = v</code>	<code>a.__setitem__(k, v)</code>
<code>del a [k]</code>	<code>a.__delitem__(k)</code>
<code>a(arg1, arg2, ...)</code>	<code>a.__call__(arg1, arg2, ...)</code>
<code>len(a)</code>	<code>a.__len__()</code>

(续)

常见的语法	特别方法的形式
hash(a)	a.__hash__()
iter(a)	a.__iter__()
next(a)	a.__next__()
bool(a)	a.__bool__()
float(a)	a.__float__()
int(a)	a.__int__()
repr(a)	a.__repr__()
reversed(a)	a.__reversed__()
str(a)	a.__str__()

像 `3 *'love me'` 一样，当一个二元操作符应用于两个不同类型的实例中时，Python 对根据左操作数的类进行判断。在这个例子中，对于使用 `__mul__` 方法把字符串与实例相乘，可以通过检查 `int` 类是否提供了相应的定义。然而，如果这个类没有实现这一行为，Python 就会以一种名为 `__rmul__`（即“右乘”）的特殊方法来检查右操作数的类的定义。该方法为新用户定义的类提供了一个支持包含已存在类（所给的已存在的类可能没有定义引用该新类的行为）的实例的混合操作的方法。`__mul__` 和 `__rmul__` 的区别也允许类根据情况定义不同的语义，如操作数在矩阵乘法中就是不可交换的（即 `A * x` 可能与 `x * A` 不同）。

非运算符重载

当使用用户自定义的类时，除了传统的操作符重载，Python 依靠特殊的命名方法来控制各种功能的行为。例如，`str(foo)` 语句，是 `string` 类的构造函数的一个调用。如果参数是用户定义的类的实例，`string` 类的原作者当然不知道应该如何根据这个实例构造字符串。所以字符串构造函数调用一个专门的命名方法，`foo.__str__()`，它必须返回一个恰当的字符串表示形式。

类似的方法也被用于通过一个用户自定义类来构造 `int`、`float` 或 `bool` 类型。将一个用户自定义类转换为一个 `Boolean` 值尤为重要，因为即使当 `foo` 不是一个 `Boolean` 值（见 1.4.1 节）时也可以使用 `if foo:` 语句。对于用户自定义的类，用专门的方法 `foo.__bool__()` 返回对应的真值。

几个其他的顶层功能依赖于调用特殊的命名方法。例如，调用顶层的 `len` 函数是确定一个容器类型大小的标准方法。注意：调用 `len(foo)` 不是传统的用点运算符的方法调用语法。在一个用户已定义的类的情况下，顶层的 `len` 函数依赖于调用该类特别的命名方法 `__len__`。也就是说，`len(foo)` 的返回值是通过调用 `foo.__len__()` 得到的。当作用于数据结构时，我们经常定义 `__len__` 方法来返回一个结构的大小。

隐式的方法

作为一般规则，如果在用户已定义的类中没有实现特定的特殊方法，则依赖于该方法的标准语法将引发异常。例如，用户自定义类未定义 `__add__` 或者 `__radd__` 方法，则计算自定义类的实例相加的语句 `a + b` 将会引发异常。

然而，当缺乏特殊方法时，有一些操作符已经由 Python 提供了默认定义，也有一些操作符的定义来源于其他定义。例如，支持 `if foo:` 语句的 `__bool__` 方法有默认语义，以至于除了 `None` 以外的每个对象的值都为 `True`。然而，对于容器类型，通常定义 `__len__` 方法返

回容器的大小。如果这种方式存在，对于长度不为 0 的实例，`bool(foo)` 的值默认情况下为 `True`，对于长度为 0 的实例，值默认情况下为 `False`，允许用类似于 `if waitlist:` 的语句测试是否在等待队列中由一个或多个条目。

在 2.3.4 节中，我们将讨论 Python 通过特殊方法 `__iter__` 为集合提供迭代器的机制。也就是说，如果一个容器类实现了 `__len__` 和 `__getitem__` 方法，则它可以自动提供一个默认迭代器（用我们在 2.3.4 节中讨论的方法）。此外，一旦定义了迭代器，就提供了 `__contains__` 的默认功能。

在 1.3 节中，我们指出了表达式 `a is b` 和表达式 `a == b` 之间的区别，前者评估标识符 `a` 和 `b` 是否为同一对象的别名，后者测试是否两个标识符引用等价值的概念。“等价”的概念依赖于类的上下文，并用 `__eq__` 方法定义语义。然而，如果没有实现 `__eq__` 方法，语句 `a == b` 和 `a is b` 语义是等价的，即一个实例只和其自身是等价的，和其他实例都不相等。

我们也应该注意到，Python 并没有自动提供一些我们认为自然而然的表达式。例如，`__eq__` 方法支持 `a == b` 语句，但该方法不影响 `a != b` 语句的值（该值通过 `__ne__` 方法计算，通常返回 `not (a == b)` 作为结果）。同样，提供 `__lt__` 方法支持 `a < b` 语句，并且间接支持 `b > a` 语句，但是提供的 `__lt__` 和 `__eq__` 都没有 `a <= b` 的语义。

2.3.3 例子：多维向量类

为了演示通过特殊方法使用运算符重载，我们给出一个 `Vector` 类的实现方法，代表了一个多维空间中向量的坐标。例如，在三维空间中，也许我们希望用坐标 $\langle 5, -2, 3 \rangle$ 代表一个向量。虽然直接使用 Python 列表代表那些坐标可能更有吸引力，但是列表不能为几何向量提供适当的抽象。特殊的是，如果使用列表，表达式 $[5, -2, 3] + [1, 4, 2]$ 的结果是 $[5, -2, 3, 1, 4, 2]$ 。当用向量工作时，如果 $u = \langle 5, -2, 3 \rangle$ 并且 $v = \langle 1, 4, 2 \rangle$ ，我们希望用表达式 $u + v$ 来返回一个坐标为 $\langle 6, 2, 5 \rangle$ 三维向量。

