

Python数据结构 与算法分析

(第2版)



[美] 布拉德利·米勒 著
戴维·拉努姆
吕能 刁寿钧 译

Problem Solving with
Algorithms and Data Structures
Using Python Second Edition

- 只有洞彻数据结构与算法，才能真正精通Python
- 经典计算机科学教材，华盛顿大学等多家高校采用



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

作者简介

布拉德利·米勒

(Bradley N. Miller)

美国路德学院计算机科学名誉教授，曾获美国计算机协会软件系统奖，对Python课程开发有深入研究，由他创立的互动式教科书平台Runestone Interactive与全球600多家教育机构有合作。

戴维·拉努姆

(David L. Ranum)

IBM Watson认知软件工程师，医学信息学博士，致力于利用自然语言处理等人工智能技术解决医疗问题，曾在美国路德学院讲授计算机科学课程近三十载。

译者简介

吕能

Twitter软件工程师，开源项目Apache Heron的核心贡献者。先后在浙江大学和美国加州大学洛杉矶分校取得计算机科学学士学位和硕士学位，关注分布式实时数据引擎系统的研发，热衷于普及计算机技术知识。

刁寿钧

腾讯优图实验室后台开发工程师，毕业于复旦大学。先后从事过广告业务与智慧零售、智慧社区业务的开发工作。关注算法与数据库技术，曾协助组织IMG社区的技术沙龙活动。另译有《数据分析实战》。

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。



**Problem Solving with Algorithms and Data Structures
Using Python, Second Edition**

**Python数据结构与算法分析
(第2版)**

[美] 布拉德利·米勒 戴维·拉努姆 著
吕能 刁寿钧 译

人民邮电出版社
北京

图书在版编目（C I P）数据

Python数据结构与算法分析：第2版 / (美) 布拉德利·米勒 (Bradley N. Miller), (美) 戴维·拉努姆 (David L. Ranum) 著；吕能，刁寿钧译。— 北京：人民邮电出版社，2019.9
(图灵程序设计丛书)
ISBN 978-7-115-51721-0

I. ①P… II. ①布… ②戴… ③吕… ④刁… III. ①软件工具—程序设计 IV. ①TP311.561

中国版本图书馆CIP数据核字(2019)第155652号

内 容 提 要

了解数据结构与算法是透彻理解计算机科学的前提。随着 Python 日益广泛的应用，Python 程序员需要实现与传统的面向对象编程语言相似的数据结构与算法。本书是用 Python 描述数据结构与算法的开山之作，汇聚了作者多年的实战经验，向读者透彻讲解在 Python 环境下，如何通过一系列存储机制高效地实现各类算法。通过本书，读者将深刻理解 Python 数据结构、递归、搜索、排序、树与图的应用，等等。

本书适合所有 Python 程序员阅读。

◆ 著 [美] 布拉德利·米勒 戴维·拉努姆
译 吕能 刁寿钧
责任编辑 谢婷婷
责任印制 周昇亮

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京 印刷

◆ 开本：800×1000 1/16
印张：19.25
字数：466千字 2019年9月第1版
印数：1-3 000册 2019年9月北京第1次印刷
著作权合同登记号 图字：01-2018-0949号

定价：79.00元

读者服务热线：(010)51095183转600 印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版 权 声 明

Authorized translation from the English language edition, titled *Problem Solving with Algorithms and Data Structures Using Python, Second Edition* by Bradley N. Miller and David L. Ranum, Copyright © 2011.

All rights reserved. This book or any portion thereof may not be reproduced, transmitted or used in any manner whatsoever without the express written permission of the authors.

Simplified Chinese language edition published by Posts & Telecom Press, Copyright © 2019.

本书中文简体字版由 Franklin, Beedle & Associates 授权人民邮电出版社独家出版。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。

版权所有，侵权必究。

前　　言

致学生

既然你已经开始阅读本书，那么必定对计算机科学感兴趣。你可能也对 Python 这门编程语言感兴趣，并且已经通过之前的课程或自学有了一些编程经验。不论是何种情况，你肯定都希望学习更多知识。

本书将介绍计算机科学，也会介绍 Python。然而，书中的内容并不仅仅局限于这两个方面。学习数据结构与算法对理解计算机科学的本质非常关键。

学习计算机科学与掌握其他高难度学科没什么不同。成功的唯一途径便是循序渐进地学习其中的核心思想。刚开始接触计算机科学的人，需要多多练习来加深理解，从而为学习更复杂的内容做好准备。此外，初学者需要建立起自信心。

本书用作数据结构与算法的入门课的教材。数据结构与算法通常是计算机科学专业的第二门课程，比第一门课程更深入。但是本书的读者对象仍然是初学者，很可能还在第一门课程所教的基本思想和方法中挣扎，但已经准备好进一步探索这一领域并且进一步练习如何解决问题。

如前所述，本书将介绍计算机科学。它既会介绍抽象数据类型及数据结构，也会介绍如何编写算法和解决问题。你会学习一系列的数据结构，并且解决各种经典问题。随着学习的深入，你将反复应用在各章中掌握的工具与技术。

致教师

许多学生在学到这个阶段时会发现，计算机科学除了编程以外还有很多内容。数据结构与算法可以独立于编程来学习和理解。

本书假定学生已经学过计算机科学的入门课程，但入门课程不一定是用 Python 讲解的。他们理解基本的结构体，比如分支、迭代以及函数定义，也接触过面向对象编程，并且能够创建和使用简单的类。同时，学生也能够理解 Python 的基础数据结构，比如序列（列表和字符串）以及字典。

本书有三大特点：

- 通过简单易读的文字而不引入太多编程语法，重点关注如何解决问题，从而向学生介绍基本的数据结构与算法；
- 较早介绍基于大 O 记法的算法分析，并且通篇运用；
- 使用 Python 讲解，以促使初学者能够使用和掌握数据结构与算法。

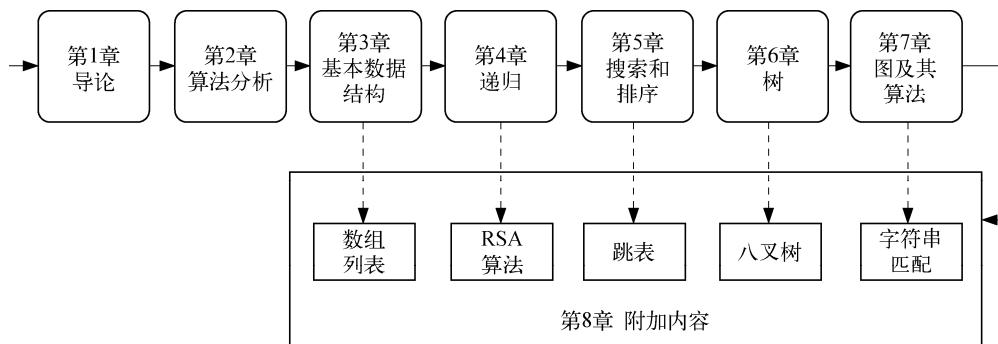
学生将首先学习线性数据结构，包括栈、队列、双端队列以及列表。我们用 Python 列表以及链表实现这些数据结构。然后学习与树有关的非线性数据结构，了解连接节点和引用结构（链表）等一系列技术。最后，学生将通过运用链式结构、链表以及 Python 字典的实现，学习图的相关知识。对于每一种结构，本书都尽力在使用 Python 提供的内建数据类型的同时展现众多的实现技巧。这种讲法在向学生揭示各种主要实现方法的同时，也强调 Python 的易用性。

Python 是一门非常适合于讲解算法的语言，语法干净简洁，用户环境直观，基本的数据类型十分强大和易用。其交互性在不需要额外编写驱动函数的情况下为测试数据结构单元提供了直观环境。而且，Python 为算法提供了教科书式的表示法，基本上不需要再用伪代码。这一特性有助于通过数据结构与算法来描述众多与之有关、相当有趣的现代问题。

我们相信，对于初学者来说，投入时间学习与算法和数据结构相关的基本思想是非常有益的。我们也相信，Python 是一门教授初学者入门的优秀语言。其他许多语言要求学生学习非常高级的编程概念，这会阻碍他们掌握真正需要的基础知识，从而可能导致失败，而这样的失败并不是计算机科学本身造成的，而是由于所使用的语言不当。我们的目标是提供一本教科书，量体裁衣般地聚焦于他们需要掌握的内容，以他们能理解的方式编写，创造和发展一个有助于他们成功的环境。

本书结构

本书紧紧围绕着运用经典数据结构和技术来解决问题。下面的组织结构图展示了充分利用本书的不同方式。



第 1 章做一些背景知识的准备，我们来复习一下计算机科学、问题解决、面向对象编程以及 Python。基础扎实的学生可以跳过第 1 章，直接学习第 2 章。不过，正所谓温故而知新，适当的复习和回顾必然是值得的。

第 2 章介绍算法分析的内在思想，重点讲解大 O 记法，还将分析本书一直使用的重要 Python 数据结构。这可以帮助学生理解如何权衡各种抽象数据类型的不同实现。第 2 章也包含了在运行时使用的 Python 原生类型的实验测量例子。

第 3~7 章全面介绍在经典计算机科学问题中出现的数据结构与算法。尽管在阅读顺序上并无严格要求，但是许多话题之间都存在一定的依赖关系，所以应该按照本书的顺序学习。比如，第 3 章介绍栈，第 4 章利用栈解释递归，第 5 章利用递归实现二分搜索。

第 8 章是选学内容，包含彼此独立的几节。每一节都与之前的某一章有关。正如前面的组织结构图所示，你既可以在学习完第 7 章以后再一起学习第 8 章中的各节内容，也可以把它们与对应的那一章放在一起学习。例如，希望更早介绍数组的教师，可以在讲完第 3 章以后直接跳到 8.2 节。

新版改进

- 所有的代码都用 Python 3.2 写成。
- 第 1 章介绍 Python 的集合以及异常处理。
- 不再使用第三方绘图包。新版中的所有图形都通过原生的 `turtle` 模块绘制。
- 新加的第 2 章着重讲解算法分析。此外，这一章还包括对全书出现的关键 Python 数据结构的分析。
- 第 3 章新增了关于链表实现的一节。
- 将“动态规划”一节移到了第 4 章的最后。
- 更关注图递归算法，包括递归树绘制以及递归迷宫搜索程序。
- 所有的数据结构源代码都被放在一个 Python 包中，方便学生在完成作业时使用。
- 新版包含各章例子的完整源代码，从而避免了从各处复制代码的麻烦。
- 第 6 章改进了二叉搜索树。
- 第 6 章新增了关于平衡二叉树的一节。
- 第 8 章介绍 C 风格的数组和数组管理。

致谢

在完成本书的过程中，我们得到了众多朋友的帮助。感谢我们的同事 Steve Hubbard 为本书的第 1 版提供了大量的反馈，并为新版提供了众多新素材。感谢各地的同事给我们发邮件指出第

1 版存在的错误，并且为新版内容提供意见。

感谢迪科拉市 Java John's 咖啡馆的朋友 Mary 和 Bob，以及其他服务员，他们允许我俩在 Brad 休假期间成为店里的“常驻作者”。David 在常驻咖啡馆写书的那几个月中竟然没有成为咖啡爱好者。对了，在一间店名带 Java 的咖啡馆里写一本 Python 的书，确实有点讽刺。