因此，我们在代码段 2-4 中定义一个 `Vector` 类，它为几何向量的概念提供了一个更好的抽象。在内部，我们的向量依赖列表名为 `_coords` 的实例，作为它的存储机制。通过保持内部列表的封装，我们可以为类中的实例执行所请求的公共接口。示例如下：

```
v = Vector(5)                      # construct five-dimensional <0, 0, 0, 0, 0>
v[1] = 23                           # <0, 23, 0, 0, 0> (based on use of __setitem__)
v[-1] = 45                          # <0, 23, 0, 0, 45> (also via __setitem__)
print(v[4])                         # print 45 (via __getitem__)
u = v + v                           # <0, 46, 0, 0, 90> (via __add__)
print(u)                           # print <0, 46, 0, 0, 90>
total = 0
for entry in v:                     # implicit iteration via __len__ and __getitem__
    total += entry
```

代码段 2-4 一个简单 `Vector` 类的定义

```
1 class Vector:
2     """Represent a vector in a multidimensional space."""
3
4     def __init__(self, d):
5         """Create d-dimensional vector of zeros."""
6         self._coords = [0] * d
7
8     def __len__(self):
9         """Return the dimension of the vector."""
10        return len(self._coords)
```

```

11
12 def __getitem__(self, j):
13     """Return jth coordinate of vector."""
14     return self._coords[j]
15
16 def __setitem__(self, j, val):
17     """Set jth coordinate of vector to given value."""
18     self._coords[j] = val
19
20 def __add__(self, other):
21     """Return sum of two vectors."""
22     if len(self) != len(other):           # relies on __len__ method
23         raise ValueError('dimensions must agree')
24     result = Vector(len(self))          # start with vector of zeros
25     for j in range(len(self)):
26         result[j] = self[j] + other[j]
27     return result
28
29 def __eq__(self, other):
30     """Return True if vector has same coordinates as other."""
31     return self._coords == other._coords
32
33 def __ne__(self, other):
34     """Return True if vector differs from other."""
35     return not self == other            # rely on existing __eq__ definition
36
37 def __str__(self):
38     """Produce string representation of vector."""
39     return '<' + str(self._coords)[1:-1] + '>' # adapt list representation

```

很多接口可以通过调用和内部坐标列表类似的方法实现。然而，`__add__` 的实现方法却不同。假设两个操作数是长度相同的向量，该方法创建了一个新的向量，并将新向量的坐标置为各自操作数对应分量元素的和。

请注意代码段 2-4 中该方法的定义很有趣，该定义自动支持 $\mathbf{u} = \mathbf{v} + [5, 3, 10, -2, 1]$ 语法，并产生一个新的向量，该向量的各个元素是第一个向量和列表实例对应位置元素之和。这是 Python 多态性 (polymorphism) 的结果。从字面上看，“多态”的意思是“许多形式”。虽然它容易使我们想到 `__add__` 方法中的 `other` 参数是一个 `Vector` 的实例，但我们并没这样声明它。在内部，我们依赖于参数 `other` 的唯一行为是它支持 `len(other)` 并且可以访问 `other[j]`。因此，当右边的操作数是一个数字 (匹配的长度) 列表时，代码依然可以执行。

2.3.4 迭代器

在数据结构的设计中，迭代器是一个重要的概念。在 1.8 节中，我们介绍了 Python 迭代器的机制。简而言之，集合的迭代器 (iterator) 提供了一个关键行为：它支持一个名为 `__next__` 的特殊方法，如果集合有下一个元素，该方法返回该元素，否则产生一个 `StopIteration` 异常来表明没有下一个元素。

幸运的是，很少需要直接实现迭代器类。我们的首选方法是使用生成器 (generator) 语法 (已在 1.8 节中描述了)，它自动地产生一个已生成值的迭代器。

Python 也为实现 `__len__` 和 `__getitem__` 的类提供了一个自动的迭代器。为了提供一个低级迭代器的例子，代码段 2-5 演示了这种迭代器类可用于任何支持 `__len__` 和 `__getitem__` 的集合的处理。该类可被实例化为 `SequenceIterator(data)`。它通过保存在内部的数据序列引用来操作该序列以及当前的索引。每次调用 `__next__` 时，索引递增，直到序列结束。

代码段 2-5 一个支持任何序列类型的迭代器类

```

1 class Sequencelerator:
2     """An iterator for any of Python's sequence types."""
3
4     def __init__(self, sequence):
5         """Create an iterator for the given sequence."""
6         self._seq = sequence           # keep a reference to the underlying data
7         self._k = -1                   # will increment to 0 on first call to next
8
9     def __next__(self):
10        """Return the next element, or else raise StopIteration error."""
11        self._k += 1                 # advance to next index
12        if self._k < len(self._seq):
13            return(self._seq[self._k]) # return the data element
14        else:
15            raise StopIteration()    # there are no more elements
16
17     def __iter__(self):
18        """By convention, an iterator must return itself as an iterator."""
19        return self

```

2.3.5 例子：Range 类

作为本节中的最后一个例子，我们实现一个类来模拟 Python 的内置 `range` 类。在介绍这个类之前，我们先讨论内置版本的历史。在发布 Python 3 之前，`range` 作为一个函数来实现，并且用特定范围内的元素返回一个列表实例。例如，`range(2, 10, 2)` 返回列表 `[2, 4, 6, 8]`。然而，该函数的典型用法是支持类似于 `for k in range(10000000)` 的循环语法。不幸的是，这会引起一个数字范围列表的实例化和初始化，在时间和内存的使用上都造成了不必要的浪费。

在 Python 3 中，`range` 的机制是完全不同的（公平地说，这种“新”方法在 Python 2 中也存在，但是名为 `xrange`）。它使用了一种被称为惰性求值（lazy evaluation）的策略。与其创建一个新的列表实例，不如使用 `range`，它是一个类，可以有效地表示所需的元素范围，而不必在内存中明确地存储它们。为了更好地探讨内置 `range` 类，我们建议你创建一个类似于 `r = range(8, 140, 5)` 的实例。其结果是一个相对轻量级的对象，一个只有几个行为的 `range` 类的实例。`len(r)` 语句将报告给定范围中元素的数量（在我们的例子中是 27）。`range` 也支持 `__getitem__` 方法，`r[15]` 表达式返回了 `range` 中的第 16 个元素（`r[0]` 是第一个元素）。因为这个类支持 `__len__` 和 `__getitem__`，所以它自动支持迭代（见 2.3.4 节），这就是为什么可以通过 `range` 执行一个 `for` 循环。

对此，我们准备展示一个自定义的类的版本。代码段 2-6 提供了一个类，我们将其命名为 `Range`（以明确区分它与内置的 `range`）。这一实现的最大挑战是当构建 `range` 时通过调用者发送给定的参数时正确地计算属于 `range` 的元素个数。通过计算构造函数中的值，并存储为 `self._length`，把该值从 `__len__` 方法中返回就很容易了。为了正确地实现对 `__getitem__(k)` 的调用，我们只需把 `range` 的初始值加上 `k` 乘以步长（即，当 `k = 0`，我们返回初始值）。这几个值得在代码段中检查的细节：

- 当讨论一个可工作的 `range` 函数版本时，为了正确地支持可选参数，我们使用了 1.5.1 节中描述的技术。
- 我们通过 `max(0, (stop - start + step - 1)//step)` 计算元素的个数，对于正数和负数的步长该公式都需要测试。

- 在计算结果之前，`__getitem__` 方法可通过将 $-k$ 转换为 $\text{len}(\text{self}) - k$ 以正确地支持负数下标。

代码段 2-6 自定义的 Range 类的实现

```

1 class Range:
2     """A class that mimics the built-in range class."""
3
4     def __init__(self, start, stop=None, step=1):
5         """Initialize a Range instance.
6
7         Semantics is similar to built-in range class.
8         """
9
10        if step == 0:
11            raise ValueError('step cannot be 0')
12
13        if stop is None:                      # special case of range(n)
14            start, stop = 0, start           # should be treated as if range(0,n)
15
16        # calculate the effective length once
17        self._length = max(0, (stop - start + step - 1) // step)
18
19        # need knowledge of start and step (but not stop) to support __getitem__
20        self._start = start
21        self._step = step
22
23    def __len__(self):
24        """Return number of entries in the range."""
25        return self._length
26
27    def __getitem__(self, k):
28        """Return entry at index k (using standard interpretation if negative)."""
29        if k < 0:
30            k += len(self)                 # attempt to convert negative index
31
32        if not 0 <= k < self._length:
33            raise IndexError('index out of range')
34
35        return self._start + k * self._step

```

2.4 继承

组织各种软件包的结构组件的自然方法是，在一个分层（hierarchical）的方式中，在水平层次上把类似的抽象定义组合在一起，下层的组件更加具体，上层的组件更加通用。图 2-4 展示了这样一个层次的例子。用数学符号表示，一套房子是一个建筑物的子集（subset），但它是一个牧场的超集（superset）。层次之间的对应关系通常被称为“is-a”的关系，就像房子是建筑，平房是房子。

在软件开发中，层次设计是非常有用的，在最通用的层次上可以把共同的功能分组，从而促进代码的重用，进而将行为间的差别视为通用情况的扩展。在面向对象的编程中，模块化和层次化组织的机制是一种称为继承（inheritance）的技术。这个技术允许基于一个现有的类作为起点定义新的类。在面向对象的术语中，通常描述现有的类为基类（base class）、父类（parent class）或者超类（superclass），而称新定义的类为子类（subclass 或者 child class）。

有两种方式可以让子类有别于父类。子类可以通过提供一个新的覆盖（override）现有

方法的实现方法特化 (specialize) 一个现有的行为。子类也可以通过提供一些全新的方法扩展 (extend) 其父类。

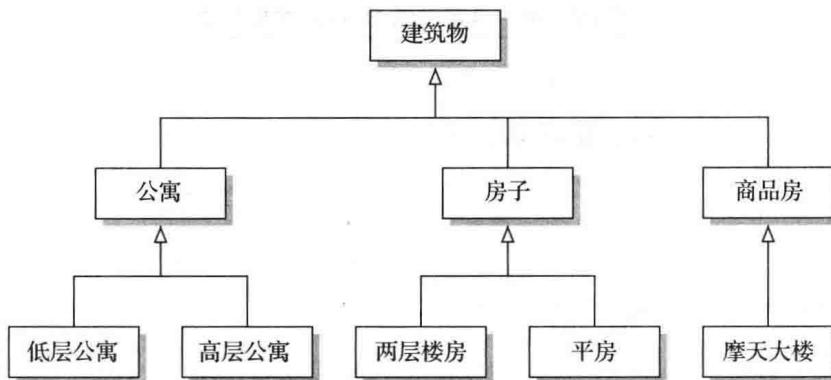


图 2-4 一个涉及建筑物的“is-a”层次图的例子

Python 的异常层次结构

富有继承层次的另一个例子是在 Python 中组织各种异常类型。我们在 1.7 节中介绍了许多类，但没有讨论它们之间的相互关系。图 2-5 说明了该层次结构中的一小部分。BaseException 类是整个层次结构的根，而更具体的 Exception 类包括了大部分我们已经讨论过的错误类型。程序设计者可以自由定义特殊的异常类，以表示在应用程序的上下文中可能出现的错误。应该声明这些用户自定义的异常类型为 Exception 的子类。