感谢 Franklin, Beedle & Associates 出版公司的各位员工，特别是 Jim Leisy 和 Tom Sumner。与他们的合作非常愉快。最后，还要特别感谢我们两人的妻子 Jane Miller 和 Brenda Ranum。她们的爱与支持使得本书终成现实。

电子书

扫描如下二维码，即可购买本书电子版。



布拉德利 · 米勒 (Bradley N. Miller)

戴维 · 拉努姆 (David L. Ranum)

目 录

第 1 章 导论	1
1.1 本章目标	1
1.2 入门	1
1.3 何谓计算机科学	1
1.3.1 何谓编程	3
1.3.2 为何学习数据结构及抽象数据类型	4
1.3.3 为何学习算法	4
1.4 Python 基础	5
1.4.1 数据	5
1.4.2 输入与输出	16
1.4.3 控制结构	18
1.4.4 异常处理	21
1.4.5 定义函数	23
1.4.6 Python 面向对象编程：定义类	24
1.5 小结	37
1.6 关键术语	38
1.7 讨论题	38
1.8 编程练习	38
第 2 章 算法分析	40
2.1 本章目标	40
2.2 何谓算法分析	40
2.2.1 大 O 记法	43
2.2.2 异序词检测示例	46
2.3 Python 数据结构的性能	49
2.3.1 列表	49
2.3.2 字典	53
2.4 小结	55
2.5 关键术语	55
2.6 讨论题	56
2.7 编程练习	56
第 3 章 基本数据结构	57
3.1 本章目标	57
3.2 何谓线性数据结构	57
3.3 栈	58
3.3.1 何谓栈	58
3.3.2 栈抽象数据类型	59
3.3.3 用 Python 实现栈	60
3.3.4 匹配括号	62
3.3.5 普通情况：匹配符号	64
3.3.6 将十进制数转换成二进制数	65
3.3.7 前序、中序和后序表达式	67
3.4 队列	75
3.4.1 何谓队列	75
3.4.2 队列抽象数据类型	75
3.4.3 用 Python 实现队列	76
3.4.4 模拟：传土豆	77
3.4.5 模拟：打印任务	79
3.5 双端队列	84
3.5.1 何谓双端队列	84
3.5.2 双端队列抽象数据类型	84
3.5.3 用 Python 实现双端队列	85
3.5.4 回文检测器	86

3.6 列表	88	5.3.2 选择排序.....	147
3.6.1 无序列表抽象数据类型	88	5.3.3 插入排序.....	149
3.6.2 实现无序列表：链表	89	5.3.4 希尔排序.....	151
3.6.3 有序列表抽象数据类型	97	5.3.5 归并排序.....	153
3.6.4 实现有序列表	97	5.3.6 快速排序.....	156
3.7 小结	100	5.4 小结.....	159
3.8 关键术语	101	5.5 关键术语	160
3.9 讨论题	101	5.6 讨论题	160
3.10 编程练习.....	102	5.7 编程练习	161
第4章 递归	105	第6章 树	163
4.1 本章目标	105	6.1 本章目标	163
4.2 何谓递归	105	6.2 示例	163
4.2.1 计算一列数之和	105	6.3 术语及定义	166
4.2.2 递归三原则	107	6.4 实现	168
4.2.3 将整数转换成任意进制的 字符串	108	6.4.1 列表之列表	168
4.3 栈帧：实现递归	110	6.4.2 节点与引用	171
4.4 递归可视化	111	6.5 二叉树的应用	173
4.5 复杂的递归问题	116	6.5.1 解析树	173
4.6 探索迷宫	118	6.5.2 树的遍历	179
4.7 动态规划	123	6.6 利用二叉堆实现优先级队列	182
4.8 小结	128	6.6.1 二叉堆的操作	182
4.9 关键术语	129	6.6.2 二叉堆的实现	183
4.10 讨论题	129	6.7 二叉搜索树	189
4.11 编程练习	129	6.7.1 搜索树的操作	190
第5章 搜索和排序	131	6.7.2 搜索树的实现	190
5.1 本章目标	131	6.7.3 搜索树的分析	201
5.2 搜索	131	6.8 平衡二叉搜索树	202
5.2.1 顺序搜索	131	6.8.1 AVL树的性能	203
5.2.2 二分搜索	134	6.8.2 AVL树的实现	204
5.2.3 散列	136	6.8.3 映射实现总结	210
5.3 排序	145	6.9 小结	211
5.3.1 冒泡排序	145	6.10 关键术语	211
		6.11 讨论题	211
		6.12 编程练习	213

第 7 章 图及其算法	214	第 8 章 附加内容	251
7.1 本章目标	214	8.1 本章目标	251
7.2 术语及定义	215	8.2 复习 Python 列表	251
7.3 图的抽象数据类型	216	8.3 复习递归	256
7.3.1 邻接矩阵	216	8.3.1 同余定理	257
7.3.2 邻接表	217	8.3.2 幂剩余	257
7.3.3 实现	218	8.3.3 最大公因数与逆元	258
7.4 宽度优先搜索	220	8.3.4 RSA 算法	261
7.4.1 词梯问题	220	8.4 复习字典：跳表	264
7.4.2 构建词梯图	221	8.4.1 映射抽象数据类型	265
7.4.3 实现宽度优先搜索	223	8.4.2 用 Python 实现字典	265
7.4.4 分析宽度优先搜索	226	8.5 复习树：量化图片	274
7.5 深度优先搜索	226	8.5.1 数字图像概述	274
7.5.1 骑士周游问题	226	8.5.2 量化图片	275
7.5.2 构建骑士周游图	227	8.5.3 使用八叉树改进量化算法	277
7.5.3 实现骑士周游	229	8.6 复习图：模式匹配	284
7.5.4 分析骑士周游	231	8.6.1 生物学字符串	285
7.5.5 通用深度优先搜索	233	8.6.2 简单比较	285
7.5.6 分析深度优先搜索	236	8.6.3 使用图：DFA	287
7.6 拓扑排序	236	8.6.4 使用图：KMP	288
7.7 强连通单元	238	8.7 小结	291
7.8 最短路径问题	241	8.8 关键术语	291
7.8.1 Dijkstra 算法	243	8.9 讨论题	291
7.8.2 分析 Dijkstra 算法	245	8.10 编程练习	292
7.8.3 Prim 算法	245	附录 A Python 图形包	293
7.9 小结	248	附录 B Python 资源	294
7.10 关键术语	249	参考资料	295
7.11 讨论题	249			
7.12 编程练习	250			

第 1 章

导 论



1.1 本章目标

- 复习计算机科学、编程以及解决问题方面的知识。
- 理解抽象这一概念及其在解决问题的过程中所发挥的作用。
- 理解并建立抽象数据类型的概念。
- 复习 Python。

1.2 入门

自从第一台利用转接线和开关来传递计算指令的电子计算机诞生以来，人们对编程的认识历经了多次变化。与社会生活的其他许多方面一样，计算机技术的变革为计算机科学家提供了越来越多的工具和平台去施展他们的才能。高效的处理器、高速网络以及大容量内存等一系列新技术，要求计算机科学家掌握更多复杂的知识。然而，在这一系列快速的变革之中，仍有一些基本原则始终保持不变。计算机科学被认为是一门利用计算机来解决问题的学科。

你肯定在学习解决问题的基本方法上投入过大量的时间，并且相信自己拥有根据问题描述构建解决方案的能力。你肯定也体会到了编写计算机程序的困难之处。大型难题及其解决方案的复杂性往往会掩盖问题解决过程的核心思想。

本章将为后续各章重点解释两个重要的话题。首先，本章会复习计算机科学以及数据结构与算法的研究必须符合的框架，尤其是学习这些内容的原因以及为什么说理解它们有助于更好地解决问题。其次，本章会复习 Python。尽管不会提供完整、详尽的 Python 参考资料，但是会针对阅读后续各章所需的基础知识及基本思想，给出示例以及相应的解释。

1.3 何谓计算机科学

要定义计算机科学，通常十分困难，这也许是因为其中的“计算机”一词。你可能已经意识到，计算机科学并不仅是研究计算机本身。尽管计算机在这一学科中是非常重要的工具，但也仅

仅只是工具而已。

计算机科学的研究对象是问题、解决问题的过程，以及通过该过程得到的解决方案。给定一个问题，计算机科学家的目标是开发一个能够逐步解决该问题的算法。算法是具有有限步骤的过程，依照这个过程便能解决问题。因此，算法就是解决方案。

可以认为计算机科学就是研究算法的学科。但是必须注意，某些问题并没有解决方案。尽管这一话题已经超出了本书讨论的范畴，但是对于学习计算机科学的人来说，认清这一事实非常重要。结合上述两类问题，可以将计算机科学更完善地定义为：研究问题及其解决方案，以及研究目前无解的问题的学科。

在描述问题及其解决方案时，经常用到“可计算”一词。若存在能够解决某个问题的算法，那么该问题便是可计算的。因此，计算机科学也可以被定义为：研究可计算以及不可计算的问题，即研究算法的存在性以及不存在性。在上述任意一种定义中，“计算机”一词都没有出现。解决方案本身是独立于计算机的。

在研究问题解决过程的同时，计算机科学也研究抽象。抽象思维使得我们能分别从逻辑视角和物理视角来看待问题及其解决方案。举一个常见的例子。

试想你每天开车去上学或上班。作为车的使用者，你在驾驶时会与它有一系列的交互：坐进车里，插入钥匙，启动发动机，换挡，刹车，加速以及操作方向盘。从抽象的角度来看，这是从逻辑视角来看待这辆车，你在使用由汽车设计者提供的功能来将自己从某个地方运送到另一个地方。这些功能有时候也被称作接口。

另一方面，修车工看待车辆的角度与司机截然不同。他不仅需要知道如何驾驶，而且更需要知道实现汽车功能的所有细节：发动机如何工作，变速器如何换挡，如何控制温度，等等。这就是所谓的物理视角，即看到表面之下的实现细节。

使用计算机也是如此。大多数人用计算机来写文档、收发邮件、浏览网页、听音乐、存储图像以及打游戏，但他们并不需要了解这些功能的实现细节。大家都是从逻辑视角或者使用者的角度来看待计算机。计算机科学家、程序员、技术支持人员以及系统管理员则从另一个角度来看待计算机。他们必须知道操作系统的原理、网络协议的配置，以及如何编写各种脚本来控制计算机。他们必须能够控制用户不需要了解的底层细节。

上面两个例子的共同点在于，抽象的用户（或称客户）只需要知道接口是如何工作的，而并不需要知道实现细节。这些接口是用户用于与底层复杂的实现进行交互的方式。下面是抽象的另一个例子，来看看 Python 的 `math` 模块。一旦导入这一模块，便可以进行如下的计算。

```
>>> import math  
>>> math.sqrt(16)  
4.0  
>>>
```

这是一个过程抽象的例子。我们并不需要知道平方根究竟是如何计算出来的，而只需要知道

计算平方根的函数名是什么以及如何使用它。只要正确地导入模块，便可以认为这个函数会返回正确的结果。由于其他人已经实现了平方根问题的解决方案，因此我们只需要知道如何使用该函数即可。这有时候也被称为过程的“黑盒”视角。我们仅需要描述接口：函数名、所需参数，以及返回值。所有的计算细节都被隐藏了起来，如图 1-1 所示。