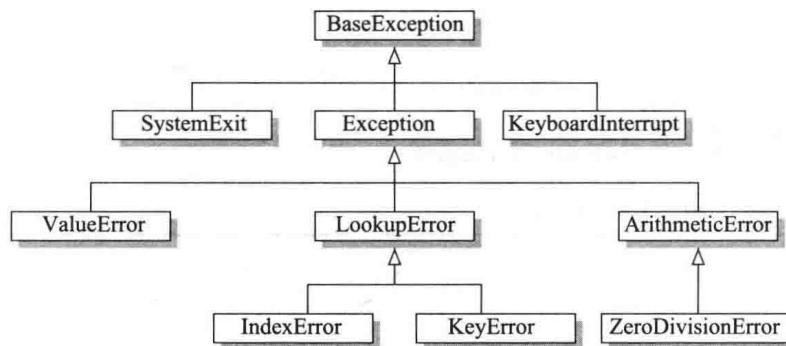


图 2-5 Python 异常类型层次的一部分

2.4.1 扩展 CreditCard 类

为了表示 Python 中的层次机制，我们再来看 2.3 节中的信用卡类，实现子集 Predatory CreditCard。因为没有更好的名字，所以我们将其命名为 PredatoryCreditCard。新类和原始的类将有两方面的不同：①当尝试收费由于超过信用卡额度被拒绝时，将会收取 5 美元的费用；②将有一个对未清余额按月收取利息的机制，即基于构造函数的一个参数年利率 (Annual Percentage Rate, APR)。

在实现这一目标时，我们展示了特化和扩展技术。在进行无效的收费时，我们覆盖了现有的收费方法，并由此特化它以提供新的功能（虽然新的版本利用了被覆盖版本的调用）。为了给收取利息提供支持，我们用名为 process_month 的新方法扩展该类。

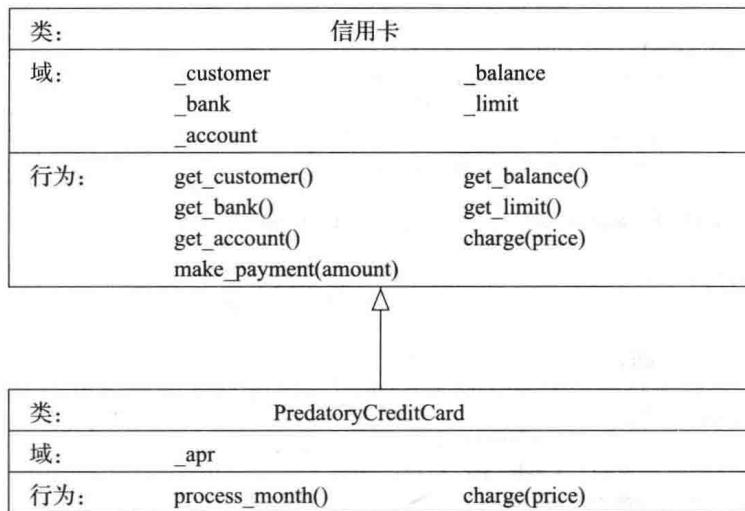


图 2-6 继承关系图

图 2-6 描述了我们在设计新 `PredatoryCreditCard` 类中使用的继承关系，代码段 2-7 给出了一个完整的 Python 类的实现。

为了表明新类从现有的 `CreditCard` 类中继承，我们的定义从 `class PredatoryCreditCard(CreditCard)` 语法开始。新类的主体提供了三个成员函数：`__init__`、`charge` 和 `process_month`。`__init__` 构造函数的作用和 `CreditCard` 构造函数非常类似，除新的类之外，还有一个额外的参数来指定年利率。新构造函数的主体依赖调用继承的构造函数来执行大部分的初始化处理（事实上，除了记录百分比之外的一切）。调用继承构造函数的机制依赖于 `super()` 语法。具体来讲，即第 15 行命令

```
super().__init__(customer, bank, acnt, limit)
```

调用从 `CreditCard` 父类继承的 `__init__` 方法。值得注意的是，这个方法只接受 4 个参数。我们在名为 `_apr` 的新域中记录 APR 的值。

同样，`PredatoryCreditCard` 类提供了一个新收费策略的实现方法，该方法重写了继承的方法。然而，新方法的实现取决于对继承方法的调用，用第 24 行中的语句 `super().charge(price)`。调用函数的返回值表明是否收费成功。

代码段 2-7 评估利息和费用的 CreditCard 子类

```

1 class PredatoryCreditCard(CreditCard):
2     """An extension to CreditCard that compounds interest and fees."""
3
4     def __init__(self, customer, bank, acnt, limit, apr):
5         """Create a new predatory credit card instance.
6
7         The initial balance is zero.
8
9         customer  the name of the customer (e.g., 'John Bowman')
10        bank      the name of the bank (e.g., 'California Savings')
11        acnt      the account identifier (e.g., '5391 0375 9387 5309')
12        limit      credit limit (measured in dollars)
13        apr       annual percentage rate (e.g., 0.0825 for 8.25% APR)
14
15        super().__init__(customer, bank, acnt, limit)      # call super constructor
16        self._apr = apr

```

```

17
18 def charge(self, price):
19     """ Charge given price to the card, assuming sufficient credit limit.
20
21     Return True if charge was processed.
22     Return False and assess $5 fee if charge is denied.
23     """
24     success = super().charge(price)           # call inherited method
25     if not success:
26         self._balance += 5                  # assess penalty
27     return success                         # caller expects return value
28
29 def process_month(self):
30     """ Assess monthly interest on outstanding balance."""
31     if self._balance > 0:
32         # if positive balance, convert APR to monthly multiplicative factor
33         monthly_factor = pow(1 + self._apr, 1/12)
34         self._balance *= monthly_factor

```

我们检查返回值，以决定是否评估费用。相应的，我们返回该值给方法的调用者，这样可以使得新的收费方法与原来的方法有一个类似的外部接口。

`process_month` 方法是一种新行为，所以没有依赖继承的版本。在我们的模型中，这种方法应该每月由银行调用一次，来收取新的利息费用。实施这种方法最具挑战性的是确保我们已经有将年利率转换为月利率的知识。我们不能简单地将年利率除以 12 来得到月利率（这样太没道理，因为这将导致 APR 比实际的更高）。正确的计算方法是 $1 + \text{self._apr}$ 开十二次方，并用它作为乘法因子。例如，如果一个 APR 是 0.0825（代表 8.25%），我们计算 $\sqrt[12]{1.0825} \approx 1.006628$ ，因此每月收取 0.6628% 的利息。按照这种方式，每年 100 美元的债务一年将累计 8.25 美元的复利。

保护成员

`PredatoryCreditCard` 子类直接访问数据成员 `self._balance`，这个数据成员是由 `CreditCard` 父类建立的。按照约定，名字带下划线表示它是一个非公有成员，所以我们可能会问是否可以照这种方式访问它。虽然一般类的用户不会这样做，但是我们这里的子类与父类有些特权关系。一些面向对象的语言（如 Java, C++）指出了非公有成员的区别，即允许声明的受保护（`protected`）或私有（`private`）的访问模式。被声明为受保护的成员可以访问子类，但是不能访问一般的公有类；被声明为私有的成员既不能访问子类，也不能访问公有类。在这方面，如果它是受保护的（但不是私有的），我们就用 `_balance`。

Python 不支持正式的访问控制，但以一个下划线开头的名字都被看作受保护的，而以双下划线开头的名字（除了特殊的方法）是被看作私有的。在选择使用受保护的数据时，我们已经创建了一个依赖，在该依赖中，如果 `CreditCard` 类的作者改变了内部设计，`PredatoryCreditCard` 可能也会改变。要注意的是，我们可能在 `process_month` 方法中依赖公有的 `get_balance()` 方法来检索当前的余额。但是 `CreditCard` 类的设计不能为子类提供一个有效的方式来改变余额，除了直接操作数据成员。用 `charge` 方法来为余额增加费用和利息可能是很有吸引力的。然而，这种方法不允许余额超过客户的信用额度，但是如果有担保的话，银行可能会让利率超出信贷限额。如果重新设计原始的 `CreditCard` 类，我们可以添加一个非公有的方法 `_set_balance`，子类可以用该方法来改变余额而不直接访问数据成员 `_balance`。

2.4.2 数列的层次图

作为使用继承的第二个例子，我们将介绍迭代数列的类的层次。数列是指数字的序列，其中每个数字都依赖于一个或更多的前面的数字。例如，一个等差数列 (arithmetic progression) 通过给前一个数值增加一个固定常量来确定下一个数字，一个等比数列 (geometric progression) 通过前一个值乘以固定常量来确定下一个数字。在一般情况下，数列需要一个初始值，以及在一个或多个先前值的基础上确定新值的方法。

为了最大限度地提高代码的可重用性，我们给出了一个由通用基类产生的名为 Progression 类 (见图 2-7) 的分层。从技术上讲，Progression 类产生全为数字的数列：0, 1, 2, …然而，该类被设计为其他数列类型的基类，提供尽可能多的公共函数，并由此把子类的负担减至最小。

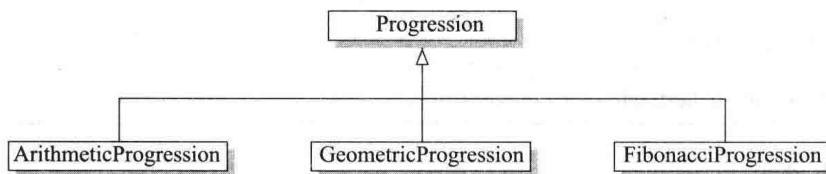


图 2-7 数列类的层级结构

代码段 2-8 提供了基本 Progression 类的实现方法。这个类的构造函数接受数列的起始值（默认为 0），并用该值初始化 `self._current` 数据成员。

Progression 类实现 Python 迭代器（见 2.3.4 节）的公约，即特殊的 `__next__` 和 `__iter__` 方法。如果类的用户创建了一个 `seq = Progression()` 的数列序列，`next(seq)` 的每次调用会返回数列中的下一个值。也可以使用 `for-loop` 的语法 `for value in seq:`，但是我们注意到，默认的数列被定义为无穷序列。

为了更好地从数列的核心逻辑中将迭代器公约的技术分离出来，我们的框架依靠一个名为 `_advance` 的非公有方法来更新 `self._current` 域的值。在默认的实现方法中，`_advance` 添加了一个当前值，但我们的目的是子类重写 `_advance` 方法，以提供不同的方法来计算下一个值。

为方便起见，Progression 类还提供了一个名为 `print_progression` 的实体方法，该方法显示了数列接下来的 `n` 个值。

代码段 2-8 一个通用数字数列类

```

1 class Progression:
2     """Iterator producing a generic progression.
3     Default iterator produces the whole numbers 0, 1, 2, ...
4     """
5
6     def __init__(self, start=0):
7         """Initialize current to the first value of the progression."""
8         self._current = start
9
10    def _advance(self):
11        """Update self._current to a new value.
12        This should be overridden by a subclass to customize progression.
13    """
14
15
  
```

```

16     By convention, if current is set to None, this designates the
17     end of a finite progression.
18     """
19     self._current += 1
20
21     def __next__(self):
22         """Return the next element, or else raise StopIteration error."""
23         if self._current is None:      # our convention to end a progression
24             raise StopIteration()
25         else:
26             answer = self._current    # record current value to return
27             self._advance()          # advance to prepare for next time
28             return answer            # return the answer
29
30     def __iter__(self):
31         """By convention, an iterator must return itself as an iterator."""
32         return self
33
34     def print_progression(self, n):
35         """Print next n values of the progression."""
36         print(' '.join(str(next(self)) for j in range(n)))