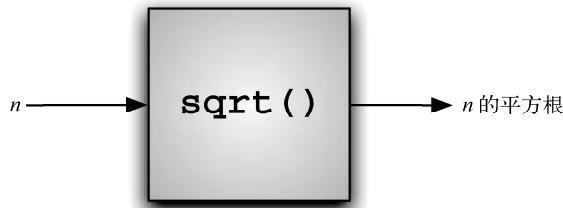


图 1-1 过程抽象

1.3.1 何谓编程

编程是指通过编程语言将算法编码以使其能被计算机执行的过程。尽管有众多的编程语言和不同类型的计算机，但是首先得有一个解决问题的算法。如果没有算法，就不会有程序。

计算机科学的研究对象并不是编程。但是，编程是计算机科学家所做工作的一个重要组成部分。通常，编程就是为解决方案创造表达方式。因此，编程语言对算法的表达以及创造程序的过程是这一学科的基础。

通过定义表达问题实例所需的数据，以及得到预期结果所需的计算步骤，算法描述出了问题的解决方案。编程语言必须提供一种标记方式，用于表达过程和数据。为此，编程语言提供了众多的控制语句和数据类型。

控制语句使算法步骤能够以一种方便且明确的方式表达出来。算法至少需要能够进行顺序执行、决策分支、循环迭代的控制语句。只要一种编程语言能够提供这些基本的控制语句，它就能够被用于描述算法。

计算机中的所有数据实例均由二进制字符串来表达。为了赋予这些数据实际的意义，必须要有数据类型。数据类型能够帮助我们解读二进制数据的含义，从而使我们能从待解决问题的角度来看待数据。这些内建的底层数据类型（又称原生数据类型）提供了算法开发的基本单元。

举例来说，大部分编程语言都为整数提供了相应的数据类型。根据整数（如 23、654 以及 -19）的常见定义，计算机内存中的二进制字符串可以被理解成整数。除此以外，数据类型也描述了该类数据能参与的所有运算。对于整数来说，就有加减乘除等常见运算。并且，对于数值类型的数据，以上运算均成立。

我们经常遇到的困难是，问题及其解决方案都过于复杂。尽管由编程语言提供的简单的控制语句和数据类型能够表达复杂的解决方案，但它们在解决问题的过程中仍然存在不足。因此，我

们需要想办法控制复杂度以利于找到解决方案。

1.3.2 为何学习数据结构及抽象数据类型

为了控制问题及其求解过程的复杂度，计算机科学家利用抽象来帮助自己专注于全局，从而避免迷失在众多细节中。通过对问题进行建模，可以更高效地解决问题。模型可以帮助计算机科学家更一致地描述算法要用到的数据。

如前所述，过程抽象将功能的实现细节隐藏起来，从而使用户能从更高的视角来看待功能。数据抽象的基本思想与此类似。抽象数据类型（有时简称为 ADT）从逻辑上描述了如何看待数据及其对应运算而无须考虑具体实现。这意味着我们仅需要关心数据代表了什么，而可以忽略它们的构建方式。通过这样的抽象，我们对数据进行了一层封装，其基本思想是封装具体的实现细节，使它们对用户不可见，这被称为信息隐藏。

图 1-2 展示了抽象数据类型及其原理。用户通过利用抽象数据类型提供的操作来与接口交互。抽象数据类型是与用户交互的外壳。真正的实现则隐藏在内部。用户并不需要关心各种实现细节。

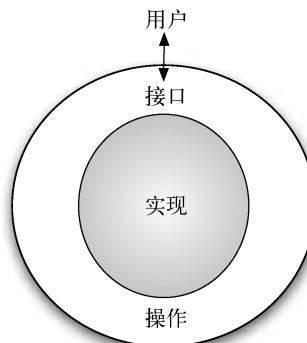


图 1-2 抽象数据类型

抽象数据类型的实现常被称为数据结构，这需要我们通过编程语言的语法结构和原生数据类型从物理视角看待数据。正如之前讨论的，分成这两种不同的视角有助于为问题定义复杂的数据模型，而无须考虑模型的实现细节。这便提供了一个独立于实现的数据视角。由于实现抽象数据类型通常会有很多种方法，因此独立于实现的数据视角使程序员能够改变实现细节而不影响用户与数据的实际交互。用户能够始终专注于解决问题。

1.3.3 为何学习算法

计算机科学家通过经验来学习：观察他人如何解决问题，然后亲自解决问题。接触各种问题解决技巧并学习不同算法的设计方法，有助于解决新的问题。通过学习一系列不同的算法，可以举一反三，从而在遇到类似的问题时，能够快速加以解决。

各种算法之间往往差异巨大。回想前文提到的平方根的例子，完全可能有多种方法来实现计算平方根的函数。算法一可能使用了较少的资源，算法二返回结果所需的时间可能是算法一的 10 倍。我们需要某种方式来比较这两种算法。尽管这两种算法都能得到结果，但是其中一种可能比另一种“更好”——更高效、更快，或者使用的内存更少。随着对算法的进一步学习，你会掌握比较不同算法的分析技巧。这些技巧只依赖于算法本身的特性，而不依赖于程序或者实现算法的计算机的特性。

最坏的情况是遇到难以解决的问题，即没有算法能够在合理的时间内解决该问题。因此，至关重要的一点是，要能区分有解的问题、无解的问题，以及虽然有解但是需要过多的资源和时间来求解的问题。

在选择算法时，经常会有所权衡。除了有解决问题的能力之外，计算机科学家也需要知晓如何评估一个解决方案。总之，问题通常有很多解决方案，如何找到一个解决方案并且确定其为优秀的方案，是需要反复练习、熟能生巧的。

1.4 Python 基础

本节将复习 Python，并且为前一节提到的思想提供更详细的例子。如果你刚开始学习 Python 或者觉得自己需要更多的信息，建议你查看本书结尾列出的 Python 资源。本节的目标是帮助你复习 Python 并且强化一些会在后续各章中变得非常重要的概念。

Python 是一门现代、易学、面向对象的编程语言。它拥有强大的内建数据类型以及简单易用的控制语句。由于 Python 是一门解释型语言，因此只需要查看和描述交互式会话就能进行学习。你应该记得，解释器会显示提示符>>>，然后计算你提供的 Python 语句。例如，以下代码显示了提示符、print 函数、结果，以及下一个提示符。

```
>>> print("Algorithms and Data Structures")
>>> Algorithms and Data Structures
>>>
```

1.4.1 数据

前面提到，Python 支持面向对象编程范式。这意味着 Python 认为数据是问题解决过程中的关键点。在 Python 以及其他所有面向对象编程语言中，类都是对数据的构成（状态）以及数据能做什么（行为）的描述。由于类的使用者只能看到数据项的状态和行为，因此类与抽象数据类型是相似的。在面向对象编程范式中，数据项被称作对象。一个对象就是类的一个实例。

1. 内建原子数据类型

我们首先复习原子数据类型。Python 有两大内建数据类实现了整数类型和浮点数类型，相应的 Python 类就是 int 和 float。标准的数学运算符，即+、-、*、/ 以及 **（幂），可以和能够改变运算优先级的括号一起使用。其他非常有用的运算符包括取余（取模）运算符%，以及整除

运算符`//`。注意，当两个整数相除时，其结果是一个浮点数，而整除运算符截去小数部分，只返回商的整数部分。

```
>>> 2+3*4
14
>>> (2+3)*4
20
>>> 2**10
1024
>>> 6/3
2.0
>>> 7/3
2.333333333333335
>>> 7//3
2
>>> 7%3
1
>>> 3/6
0.5
>>> 3//6
0
>>> 3%6
3
>>> 2**100
1267650600228229401496703205376
>>>
```

Python 通过 `bool` 类实现对表达真值非常有用的布尔数据类型。布尔对象可能的状态值是 `True` 或者 `False`，布尔运算符有 `and`、`or` 以及 `not`。

```
>>> True
True
>>> False
False
>>> False or True
True
>>> not (False or True)
False
>>> True and True
True
```

布尔对象也被用作相等 (`==`)、大于 (`>`) 等比较运算符的计算结果。此外，结合使用关系运算符与逻辑运算符可以表达复杂的逻辑问题。表 1-1 展示了关系运算符和逻辑运算符。

表 1-1 关系运算符和逻辑运算符

运 算 名	运 算 符	解 释
小于	<code><</code>	小于运算符
大于	<code>></code>	大于运算符
小于或等于	<code><=</code>	小于或等于运算符

(续)

运 算 名	运 算 符	解 释
大于或等于	<code>>=</code>	大于或等于运算符
等于	<code>==</code>	相等运算符
不等于	<code>!=</code>	不等于运算符
逻辑与	<code>and</code>	两个运算数都为 True 时结果为 True
逻辑或	<code>or</code>	某一个运算数为 True 时结果为 True
逻辑非	<code>not</code>	对真值取反, False 变为 True, True 变为 False

```
>>> 5 == 10
False
>>> 10 > 5
True
>>> (5 >= 1) and (5 <= 10)
True
```

标识符在编程语言中被用作名字。Python 中的标识符以字母或者下划线（_）开头，区分大小写，可以是任意长度。需要记住的一点是，采用能表达含义的名字是良好的编程习惯，这使程序代码更易阅读和理解。

当一个名字第一次出现在赋值语句的左边部分时，会创建对应的 Python 变量。赋值语句将名字与值关联起来。变量存的是指向数据的引用，而不是数据本身。来看看下面的代码。

```
>>> theSum = 0
>>> theSum
0
>>> theSum = theSum + 1
>>> theSum
1
>>> theSum = True
>>> theSum
True
```

赋值语句 `theSum = 0` 会创建变量 `theSum`，并且令其保存指向数据对象 0 的引用（如图 1-3 所示）。Python 会先计算赋值运算符右边的表达式，然后将指向该结果数据对象的引用赋给左边的变量名。在本例中，由于 `theSum` 当前指向的数据是整数类型，因此该变量类型为整型。如果数据的类型发生改变（如图 1-4 所示），正如上面的代码给 `theSum` 赋值 `True`，那么变量的类型也会变成布尔类型。赋值语句改变了变量的引用，这体现了 Python 的动态特性。同样的变量可以指向许多不同类型的数据。



图 1-3 变量指向数据对象的引用

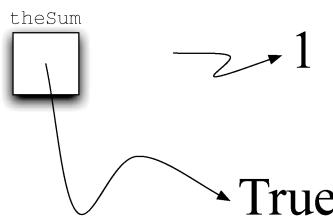


图 1-4 赋值语句改变变量的引用

2. 内建集合数据类型

除了数值类和布尔类，Python 还有众多强大的内建集合类。列表、字符串以及元组是概念上非常相似的有序集合，但是只有理解它们的差别，才能正确运用。集（set）和字典是无序集合。

列表是零个或多个指向 Python 数据对象的引用的有序集合，通过在方括号内以逗号分隔的一系列值来表达。空列表就是 `[]`。列表是异构的，这意味着其指向的数据对象不需要都是同一个类，并且这一集合可以被赋值给一个变量。下面的代码段展示了列表含有多个不同的 Python 数据对象。

```
>>> [1,3,True,6.5]
[1, 3, True, 6.5]
>>> myList = [1,3,True,6.5]
>>> myList
[1, 3, True, 6.5]
```