```

一个等差数列类

特殊数列的第一个例子是等差数列。数列默认逐步增加自身的值，等差数列通过给数列的每一项增加一个固定的常量来产生下一个值。例如，用一个初值为 0、增量为 4 的等差数列将产生序列 0, 4, 8, 12, …

代码段 2-9 介绍了 ArithmeticProgression 类的实现方法，该类以 Progression 类作为它的基类。新类的构造函数接受增量和初值两个参数，每个参数都有默认值。我们约定，ArithmeticProgression(4) 产生序列 0, 4, 8, 12, …，ArithmeticProgression(4, 1) 产生序列 1, 5, 9, 13, …。

代码段 2-9 一个产生等差数列的类

```

1  class ArithmeticProgression(Progression):      # inherit from Progression
2      """Iterator producing an arithmetic progression."""
3
4      def __init__(self, increment=1, start=0):
5          """Create a new arithmetic progression.
6
7          increment  the fixed constant to add to each term (default 1)
8          start      the first term of the progression (default 0)
9          """
10     super().__init__(start)                      # initialize base class
11     self._increment = increment
12
13     def _advance(self):                         # override inherited version
14         """Update current value by adding the fixed increment."""
15         self._current += self._increment

```

ArithmeticProgression 构造函数的主体调用超类的构造函数来初始化 `_current` 数据成员作为所需的初值，然后直接为等差数列建立新的 `_increment` 数据成员。实现中唯一遗留的细节是重写 `_advance` 方法以便给当前的值加上增量。

一个等比数列类

第二个特殊数列的例子是一个等比数列，其中每个值由固定常量乘以先前的值而产生，该固定常量被称为等比数列的基数。等比数列的初值通常为 1，而不是 0，因为任何因子乘以 0 其结果都是 0。举一个例子，一个以 2 为基数的等比数列为 1, 2, 4, 8, 16, …

代码段 2-10 介绍了 GeometricProgression 类的实现方法。构造函数以 2 作为默认基数，并用 1 作为默认的初值，但其中任意一个都可以使用可选参数。

代码段 2-10 一个产生等比数列的类

```

1 class GeometricProgression(Progression):          # inherit from Progression
2     """Iterator producing a geometric progression."""
3
4     def __init__(self, base=2, start=1):
5         """Create a new geometric progression.
6
7             base      the fixed constant multiplied to each term (default 2)
8             start     the first term of the progression (default 1)
9
10            """
11            super().__init__(start)
12            self._base = base
13
14    def _advance(self):                      # override inherited version
15        """Update current value by multiplying it by the base value."""
16        self._current *= self._base

```

一个斐波那契数列类

作为最后一个例子，我们介绍如何使用数列框架来产生一个斐波那契数列（Fibonacci progression）。我们在 1.8 节的“生成器”部分讨论过斐波纳契数列。斐波那契数列的每一个值是最近的两个值之和。为了产生序列，通常以 0 和 1 作为最前面的两个值，从而产生斐波那契数列：0, 1, 1, 2, 3, 5, 8, …一般而言，这样的数列可以从任意两个初值中生成。例如，如果从 4 和 6 开始，则产生的数列为 4, 6, 10, 16, 26, 42, …

在代码段 2-11 中，我们用数列框架来定义一个新的 FibonacciProgression 类。这个类与等差、等比数列有显著不同，因为我们不能独立地从当前值产生斐波那契数列的下一个值。我们必须得到两个最新的值。基础的 Progression 类已经提供了用以存储最新值的 `_current` 数据成员。FibonacciProgression 类则介绍了一个名为 `_prev` 的新成员来存储当前生成的值。

代码段 2-11 一个产生斐波那契数列的类

```

1 class FibonacciProgression(Progression):
2     """Iterator producing a generalized Fibonacci progression."""
3
4     def __init__(self, first=0, second=1):
5         """Create a new fibonacci progression.
6
7             first      the first term of the progression (default 0)
8             second    the second term of the progression (default 1)
9
10            """
11            super().__init__(first)           # start progression at first
12            self._prev = second - first      # fictitious value preceding the first
13
14    def _advance(self):
15        """Update current value by taking sum of previous two."""
16        self._prev, self._current = self._current, self._prev + self._current

```

先前存储的值和 `_advance` 的实现是直接相关的（我们使用了一个类似于 1.9 节中的同时赋值的方法）。然而，问题是如何在构造函数中初始化先前的值。需要提供第一个和第二个值作为构造函数的参数。第一个值被存储为 `_current`，这样它就变为第一个被访问的值。继续计算，一旦第一个值被访问，我们将通过赋值来设置新的当前值（第二个值将访问该

值), 等于第一值加上“先前的值”。通过 `(second - first)` 来初始化先前的值, 初始时将 `first + (second - first) = second` 设置为所需的当前值。

测试数列

为了完成演示, 代码段 2-12 为所有数列类提供了一个单元测试, 并在代码段 2-13 中显示了测试的输出。

代码段 2-12 我们数列类的单元测试

```
if __name__ == '__main__':
    print('Default progression:')
    Progression().print_progression(10)

    print('Arithmetic progression with increment 5:')
    ArithmeticProgression(5).print_progression(10)

    print('Arithmetic progression with increment 5 and start 2:')
    ArithmeticProgression(5, 2).print_progression(10)

    print('Geometric progression with default base:')
    GeometricProgression().print_progression(10)

    print('Geometric progression with base 3:')
    GeometricProgression(3).print_progression(10)

    print('Fibonacci progression with default start values:')
    FibonacciProgression().print_progression(10)

    print('Fibonacci progression with start values 4 and 6:')
    FibonacciProgression(4, 6).print_progression(10)
```

代码段 2-13 代码段 2-12 测试的输出

```
Default progression:
0 1 2 3 4 5 6 7 8 9
Arithmetic progression with increment 5:
0 5 10 15 20 25 30 35 40 45
Arithmetic progression with increment 5 and start 2:
2 7 12 17 22 27 32 37 42 47
Geometric progression with default base:
1 2 4 8 16 32 64 128 256 512
Geometric progression with base 3:
1 3 9 27 81 243 729 2187 6561 19683
Fibonacci progression with default start values:
0 1 1 2 3 5 8 13 21 34
Fibonacci progression with start values 4 and 6:
4 6 10 16 26 42 68 110 178 288
```

2.4.3 抽象基类

在定义一组类的继承层次结构时, 避免重复代码的技术之一是设计一个基类, 该基类可以被需要它的其他类所继承。例如, 2.4.2 节的层次结构中包含一个 `Progression` 类, 它是三个不同的子类 (`ArithmeticProgression` 类、`GeometricProgression` 类和 `FibonacciProgression` 类) 的基类。虽然可以创建 `Progression` 基类的实例, 但这样做没有价值, 因为这只是一个增量为 1 的 `ArithmeticProgression` 类的特例。`Progression` 类的真正目的是集中实现其他数列需要的行为, 以简化这些子类的代码。

在经典的面向对象的术语中，如一个类的唯一目的是作为继承的基类，那么这个类就是一个抽象基类。更正式地说，一个抽象类不能直接实例化，而具体的类可以被实例化。根据这个定义，`Progression` 类严格来说是具体的类，尽管我们实质上把它设计为一个抽象基类。

在静态类型的语言中，如 Java 和 C++，抽象基类作为一个正式的类型，可以确保一个或多个抽象方法。这就为多态性提供了支持，因为变量可以有一个抽象基类作为其声明的类型，即使它是一个具体子类的实例。因为在 Python 中没有声明类型，这种多态性不需要一个统一的抽象基类就可以实现。出于这个原因，Python 中没有那么强烈地要求定义正式的抽象基类，尽管 Python 的 `abc` 模块提供了正式的抽象基类的定义。

我们之所以在研究数据结构时专注于抽象基类，是因为 Python 的 `collections` 模块提供了几个抽象基类，来协助自定义的数据结构与一些 Python 的内置数据结构共享一个共同的接口。这些抽象基类依赖于一个面向对象的软件设计模式，即模板方法模式。模板方法模式是一个抽象基类在提供依赖于调用其他抽象行为时的具体行为。在这种方式中，只要一个子类提供定义了缺失的抽象行为，继承的具体行为也就被定义了。

下面给出一个完整的例子，抽象基类 `collections.Sequence` 定义了 Python 的 `list`、`str` 和 `tuple` 类的共同行为，即通过一个整数索引访问序列中的元素。而且 `collections.Sequence` 类提供了 `count`、`index` 和 `__contains__` 方法的具体实现，可以被其他提供了 `__len__` 和 `__getitem__` 方法的具体实现的类所继承。出于演示的目的，我们提供了代码段 2-14 的实现样例。

代码段 2-14 一个类似于 `Collections.Sequence` 的抽象基类

```

1 from abc import ABCMeta, abstractmethod      # need these definitions
2
3 class Sequence(metaclass=ABCMeta):
4     """Our own version of collections.Sequence abstract base class."""
5
6     @abstractmethod
7     def __len__(self):
8         """Return the length of the sequence."""
9
10    @abstractmethod
11    def __getitem__(self, j):
12        """Return the element at index j of the sequence."""
13
14    def __contains__(self, val):
15        """Return True if val found in the sequence; False otherwise."""
16        for j in range(len(self)):
17            if self[j] == val:                      # found match
18                return True
19        return False
20
21    def index(self, val):
22        """Return leftmost index at which val is found (or raise ValueError)."""
23        for j in range(len(self)):
24            if self[j] == val:                      # leftmost match
25                return j
26        raise ValueError('value not in sequence')  # never found a match
27
28    def count(self, val):
29        """Return the number of elements equal to given value."""
30        k = 0
31        for j in range(len(self)):
32            if self[j] == val:                      # found a match
33                k += 1
34        return k

```

这个实现依赖于 Python 的两个高级技术。第一个技术是声明 abc 模块中的 ABCMeta 类作为 Sequence 类的元类。元类不同于超类，它为类定义本身提供了一个模板。具体来说，ABCMeta 声明确保类的构造函数引发异常。

第二个先进技术是在 `__len__` 和 `__getitem__` 方法声明前立即使用 `@abstractmethod` 声明。这就声明了这两种特定的方法是抽象的，也意味着不需要在 Sequence base 类中提供实现，但我们期望任何具体的子类来实现这两种方法。Python 通过禁止没有重载抽象方法的具体实现的任何子类实例化来强制执行这个期望。

在 `__len__` 和 `__getitem__` 方法将存在于具体子类的假设下，Sequence 类定义的其余部分提供了其他行为的完整实现。如果你仔细检查源代码，会发现除了语法 `len(self)` 和 `self[j]` 分别通过特殊方法 `__len__` 和 `__getitem__` 支持外，`__contains__` 和 `index` 的具体实现不依赖于实例本身的一切假设，迭代支持也是自动的，正如 2.3.4 节所描述的那样。

在本书的其余部分，我们省略使用 abc 模块的形式。如果需要一个抽象基类，我们只是简单地在文档中记录对子类提供的功能的期望，而不需要正式声明抽象类。但是我们将使用的抽象基类是在 collection 模块（如 Sequence）中定义好的。使用这样的一个类，我们只需要依靠标准的继承技术。