注意，当 Python 计算一个列表时，这个列表自己会被返回。然而，为了记住该列表以便后续处理，其引用需要被赋给一个变量。

由于列表是有序的，因此它支持一系列可应用于任意 Python 序列的运算，如表 1-2 所示。

表 1-2 可应用于任意 Python 序列的运算

运 算 名	运 算 符	解 释
索引	[]	取序列中的某个元素
连接	+	将序列连接在一起
重复	*	重复 N 次连接
成员	in	询问序列中是否有某元素
长度	len	询问序列的元素个数
切片	[:]	取出序列的一部分

需要注意的是，列表和序列的下标从 0 开始。`myList[1:3]`会返回一个包含下标从 1 到 2 的元素列表（并没有包含下标为 3 的元素）。

如果需要快速初始化列表，可以通过重复运算来实现，如下所示。

```
>>> myList = [0] * 6
>>> myList
[0, 0, 0, 0, 0, 0]
```

非常重要的一点是，重复运算返回的结果是序列中指向数据对象的引用的重复。下面的例子可以很好地说明这一点。

```
>>> myList = [1,2,3,4]
>>> A = [myList]*3
>>> A
[[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]]
>>> myList[2] = 45
>>> A
[[1, 2, 45, 4], [1, 2, 45, 4], [1, 2, 45, 4]]
>>>
```

变量 A 包含 3 个指向 myList 的引用。myList 中的一个元素发生改变，A 中的 3 处都随即改变。

列表支持一些用于构建数据结构的方法，如表 1-3 所示。后面的例子展示了用法。

表 1-3 Python 列表提供的方法

方 法 名	用 法	解 释
append	alist.append(item)	在列表末尾添加一个新元素
insert	alist.insert(i,item)	在列表的第 i 个位置插入一个元素
pop	alist.pop()	删除并返回列表中最后一个元素
pop	alist.pop(i)	删除并返回列表中第 i 个位置的元素
sort	alist.sort()	将列表元素排序
reverse	alist.reverse()	将列表元素倒序排列
del	del alist[i]	删除列表中第 i 个位置的元素
index	alist.index(item)	返回 item 第一次出现时的下标
count	alist.count(item)	返回 item 在列表中出现的次数
remove	alist.remove(item)	从列表中移除第一次出现的 item

```
>>> myList
[1024, 3, True, 6.5]
>>> myList.append(False)
>>> myList
[1024, 3, True, 6.5, False]
>>> myList.insert(2,4.5)
>>> myList
[1024, 3, 4.5, True, 6.5, False]
>>> myList.pop()
False
>>> myList
[1024, 3, 4.5, True, 6.5]
>>> myList.pop(1)
3
>>> myList
[1024, 4.5, True, 6.5]
>>> myList.pop(2)
True
```

```
>>> myList
[1024, 4.5, 6.5]
>>> myList.sort()
>>> myList
[4.5, 6.5, 1024]
>>> myList.reverse()
>>> myList
[1024, 6.5, 4.5]
>>> myList.count(6.5)
1
>>> myList.index(4.5)
2
>>> myList.remove(6.5)
>>> myList
[1024, 4.5]
>>> del myList[0]
>>> myList
[4.5]
```

你会发现，像 `pop` 这样的方法在返回值的同时也会修改列表的内容，`reverse` 等方法则仅修改列表而不返回任何值。`pop` 默认返回并删除列表的最后一个元素，但是也可以用来返回并删除特定的元素。这些方法默认下标从 0 开始。你也会注意到那个熟悉的句点符号，它被用来调用某个对象的方法。`myList.append(False)` 可以读作“请求 `myList` 调用其 `append` 方法并将 `False` 这一值传给它”。就连整数这类简单的数据对象都能通过这种方式调用方法。

```
>>> (54).__add__(21)
75
>>>
```

在上面的代码中，我们请求整数对象 54 执行其 `add` 方法（该方法在 Python 中被称为 `__add__`），并且将 21 作为要加的值传给它。其结果是两数之和，即 75。我们通常会将其写作 $54+21$ 。稍后会更多地讨论这些方法。

`range` 是一个常见的 Python 函数，我们常把它与列表放在一起讨论。`range` 会生成一个代表值序列的范围对象。使用 `list` 函数，能够以列表形式看到范围对象的值。下面的代码展示了这一点。

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

范围对象表示整数序列。默认情况下，它从 0 开始。如果提供更多的参数，它可以在特定的点开始和结束，并且跳过中间的值。在第一个例子中，`range(10)` 从 0 开始并且一直到 9 为止（不包含 10）；在第二个例子中，`range(5, 10)` 从 5 开始并且到 9 为止（不包含 10）；`range(5, 10, 2)` 的结果类似，但是元素的间隔变成了 2（10 还是没有包含在其中）。

字符串是零个或多个字母、数字和其他符号的有序集合。这些字母、数字和其他符号被称为字符。常量字符串值通过引号（单引号或者双引号均可）与标识符进行区分。

```
>>> "David"
'David'
>>> myName = "David"
>>> myName[3]
'i'
>>> myName*2
'DavidDavid'
>>> len(myName)
5
>>>
```

由于字符串是序列，因此之前提到的所有序列运算符都能用于字符串。此外，字符串还有一些特有的方法，表 1-4 列举了其中一些。

表 1-4 Python 字符串提供的方法

方 法 名	用 法	解 释
center	astring.center(w)	返回一个字符串，原字符串居中，使用空格填充新字符串，使其长度为 w
count	astring.count(item)	返回 item 出现的次数
ljust	astring.ljust(w)	返回一个字符串，将原字符串靠左放置并填充空格至长度 w
rjust	astring.rjust(w)	返回一个字符串，将原字符串靠右放置并填充空格至长度 w
lower	astring.lower()	返回均为小写字母的字符串
upper	astring.upper()	返回均为大写字母的字符串
find	astring.find(item)	返回 item 第一次出现时的下标
split	astring.split(schar)	在 schar 位置将字符串分割成子串

```
>>> myName
'David'
>>> myName.upper()
'DAVID'
>>> myName.center(10)
' David '
>>> myName.find('v')
2
>>> myName.split('v')
['Da', 'id']
```

`split` 在处理数据的时候非常有用。`split` 接受一个字符串，并且返回一个由分隔字符作为分割点的字符串列表。在本例中，v 就是分割点。如果没有提供分隔字符，那么 `split` 方法将

会寻找如制表符、换行符和空格等空白字符。

列表和字符串的主要区别在于，列表能够被修改，字符串则不能。列表的这一特性被称为可修改性。列表具有可修改性，字符串则不具有。例如，可以通过使用下标和赋值操作来修改列表中的一个元素，但是字符串不允许这一改动。

```
>>> myList
[1, 3, True, 6.5]
>>> myList[0]=2**10
>>> myList
[1024, 3, True, 6.5]
>>>
>>> myName
'David'
>>> myName[0]='X'

Traceback (most recent call last):
  File "<pyshell#84>", line 1, in <toplevel>
    myName[0]='X'
TypeError: object doesn't support item assignment
>>>
```

由于都是异构数据序列，因此元组与列表非常相似。它们的区别在于，元组和字符串一样是不可修改的。元组通常写成由括号包含并且以逗号分隔的一系列值。与序列一样，元组允许之前描述的任一操作。

```
>>> myTuple = (2,True,4.96)
>>> myTuple
(2, True, 4.96)
>>> len(myTuple)
3
>>> myTuple[0]
2
>>> myTuple * 3
(2, True, 4.96, 2, True, 4.96, 2, True, 4.96)
>>> myTuple[0:2]
(2, True)
>>>
```

然而，如果尝试改变元组中的一个元素，就会遇到错误。请注意，错误消息指明了问题的出处及原因。

```
>>> myTuple[1]=False
Traceback (most recent call last):
  File "<pyshell#137>", line 1, in <toplevel>
    myTuple[1]=False
TypeError: object doesn't support item assignment
>>>
```

集（set）是由零个或多个不可修改的Python数据对象组成的无序集合。集不允许重复元素，并且写成由花括号包含、以逗号分隔的一系列值。空集由set()来表示。集是异构的，并且可以

通过下面的方法赋给变量。

```
>>> {3, 6, "cat", 4.5, False}
{False, 4.5, 3, 6, 'cat'}
>>> mySet = {3, 6, "cat", 4.5, False}
>>> mySet
{False, 4.5, 3, 6, 'cat'}
>>>
```

尽管集是无序的，但它还是支持之前提到的一些运算，如表 1-5 所示。

表 1-5 Python 集支持的运算

运 算 名	运 算 符	解 释
成员	in	询问集中是否有某元素
长度	len	获取集的元素个数
	aset otherset	返回一个包含 aset 与 otherset 所有元素的新集
&	aset & otherset	返回一个包含 aset 与 otherset 共有元素的新集
-	aset - otherset	返回一个集，其中包含只出现在 aset 中的元素
<=	aset <= otherset	询问 aset 中的所有元素是否都在 otherset 中

```
>>> mySet
{False, 4.5, 3, 6, 'cat'}
>>> len(mySet)
5
>>> False in mySet
True
>>> "dog" in mySet
False
>>>
```

集支持一系列方法，如表 1-6 所示。在数学中运用过集合概念的人应该对它们非常熟悉。注意，union、intersection、issubset 和 difference 都有可用的运算符。

表 1-6 Python 集提供的方法

方 法 名	用 法	解 释
union	aset.union(otherset)	返回一个包含 aset 和 otherset 所有元素的集
intersection	aset.intersection(otherset)	返回一个仅包含两个集共有元素的集
difference	aset.difference(otherset)	返回一个集，其中仅包含只出现在 aset 中的元素
issubset	aset.issubset(otherset)	询问 aset 是否为 otherset 的子集
add	aset.add(item)	向 aset 添加一个元素
remove	aset.remove(item)	将 item 从 aset 中移除
pop	aset.pop()	随机移除 aset 中的一个元素
clear	aset.clear()	清除 aset 中的所有元素

```

>>> mySet
{False, 4.5, 3, 6, 'cat'}
>>> yourSet = {99,3,100}
>>> mySet.union(yourSet)
{False, 4.5, 3, 100, 6, 'cat', 99}
>>> mySet | yourSet
{False, 4.5, 3, 100, 6, 'cat', 99}
>>> mySet.intersection(yourSet)
{3}
>>> mySet & yourSet
{3}
>>> mySet.difference(yourSet)
{False, 4.5, 6, 'cat'}
>>> mySet - yourSet
{False, 4.5, 6, 'cat'}
>>> {3,100}.issubset(yourSet)
True
>>> {3,100}<=yourSet
True
>>> mySet.add("house")
>>> mySet
{False, 4.5, 3, 6, 'house', 'cat'}
>>> mySet.remove(4.5)
>>> mySet
{False, 3, 6, 'house', 'cat'}
>>> mySet.pop()
False
>>> mySet
{3, 6, 'house', 'cat'}
>>> mySet.clear()
>>> mySet
set()
>>>

```

最后要介绍的 Python 集合是字典。字典是无序结构，由相关的元素对构成，其中每对元素都由一个键和一个值组成。这种键-值对通常写成键:值的形式。字典由花括号包含的一系列以逗号分隔的键-值对表达，如下所示。

```

>>> capitals = {'Iowa':'DesMoines', 'Wisconsin':'Madison'}
>>> capitals
{'Wisconsin':'Madison', 'Iowa':'DesMoines'}
>>>

```

可以通过键访问其对应的值，也可以向字典添加新的键-值对。访问字典的语法与访问序列的语法十分相似，只不过是使用键来访问，而不是下标。添加新值也类似。

```

>>> capitals['Iowa']
'DesMoines'
>>> capitals['Utah']='SaltLakeCity'
>>> capitals
{'Utah':'SaltLakeCity', 'Wisconsin':'Madison', 'Iowa':'DesMoines'}
>>> capitals['California']='Sacramento'
>>> capitals

```

```

{'Utah':'SaltLakeCity', 'Wisconsin':'Madison', 'Iowa':'DesMoines',
'California':'Sacramento'}
>>> len(capitals)
4
>>>

```

需要谨记，字典并不是根据键来进行有序维护的。第一个添加的键-值对 ('Utah':'SaltLakeCity') 被放在了字典的第一位，第二个添加的键-值对 ('California':'Sacramento') 则被放在了最后。键的位置是由散列来决定的，第 5 章会详细介绍散列。以上示例也说明，len 函数对字典的功能与对其他集合的功能相同。

字典既有运算符，又有方法。表 1-7 和表 1-8 分别展示了它们。keys、values 和 items 方法均会返回包含相应值的对象。可以使用 list 函数将字典转换成列表。在表 1-8 中可以看到，get 方法有两种版本。如果键没有出现在字典中，get 会返回 None。然而，第二个可选参数可以返回特定值。

表 1-7 Python 字典支持的运算

运 算 名	运 算 符	解 释
[]	myDict[k]	返回与 k 相关联的值，如果没有则报错
in	key in adict	如果 key 在字典中，返回 True，否则返回 False
del	del adict[key]	从字典中删除 key 的键-值对

表 1-8 Python 字典提供的方法

方 法 名	用 法	解 释
keys	adict.keys()	返回包含字典中所有键的 dict_keys 对象
values	adict.values()	返回包含字典中所有值的 dict_values 对象
items	adict.items()	返回包含字典中所有键-值对的 dict_items 对象
get	adict.get(k)	返回 k 对应的值，如果没有则返回 None
get	adict.get(k, alt)	返回 k 对应的值，如果没有则返回 alt

```

>>> phoneext={'david':1410, 'brad':1137}
>>> phoneext
{'brad':1137, 'david':1410}
>>> phoneext.keys()
dict_keys(['brad', 'david'])
>>> list(phoneext.keys())
['brad', 'david']
>>> phoneext.values()
dict_values([1137, 1410])
>>> list(phoneext.values())
[1137, 1410]
>>> phoneext.items()
dict_items([('brad', 1137), ('david', 1410)])
>>> list(phoneext.items())
[('brad', 1137), ('david', 1410)]

```

```
>>> phoneext.get("kent")
>>> phoneext.get("kent", "NO ENTRY")
'NO ENTRY'
>>>
```

1.4.2 输入与输出

程序经常需要与用户进行交互，以获得数据或者提供某种结果。目前的大多数程序使用对话框作为要求用户提供某种输入的方式。尽管 Python 确实有方法来创建这样的对话框，但是可以利用更简单的函数。Python 提供了一个函数，它使得我们可以要求用户输入数据并且返回一个字符串的引用。这个函数就是 `input`。

`input` 函数接受一个字符串作为参数。由于该字符串包含有用的文本来提示用户输入，因此它经常被称为 **提示字符串**。举例来说，可以像下面这样调用 `input`。

```
aName = input('Please enter your name: ')
```

不论用户在提示字符串后面输入什么内容，都会被存储在 `aName` 变量中。使用 `input` 函数，可以非常简便地写出程序，让用户输入数据，然后再对这些数据进行进一步处理。例如，在下面的两条语句中，第一条要求用户输入姓名，第二条则打印出对输入字符串进行一些简单处理后的结果。

```
aName = input("Please enter your name ")
print("Your name in all capitals is ", aName.upper(),
      "and has length", len(aName))
```

需要注意的是，`input` 函数返回的值是一个字符串，它包含用户在提示字符串后面输入的所有字符。如果需要将这个字符串转换成其他类型，必须明确地提供类型转换。在下面的语句中，用户输入的字符串被转换成了浮点数，以便于后续的算术处理。

```
sradius = input("Please enter the radius of the circle ")
radius = float(sradius)
diameter = 2 * radius
```

格式化字符串

`print` 函数为输出 Python 程序的值提供了一种非常简便的方法。它接受零个或者多个参数，并且将单个空格作为默认分隔符来显示结果。通过设置 `sep` 这一实际参数可以改变分隔符。此外，每一次打印都默认以换行符结尾。这一行为可以通过设置实际参数 `end` 来更改。下面是一些例子。

```
>>> print("Hello")
Hello
>>> print("Hello", "World")
Hello World
>>> print("Hello", "World", sep="***")
Hello***World
>>> print("Hello", "World", end="***")
Hello World***>>>
```

更多地控制程序的输出格式经常十分有用。幸运的是，Python 提供了另一种叫作格式化字符串的方式。格式化字符串是一个模板，其中包含保持不变的单词或空格，以及之后插入的变量的占位符。例如，下面的语句包含 `is` 和 `years old.`，但是名字和年龄会根据运行时变量的值而发生改变。

```
print(aName, "is", age, "years old.")
```

使用格式化字符串，可以将上面的语句重写成下面的语句。

```
print("%s is %d years old." % (aName, age))
```

这个简单的例子展示了一个新的字符串表达式。`%` 是字符串运算符，被称作格式化运算符。表达式的左边部分是模板（也叫格式化字符串），右边部分则是一系列用于格式化字符串的值。需要注意的是，右边的值的个数与格式化字符串中`%`的个数一致。这些值将依次从左到右地被换入格式化字符串。

让我们更进一步地观察这个格式化表达式的左右两部分。格式化字符串可以包含一个或者多个转换声明。转换字符告诉格式化运算符，什么类型的值会被插入到字符串中的相应位置。在上面的例子中，`%s` 声明了一个字符串，`%d` 则声明了一个整数。其他可能的类型声明还包括 `i`、`u`、`f`、`e`、`g`、`c` 和`%`。表 1-9 总结了所有的类型声明。

表 1-9 格式化字符串可用的类型声明

字 符	输出格式
d、i	整数
u	无符号整数
f	m.dddd 格式的浮点数
e	m.ddde+/-xx 格式的浮点数
E	m.dddE+/-xx 格式的浮点数
g	对指数小于-4 或者大于 5 的使用 <code>%e</code> ，否则使用 <code>%f</code>
c	单个字符
s	字符串，或者任意可以通过 <code>str</code> 函数转换成字符串的 Python 数据对象
%	插入一个常量%符号

可以在`%`和格式化字符之间加入一个格式化修改符。格式化修改符可以根据给定的宽度对值进行左对齐或者右对齐，也可以通过小数点之后的一些数字来指定宽度。表 1-10 解释了这些格式化修改符。

表 1-10 格式化修改符

修 改 符	例 子	解 释
数字	%20d	将值放在 20 个字符宽的区域中
-	%-20d	将值放在 20 个字符宽的区域中，并且左对齐
+	%+20d	将值放在 20 个字符宽的区域中，并且右对齐
0	%020d	将值放在 20 个字符宽的区域中，并在前面补上 0
.	%20.2f	将值放在 20 个字符宽的区域中，并且保留小数点后 2 位
(name)	%(name)d	从字典中获取 name 键对应的值

格式化运算符的右边是将被插入格式化字符串的一些值。这个集合可以是元组或者字典。如果这个集合是元组，那么值就根据位置次序被插入。也就是说，元组中的第一个元素对应于格式化字符串中的第一个格式化字符。如果这个集合是字典，那么值就根据它们对应的键被插入，并且所有的格式化字符必须使用 (name) 修改符来指定键名。

```
>>> price = 24
>>> item = "banana"
>>> print("The %s costs %d cents" % (item,price))
The banana costs 24 cents
>>> print("The %+10s costs %.2f cents" % (item,price))
The      banana costs    24.00 cents
>>> print("The %+10s costs %10.2f cents" % (item,price))
The      banana costs      24.00 cents
>>> itemdict = {"item":"banana","cost":24}
>>> print("The %(item)s costs %(cost)7.1f cents" % itemdict)
The banana costs      24.0 cents
>>>
```

除了格式化字符串可以使用格式化字符和修改符之外，Python 的字符串还包含了一个 `format` 方法。该方法可以与新的 `Formatter` 类结合起来使用，从而实现复杂字符串的格式化。可以在 Python 参考手册中找到更多关于这些特性的内容。

1.4.3 控制结构

正如前文所述，算法需要两个重要的控制结构：迭代和分支。Python 通过多种方式支持这两种控制结构。程序员可以根据需要选择最有效的结构。

对于迭代，Python 提供了标准的 `while` 语句以及非常强大的 `for` 语句。`while` 语句会在给定条件为真时重复执行一段代码，如下所示。

```
>>> counter = 1
>>> while counter <= 5:
...     print("Hello, world")
...     counter = counter + 1
...
Hello, world
```

```
Hello, world  
Hello, world  
Hello, world  
Hello, world
```

这段代码将“Hello, world”打印了5遍。Python会在每次重复执行前计算while语句中的条件表达式。由于Python本身要求强制缩进，因此可以非常容易地看清楚while语句的结构。

while语句是非常普遍的迭代结构，我们在很多不同的算法中都会用到它。在很多情况下，迭代过程由复合条件来控制。

```
while counter <= 10 and not done:
```

在这个例子中，迭代语句只有在上面两个条件都满足的情况下才会被执行。变量counter的值需要小于或等于10，并且变量done的值需要为False(not False就是True)，因此True and True的最后结果才是True。

while语句在众多情况下都非常有用，另一个迭代结构for语句则可以很好地和Python的各种集合结合在一起使用。for语句可以用于遍历一个序列集合的每个成员，如下所示。

```
>>> for item in [1,3,6,2,5]:  
...     print(item)  
...  
1  
3  
6  
2  
5
```

for语句将列表[1,3,6,2,5]中的每一个值依次赋给变量item。然后，迭代语句就会被执行。这种做法对任意的序列集合（列表、元组以及字符串）都有效。

for语句的一个常见用法是在一定的值范围内进行有限次数的迭代。下面的语句会执行print函数5次。range函数会返回一个包含序列0、1、2、3、4的范围对象，然后每个值都会被赋给变量item。接着，Python会计算该值的平方并且打印结果。

```
>>> for item in range(5):  
...     print(item**2)  
...  
0  
1  
4  
9  
16  
>>>
```

for语句的另一个非常有用的使用场景是处理字符串中的每一个字符。下面的代码段遍历一个字符串列表，并且将每一个字符串中的每一个字符都添加到结果列表中。最终的结果就是一个包含所有字符串的所有字符的列表。

```
>>> wordlist = ['cat', 'dog', 'rabbit']
>>> letterlist = []
>>> for aword in wordlist:
...     for aletter in aword:
...         letterlist.append(aletter)
...
>>> letterlist
['c', 'a', 't', 'd', 'o', 'g', 'r', 'a', 'b', 'b', 'i', 't']
>>>
```

分支语句允许程序员进行询问，然后根据结果，采取不同的行动。绝大多数的编程语言都提供两种有用的分支结构：`if else` 和 `if`。以下是使用 `if else` 语句的一个简单的二元分支示例。

```
if n < 0:
    print("Sorry, value is negative")
else:
    print(math.sqrt(n))
```