例如，2.3.5 节的代码段 2-6 中的 Range 类就是一个支持 `__len__` 和 `__getitem__` 方法的类，但该类不支持方法 `count` 和 `index`。我们最初将 Sequence 类声明为一个超类，那么它也将继承 `count` 和 `index` 方法。声明语法如下：

```
class Range(collections.Sequence):
```

最后，需要强调的是，如果一个子类对从基类继承的行为提供自己的实现，那么新的定义会覆盖之前继承的。当我们有能力自己实现一个比通用方法更有效率的方法时，这种技术就可以被使用。例如，Sequence 类中的 `__contains__` 方法的通用实现是基于在循环中搜索想要的值。但对于 Range 类，这里有一个更有效的方法。如，表达式 `100000 in Range(0, 2000000, 100)` 很明显计算为真，甚至不用去检测范围中的元素，因为范围是从 0 开始，以 100 递增，直至数字达到 2 000 000。它一定包括 100 000，因为它是 100 的倍数，也在 $0 \sim 2\,000\,000$ 之间。练习 C-2.27 提出的目标是实现 Range.`__contains__` 方法，并且不使用（超时）循环。

2.5 命名空间和面向对象

命名空间是一个抽象名词，它管理着特定范围内定义的所有标识符，将每个名称映射到相应的值。在 Python 中，函数、类和模块都是第一类对象，所以命名空间内与标识符相关的“值”可能实际上是一个函数、类或模块。

我们在 1.10 节探讨了 Python 使用命名空间来管理全局范围内定义的标识符，以及在函数调用时局部范围内定义的标识符。在这一节，我们将讨论面向对象管理中命名空间的重要作用。

2.5.1 实例和类命名空间

首先，我们开始探讨所谓的实例命名空间，就是管理单个对象的特定属性。例如，CreditCard 类的每个实例都包含不同的余额、不同的账号、不同的信用额度等（虽然某些情况下巧合地有着相同的余额或信用额度）。每张信用卡将有一个专用的实例命名空间来管理

这些值。

每个已定义的类都有一个单独的类命名空间。这个命名空间用于管理一个类的所有实例所共享的成员或没有引用任何特定实例的成员。例如，2.3 节的 CreditCard 类中的 make_payment 方法不是被该类中的每个实例单独存储，该成员函数存储在 CreditCard 类的命名空间中。基于代码段 2-1 和 2-2 中的定义，CreditCard 类的命名空间包含的函数有 __init__、__get_customer__、get_bank、get_account、get_balance、get_limit、charge 和 make_payment。PredatoryCreditCard 类有自己的命名空间，其中包含了我们为该子类定义的三种方法：__init__、charge 和 process_month。

图 2-8 提供了三个命名空间：第一个类命名空间包含 CreditCard 类的方法，第二个类命名空间包含 PredatoryCreditCard 类的方法，最后一个是 PredatoryCreditCard 类的实例命名空间。我们注意到名为 charge 的函数有两种不同的定义：一个是在 CreditCard 类，另一个是在 PredatoryCreditCard 类中重写了该方法。类似的，也有两种不同的 __init__ 实现。但 process_month 是仅在 PredatoryCreditCard 类的范围内定义的名字。实例命名空间包含了该实例的所有数据成员（包括 PredatoryCreditCard 类构造方法中定义的 _apr 成员）。

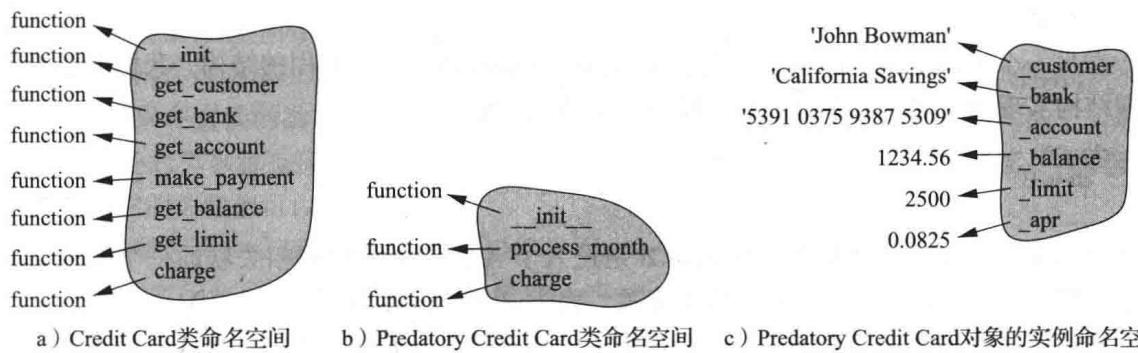


图 2-8 三种命名空间的概念视图

条目是怎样在命名空间中建立的

为什么有的成员（如 _balance）驻留在 Credit Card 类的实例命名空间，而有的成员（如 make_payment）驻留在类命名空间？理解这一问题是非常重要的。当新的信用卡实例构造好后，balance 成员就在 __init__ 建立起来了。原始的赋值使用语法 self.balance = 0，其中 self 是新创建实例中的标识符。在这种赋值中，self._balance 中 self 作为限定符使用，这使得 _balance 标识符直接被添加到实例命名空间中。

当使用继承时，每个对象仍有单一的实例命名空间。例如，当构造 PredatoryCreditCard 类的一个实例后，_apr 属性以及如 _balance 和 _limit 等属性都驻留在该实例的命名空间，因为所有赋值都使用一个特定的语法，如 self._apr。

一个类命名空间包含所有直接在类定义体内的声明。例如，CreditCard 类定义有以下结构：

```
class CreditCard:
    def make_payment(self, amount):
        ...
```

因为 make_payment 函数是在 CreditCard 类中声明的，所以它也与 CreditCard 类命名空间中的名字 make_payment 相关联。尽管成员函数是最典型的在类命名空间中声明的条目类型，但我们接下来还会讨论其他数据值的类型，甚至讨论其他类是怎样在类命名空间中声明的。