在这个例子中，Python 会检查 `n` 所指向的对象是否小于 0。如果是，就会打印一条消息，说明它是负值；如果不是，就会执行 `else` 分支来计算它的平方根。

和其他所有控制结构一样，分支结构支持嵌套，一个问题的结果能帮助决定是否需要继续问下一个问题。例如，假设 `score` 是指向计算机科学考试分数的变量。

```
if score >= 90:
    print('A')
else:
    if score >= 80:
        print('B')
    else
        if score >= 70:
            print('C')
        else:
            if score >= 60:
                print('D')
            else:
                print('F')
```

这一代码段通过打印字母等级来对变量 `score` 进行分类。如果分数大于或等于 90，这一语句会打印 A；如果小于 90（`else`），会接着问下一个问题。如果分数大于或等于 80，因为小于 90，所以它一定介于 80 和 89 之间，那么语句就会打印 B。可以发现，Python 的缩进模式帮助我们在不需要额外语法元素的情况下有效地关联对应的 `if` 和 `else`。

另一种表达嵌套分支的语法是使用 `elif` 关键字。将 `else` 和下一个 `if` 结合起来，可以减少额外的嵌套层次。注意，最后的 `else` 仍然是必需的，它用来在所有分支条件都不满足的情况下提供默认分支。

```
if score >= 90:
    print('A')
elif score >= 80:
    print('B')
```

```

elif score >= 70:
    print('C')
elif score >= 60:
    print('D')
else:
    print('F')

```

Python 也有单路分支结构，即 `if` 语句。如果条件为真，就会执行相应的代码。如果条件为假，程序会跳过 `if` 语句，执行下面的语句。例如，下面的代码段会首先检查变量 `n` 的值是否为负。如果值为负，那么就取它的绝对值，再计算它的平方根。

```

if n < 0:
    n = abs(n)
print(math.sqrt(n))

```

列表可以通过使用迭代结构和分支结构来创建。这种方式被称为**列表解析式**。通过列表解析式，可以根据一些处理和分支标准轻松创建列表。举例来说，如果想创建一个包含前 10 个完全平方数的列表，可以使用以下的 `for` 语句。

```

>>> sqlist = []
>>> for x in range(1,11):
...     sqlist.append(x*x)
>>> sqlist
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

```

使用列表解析式，只需一行代码即可创建完成。

```

>>> sqlist = [x*x for x in range(1,11)]
>>> sqlist
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

```

变量 `x` 会依次取由 `for` 语句指定的 1 到 10 为值。之后，计算 `x*x` 的值并将结果添加到正在构建的列表中。列表解析式也允许添加一个分支语句来控制添加到列表中的元素，如下所示。

```

>>> sqlist = [x*x for x in range(1,11) if x%2 != 0]
>>> sqlist
[1, 9, 25, 49, 81]

```

这一列表解析式构建的列表只包含 1 到 10 中奇数的平方数。任意支持迭代的序列都可用于列表解析式。

```

>>>[ch.upper() for ch in 'comprehension' if ch not in 'aeiou']
['C', 'M', 'P', 'R', 'H', 'N', 'S', 'N']

```

1.4.4 异常处理

在编写程序时通常会遇到两种错误。第一种是语法错误，也就是说，程序员在编写语句或者

表达式时出错。例如，在写 for 语句时忘记加冒号。

```
>>> for i in range(10)
SyntaxError: invalid syntax (<pyshell#61>, line 1)
```

在这个例子中，Python 解释器发现，由于语句不符合 Python 语法规规范，因此它无法执行这条指令。初学者经常会犯语法错误。

第二种是逻辑错误，即程序能执行完成但返回了错误的结果。这可能是由于算法本身有错，或者程序员没有正确地实现算法。有时，逻辑错误会导致诸如除以 0、越界访问列表等非常严重的情况。这些逻辑错误会导致运行时错误，进而导致程序终止运行。通常，这些运行时错误被称为异常。

许多初级程序员简单地把异常等同于引起程序终止的严重运行时错误。然而，大多数编程语言都提供了让程序员能够处理这些错误的方法。此外，程序员也可以在检测到程序执行有问题的情况下自己创建异常。

当异常发生时，我们称程序“抛出”异常。可以用 try 语句来“处理”被抛出的异常。例如，以下代码段要求用户输入一个整数，然后从数学库中调用平方根函数。如果用户输入了一个大于或等于 0 的值，那么其平方根就会被打印出来。但是，如果用户输入了一个负数，平方根函数就会报告 ValueError 异常。

```
>>> anumber = int(input("Please enter an integer "))
Please enter an integer -23
>>> print(math.sqrt(anumber))
Traceback (most recent call last):
  File "<pyshell#102>", line 1, in <module>
    print(math.sqrt(anumber))
  ValueError: math domain error
>>>
```

可以在 try 语句块中调用 print 函数来处理这个异常。对应的 except 语句块“捕捉”到这个异常，并且为用户打印一条提示消息。

```
>>> try:
    print(math.sqrt(anumber))
except:
    print("Bad Value for square root")
    print("Using absolute value instead")
    print(math.sqrt(abs(anumber)))

Bad Value for square root
Using absolute value instead
4.79583152331
>>>
```

except 会捕捉到 sqrt 抛出的异常并打印提示消息，然后会使用对应数字的绝对值来保证 sqrt 的参数非负。这意味着程序并不会终止，而是继续执行后续语句。

程序员也可以使用 `raise` 语句来触发运行时异常。例如，可以先检查值是否为负，并在值为负时抛出异常，而不是给 `sqrt` 函数提供负数。下面的代码段显示了创建新的 `RuntimeError` 异常的结果。注意，程序仍然会终止，但是导致其终止的异常是由我们自己手动创建的。

```
>>> if anumber < 0:
...     raise RuntimeError("You can't use a negative number")
... else:
...     print(math.sqrt(anumber))
...
Traceback (most recent call last):
File "<stdin>", line 2, in <module>
RuntimeError: You can't use a negative number
>>>
```

除了 `RuntimeError` 以外，还可以抛出很多不同类型的异常。请查看 Python 参考手册，了解完整的异常类型以及如何自己创建异常。

1.4.5 定义函数

之前的过程抽象例子调用了 Python 数学模块中的 `sqrt` 函数来计算平方根。通常来说，可以通过定义函数来隐藏任何计算的细节。函数的定义需要一个函数名、一系列参数以及一个函数体。函数也可以显式地返回一个值。例如，下面定义的简单函数会返回传入值的平方。

```
>>> def square(n):
...     return n**2
...
>>> square(3)
9
>>> square(square(3))
81
>>>
```

这个函数定义包含函数名 `square` 以及一个括号包含的形式参数列表。在这个函数中，`n` 是唯一的形式参数，这意味着 `square` 只需要一份数据就能完成任务。计算 `n**2` 并返回结果的细节被隐藏起来。如果要调用 `square` 函数，可以为其提供一个实际参数值（在本例中是 3），并要求 Python 环境计算。注意，`square` 函数的返回值可以作为参数传递给另一个函数调用。

通过运用著名的牛顿迭代法，可以自己实现平方根函数。用于近似求解平方根的牛顿迭代法使用迭代计算的方法来求解正确的结果。

$$\text{newguess} = \frac{1}{2} \times (\text{oldguess} + \frac{n}{\text{oldguess}})$$

以上公式接受一个值 `n`，并且通过在每一次迭代中将 `newguess` 赋值给 `oldguess` 来反复猜测平方根。初次猜测的平方根是 `n/2`。代码清单 1-1 展示了该函数的定义，它接受值 `n` 并且返回 20 轮迭代之后的 `n` 的平方根。牛顿迭代法的细节都被隐藏在函数定义之中，用户不需要知道任何实现细节就可以调用该函数来求解平方根。代码清单 1-1 同时也展示了#的用法。任何跟在#之后一行

内的字符都是注释。Python 解释器不会执行这些注释。

代码清单 1-1 通过牛顿迭代法求解平方根

```
1 def squareroot(n):
2     root = n/2 #initial guess will be 1/2 of n
3     for k in range(20):
4         root = (1/2)*(root + (n / root))
5
6     return root

>>> squareroot(9)
3.0
>>> squareroot(4563)
67.549981495186216
>>>
```

1.4.6 Python 面向对象编程：定义类

前文说过，Python 是一门面向对象的编程语言。到目前为止，我们已经使用了一些内建的类来展示数据和控制结构的例子。面向对象编程语言最强大的一项特性是允许程序员（问题求解者）创建全新的类来对求解问题所需的数据进行建模。

我们之前使用了抽象数据类型来对数据对象的状态及行为进行逻辑描述。通过构建能实现抽象数据类型的类，可以利用抽象过程，同时为真正在程序中运用抽象提供必要的细节。每当需要实现抽象数据类型时，就可以创建新类。

1. Fraction 类

要展示如何实现用户定义的类，一个常用的例子是构建实现抽象数据类型 `Fraction` 的类。我们已经看到，Python 提供了很多数值类。但是在有些时候，需要创建“看上去很像”分数的数据对象。

像 $\frac{3}{5}$ 这样的分数由两部分组成。上面的值称作分子，可以是任意整数。下面的值称作分母，可以是任意大于 0 的整数（负的分数带有负的分子）。尽管可以用浮点数来近似表示分数，但我们在此次希望能精确表示分数的值。

`Fraction` 对象的表现应与其他数值类型一样。我们可以针对分数进行加、减、乘、除等运算，也能够使用标准的斜线形式来显示分数，比如 $3/5$ 。此外，所有的分数方法都应该返回结果的最简形式。这样一来，不论进行何种运算，最后的结果都是最简分数。

在 Python 中定义新类的做法是，提供一个类名以及一整套与函数定义语法类似的方法定义。以下是一个方法定义框架。

```
class Fraction:
    # 方法定义
```

所有类都应该首先提供构造方法。构造方法定义了数据对象的创建方式。要创建一个 Fraction 对象，需要提供分子和分母两部分数据。在 Python 中，构造方法总是命名为`__init__`（即在 `init` 的前后分别有两个下划线），如代码清单 1-2 所示。

代码清单 1-2 Fraction 类及其构造方法

```
1 class Fraction:
2
3     def __init__(self, top, bottom):
4
5         self.num = top
6         self.den = bottom
```

注意，形式参数列表包含 3 项。`self` 是一个总是指向对象本身的特殊参数，它必须是第一个形式参数。然而，在调用方法时，从来不需要提供相应的实际参数。如前所述，分数需要分子与分母两部分状态数据。构造方法中的 `self.num` 定义了 Fraction 对象有一个叫作 `num` 的内部数据对象作为其状态的一部分。同理，`self.den` 定义了分母。这两个实际参数的值在初始时赋给了状态，使得新创建的 Fraction 对象能够知道其初始值。

要创建 Fraction 类的实例，必须调用构造方法。使用类名并且传入状态的实际值就能完成调用（注意，不要直接调用`__init__`）。

```
myfraction = Fraction(3,5)
```

以上代码创建了一个对象，名为 `myfraction`，值为 $3/5$ 。图 1-5 展示了这个对象。

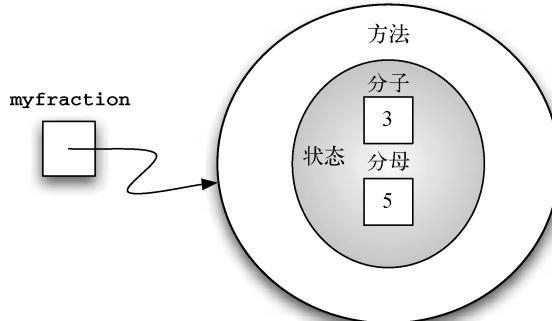


图 1-5 Fraction 类的一个实例

接下来实现这一抽象数据类型所需要的行为。考虑一下，如果试图打印 Fraction 对象，会发生什么呢？

```
>>> myf = Fraction(3,5)
>>> print(myf)
<__main__.Fraction instance at 0x409b1acc>
```

Fraction 对象 myf 并不知道如何响应打印请求。print 函数要求对象将自己转换成一个可以被写到输出端的字符串。myf 唯一能做的就是显示存储在变量中的实际引用（地址本身）。这不是我们想要的结果。

有两种办法可以解决这个问题。一种是定义一个 show 方法，使得 Fraction 对象能够将自己作为字符串来打印。代码清单 1-3 展示了该方法的实现细节。如果像之前那样创建一个 Fraction 对象，可以要求它显示自己（或者说，用合适的格式将自己打印出来）。不幸的是，这种方法并不通用。为了能正确打印，我们需要告诉 Fraction 类如何将自己转换成字符串。要完成任务，这是 print 函数所必需的。

代码清单 1-3 show 方法

```
1 def show(self):
2     print(self.num, "/", self.den)
```

```
>>> myf = Fraction(3,5)

>>> myf.show()
3/5
>>> print(myf)
<__main__.Fraction instance at 0x40bce9ac>
>>>
```

Python 的所有类都提供了一套标准方法，但是可能没有正常工作。其中之一就是将对象转换成字符串的方法 `__str__`。这个方法的默认实现是像我们之前所见的那样返回实例的地址字符串。我们需要做的是为这个方法提供一个“更好”的实现，即重写默认实现，或者说重新定义该方法的行为。

为了达到这一目标，仅需定义一个名为 `__str__` 的方法，并且提供新的实现，如代码清单 1-4 所示。除了特殊参数 `self` 之外，该方法定义不需要其他信息。新的方法通过将两部分内部状态数据转换成字符串并在它们之间插入字符 / 来将分数对象转换成字符串。一旦要求 Fraction 对象转换成字符串，就会返回结果。注意该方法的各种用法。

代码清单 1-4 __str__ 方法

```
1 def __str__(self):
2     return str(self.num) + "/" + str(self.den)
```

```
>>> myf = Fraction(3,5)
>>> print(myf)
3/5
>>> print("I ate", myf, "of the pizza")
I ate 3/5 of the pizza
>>> myf.__str__()
```

```
'3/5'
>>> str(myf)
'3/5'
>>>
```

可以重写 Fraction 类中的很多其他方法，其中最重要的一些是基本的数学运算。我们想创建两个 Fraction 对象，然后将它们相加。目前，如果试图将两个分数相加，会得到下面的结果。

```
>>> f1 = Fraction(1,4)
>>> f2 = Fraction(1,2)
>>> f1+f2

Traceback (most recent call last):
  File "<pyshell#173>", line 1, in <toplevel-
    f1+f2
TypeError: unsupported operand type(s) for +:
  'instance' and 'instance'
>>>
```

如果仔细研究这个错误，会发现加号+无法处理 Fraction 的操作数。

可以通过重写 Fraction 类的`__add__`方法来修正这个错误。该方法需要两个参数。第一个仍然是 `self`，第二个代表了表达式中的另一个操作数。

```
f1.__add__(f2)
```

以上代码会要求 Fraction 对象 `f1` 将 Fraction 对象 `f2` 加到自己的值上。可以将其写成标准表达式：`f1 + f2`。

两个分数需要有相同的分母才能相加。确保分母相同最简单的方法是使用两个分母的乘积作为分母。

$$\frac{a}{b} + \frac{c}{d} = \frac{ad}{bd} + \frac{cb}{bd} = \frac{ad + cb}{bd}$$

代码清单 1-5 展示了具体实现。`__add__`方法返回一个包含分子和分母的新 Fraction 对象。可以利用这一方法来编写标准的分数数学表达式，将加法结果赋给变量，并且打印结果。值得注意的是，第 3 行中的\称作续行符。当一条 Python 语句被分成多行时，需要用到续行符。

代码清单 1-5 `__add__`方法

```
1  def __add__(self, otherfraction):
2
3      newnum = self.num * otherfraction.den + \
4              self.den * otherfraction.num
5      newden = self.den * otherfraction.den
6
7      return Fraction(newnum, newden)
```

```
>>> f1 = Fraction(1, 4)
>>> f2 = Fraction(1, 2)
```

```
>>> f3 = f1 + f2
>>> print(f3)
6/8
>>>
```

虽然这一方法能够与我们预想的一样执行加法运算，但是还有一处可以改进。 $\frac{1}{4}+\frac{1}{2}$ 的确等于 $\frac{6}{8}$ ，但它并不是最简分数。最好的表达应该是 $\frac{3}{4}$ 。为了保证结果总是最简分数，需要一个知道如何化简分数的辅助方法。该方法需要寻找分子和分母的最大公因数（greatest common divisor，GCD），然后将分子和分母分别除以最大公因数，最后的结果就是最简分数。

要寻找最大公因数，最著名的方法就是欧几里得算法，第8章将详细讨论。欧几里得算法指出，对于整数 m 和 n ，如果 m 能被 n 整除，那么它们的最大公因数就是 n 。然而，如果 m 不能被 n 整除，那么结果是 n 与 m 除以 n 的余数的最大公因数。代码清单1-6提供了一个迭代实现。注意，这种实现只有在分母为正的时候才有效。对于Fraction类，这是可以接受的，因为之前已经定义过，负的分数带有负的分子，其分母为正。

代码清单1-6 gcd函数

```
1 def gcd(m,n):
2     while m%n != 0:
3         oldm = m
4         oldn = n
5
6         m = oldn
7         n = oldm%oldn
8     return n
```

现在可以利用这个函数来化简分数。为了将一个分数转化成最简形式，需要将分子和分母都除以它们的最大公因数。对于分数 $\frac{6}{8}$ ，最大公因数是2。因此，将分子和分母都除以2，便得到 $\frac{3}{4}$ 。代码清单1-7展示了实现细节。

代码清单1-7 改良版__add__方法

```
1 def __add__(self, otherfraction):
2     newnum = self.num * otherfraction.den + \
3             self.den * otherfraction.num
4     newden = self.den * otherfraction.den
5     common = gcd(newnum, newden)
6     return Fraction(newnum//common, newden//common)
```

```
>>> f1 = Fraction(1,4)
>>> f2 = Fraction(1,2)
>>> f3 = f1 + f2
>>> print(f3)
3/4
>>>
```

Fraction对象现在有两个非常有用的方法，如图1-6所示。为了允许两个分数互相比较，

还需要添加一些方法。假设有两个 Fraction 对象，`f1` 和 `f2`。只有在它们是同一个对象的引用时，`f1 == f2` 才为 `True`。这被称为浅相等，如图 1-7 所示。在当前实现中，分子和分母相同的两个不同的对象是不相等的。

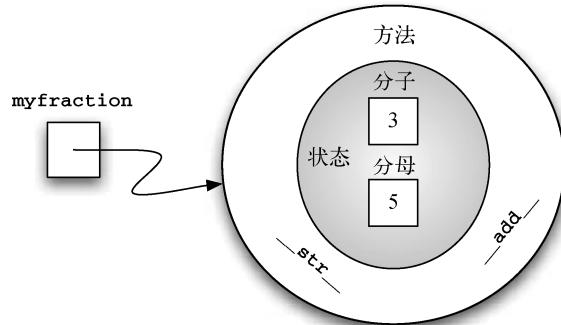


图 1-6 包含两个方法的 Fraction 实例

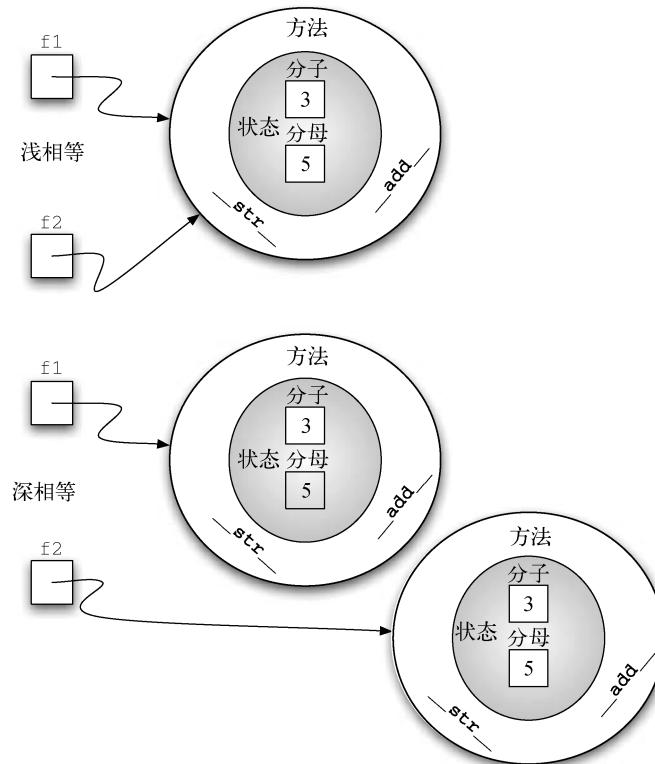


图 1-7 浅相等与深相等

通过重写`__eq__`方法，可以建立深相等——根据值来判断相等，而不是根据引用。`__eq__`是又一个在任意类中都有的标准方法。它比较两个对象，并且在它们的值相等时返回`True`，否则返回`False`。

在`Fraction`类中，可以通过统一两个分数的分母并比较分子来实现`__eq__`方法，如代码清单1-8所示。需要注意的是，其他的关系运算符也可以被重写。例如，`__le__`方法提供判断小于等于的功能。

代码清单1-8 `__eq__`方法

```
1 def __eq__(self, other):
2     firstnum = self.num * other.den
3     secondnum = other.num * self.den
4
5     return firstnum == secondnum
```

代码清单1-9提供了到目前为止`Fraction`类的完整实现。剩余的算术方法及关系方法留作练习。

代码清单1-9 `Fraction`类的完整实现

```
1 class Fraction:
2     def __init__(self, top, bottom):
3         self.num = top
4         self.den = bottom
5
6     def __str__(self):
7         return str(self.num) + "/" + str(self.den)
8
9     def show(self):
10        print(self.num, "/", self.den)
11
12    def __add__(self, otherfraction):
13        newnum = self.num * otherfraction.den + \
14                    self.den * otherfraction.num
15        newden = self.den * otherfraction.den
16        common = gcd(newnum, newden)
17        return Fraction(newnum//common, newden//common)
18
19    def __eq__(self, other):
20        firstnum = self.num * other.den
21        secondnum = other.num * self.den
22
23        return firstnum == secondnum
```

2. 继承：逻辑门与电路

最后一节介绍面向对象编程的另一个重要方面。继承使一个类与另一个类相关联，就像人们相互联系一样。孩子从父母那里继承了特征。与之类似，Python中的子类可以从父类继承特征数

据和行为。父类也称为超类。

图 1-8 展示了内建的 Python 集合类以及它们的相互关系。我们将这样的关系结构称为继承层次结构。举例来说，列表是有序集合的子。因此，我们将列表称为子，有序集合称为父（或者分别称为子类列表和超类序列）。这种关系通常被称为 IS-A 关系（IS-A 意即列表是一个有序集合）。这意味着，列表从有序集合继承了重要的特征，也就是内部数据的顺序以及诸如拼接、重复和索引等方法。

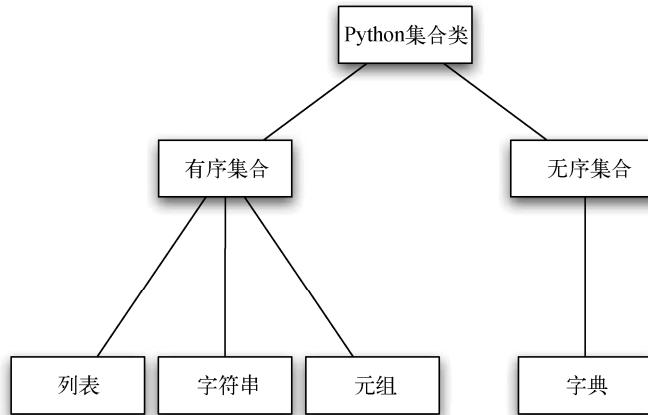


图 1-8 Python 集合类的继承层次结构

列表、字符串和元组都是有序集合。它们都继承了共同的数据组织和操作。不过，根据数据是否同类以及集合是否可修改，它们彼此又有区别。子类从父类继承共同的特征，但是通过额外的特征彼此区分。

通过将类组织成继承层次结构，面向对象编程语言使以前编写的代码得以扩展到新的应用场景中。此外，这种结构有助于更好地理解各种关系，从而更高效地构建抽象表示。

为了进一步探索这个概念，我们来构建一个模拟程序，用于模拟数字电路。逻辑门是这个模拟程序的基本构造单元，它们代表其输入和输出之间的布尔代数关系。一般来说，逻辑门都有单一的输出。输出值取决于提供的输入值。

与门（AND gate）有两个输入，每一个都是 0 或 1（分别代表 `False` 和 `True`）。如果两个输入都是 1，那么输出就是 1；如果至少有一个输入是 0，那么输出就是 0。或门（OR gate）同样也有两个输入。当至少有一个输入为 1 时，输出就为 1；当两个输入都是 0 时，输出是 0。

非门（NOT gate）与其他两种逻辑门不同，它只有一个输入。输出刚好与输入相反。如果输入是 0，输出就是 1。反之，如果输入是 1，输出就是 0。图 1-9 展示了每一种逻辑门的表示方法。每一种都有一张真值表，用于展示输入与输出的对应关系。

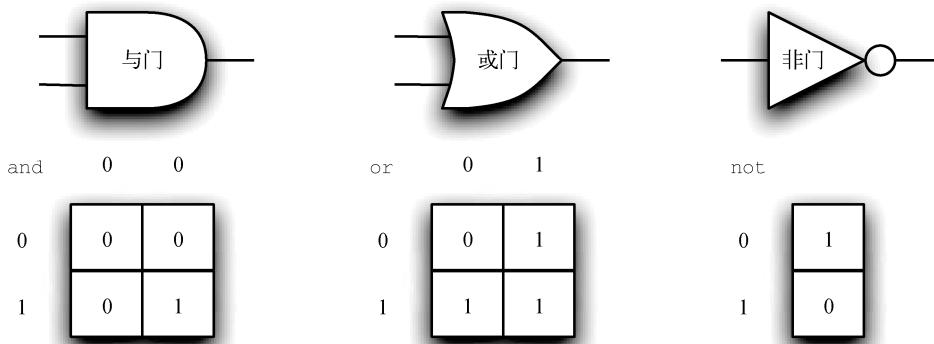


图 1-9 3 种逻辑门

通过不同的模式将这些逻辑门组合起来并提供一系列输入值，可以构建具有逻辑功能的电路。图 1-10 展示了一个包含两个与门、一个或门和一个非门的电路。两个与门的输出直接作为输入传给或门，然后其输出又输入给非门。如果在 4 个输入处（每个与门有两个输入）提供一系列值，那么非门就会输出结果。图 1-10 也展示了这一过程。

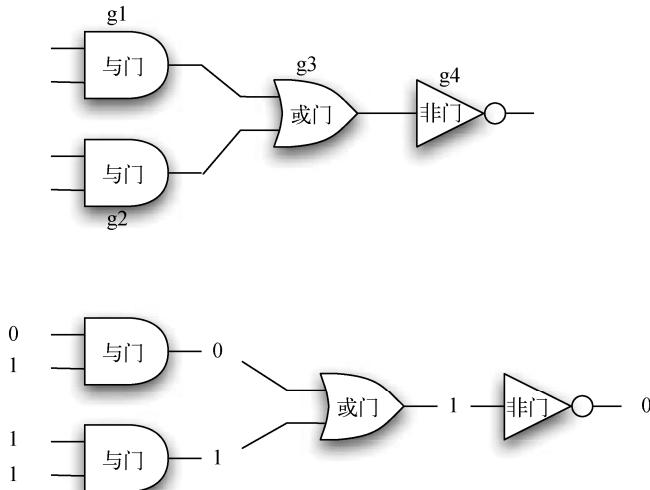


图 1-10 电路示例

为了实现电路，首先需要构建逻辑门的表示。可以轻松地将逻辑门组织成类的继承层次结构，如图 1-11 所示。顶部的 `LogicGate` 类代表逻辑门的通用特性：逻辑门的标签和一个输出。下面一层子类将逻辑门分成两种：有一个输入的逻辑门和有两个输入的逻辑门。再往下，就是具体的逻辑门。

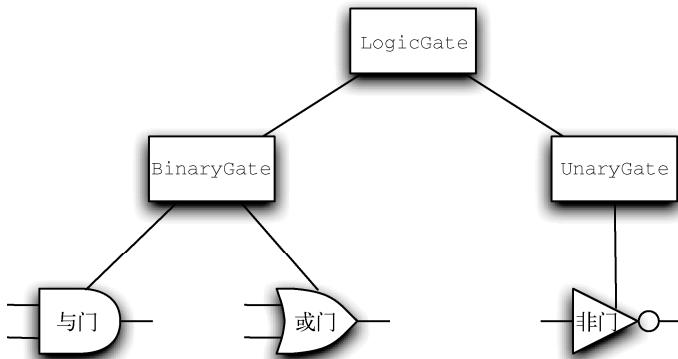


图 1-11 逻辑门的继承层次结构

现在开始通过实现最通用的类 `LogicGate` 来实现这些类。如前所述，每一个逻辑门都有一个用于识别的标签以及一个输出。此外，还需要一些方法，以便用户获取逻辑门的标签。

所有逻辑门还需要能够知道自己的输出值。这就要求逻辑门能够根据当前的输入值进行合理的逻辑运算。为了生成结果，逻辑门需要知道自己对应的逻辑运算是什么。这意味着需要调用一个方法来进行逻辑运算。代码清单 1-10 展示了 `LogicGate` 类的完整实现。

代码清单 1-10 超类 `LogicGate`

```

1  class LogicGate:
2
3      def __init__(self, n):
4          self.label = n
5          self.output = None
6
7      def getLabel(self):
8          return self.label
9
10     def getOutput(self):
11         self.output = self.performGateLogic()
12         return self.output
  
```

目前还不用实现 `performGateLogic` 函数。原因在于，我们不知道每一种逻辑门将如何进行自己的逻辑运算。这些细节会交由继承层次结构中的每一个逻辑门来实现。这是一种在面向对象编程中非常强大的思想——我们创建了一个方法，而其代码还不存在。参数 `self` 是指向实际调用方法的逻辑门对象的引用。任何添加到继承层次结构中的新逻辑门都仅需要实现之后会被调用的 `performGateLogic` 函数。一旦实现完成，逻辑门就可以提供运算结果。扩展已有的继承层次结构并提供使用新类所需的特定函数，这种能力对于重用代码来说非常重要。

我们依据输入的个数来为逻辑门分类。与门和或门有两个输入，非门只有一个输入。`BinaryGate` 是 `LogicGate` 的一个子类，并且有两个输入。`UnaryGate` 同样是 `LogicGate` 的

子类，但是仅有一个输入。在计算机电路设计中，这些输入被称作“引脚”(pin)，我们在实现中也使用这一术语。

代码清单1-11和代码清单1-12实现了这两个类。两个类中的构造方法首先使用super函数来调用其父类的构造方法。当创建BinaryGate类的实例时，首先要初始化所有从LogicGate中继承来的数据项，在这里就是逻辑门的标签。接着，构造方法添加两个输入(pinA和pinB)。这是在构建类继承层次结构时常用的模式。子类的构造方法需要先调用父类的构造方法，然后再初始化自己独有的数据。

代码清单1-11 BinaryGate类

```
1 class BinaryGate(LogicGate):
2
3     def __init__(self, n):
4         super().__init__(n)
5
6         self.pinA = None
7         self.pinB = None
8
9     def getPinA(self):
10        return int(input("Enter Pin A input for gate " + \
11                      self.getLabel() + "-->"))
12
13    def getPinB(self):
14        return int(input("Enter Pin B input for gate " + \
15                      self.getLabel() + "-->"))
```

代码清单1-12 UnaryGate类

```
1 class UnaryGate(LogicGate):
2
3     def __init__(self, n):
4         super().__init__(n)
5
6         self.pin = None
7
8     def getPin(self):
9         return int(input("Enter Pin input for gate " + \
10                      self.getLabel() + "-->"))
```

BinaryGate类增添的唯一行为就是取得两个输入值。由于这些值来自于外部，因此通过一条输入语句来要求用户提供。UnaryGate类也有类似的实现，不过它只有一个输入。

有了不同输入个数的逻辑门所对应的通用类之后，就可以为有独特行为的逻辑门构建类。例如，由于与门需要两个输入，因此AndGate是BinaryGate的子类。和之前一样，构造方法的第一行调用父类(BinaryGate)的构造方法，该构造方法又会调用它的父类(LogicGate)的构造方法。注意，由于继承了两个输入、一个输出和逻辑门标签，因此AndGate类并没有添加任何新的数据。

AndGate 类唯一需要添加的是布尔运算行为。这就是提供 `performGateLogic` 的地方。对于与门来说，`performGateLogic` 首先需要获取两个输入值，然后只有在它们都为 1 时返回 1。代码清单 1-13 展示了 AndGate 类的完整实现。

代码清单 1-13 AndGate 类

```

1  class AndGate(BinaryGate):
2
3      def __init__(self, n):
4          super().__init__(n)
5
6      def performGateLogic(self):
7
8          a = self.getPinA()
9          b = self.getPinB()
10         if a==1 and b==1:
11             return 1
12         else:
13             return 0

```

可以创建一个实例来验证 AndGate 类的行为。下面的代码展示了 `AndGate` 对象 `g1`，它有一个内部标签“G1”。当调用 `getOutput` 方法时，该对象必须首先调用它的 `performGateLogic` 方法，这个方法会获取两个输入值。一旦取得输入值，就会显示正确的结果。

```

>>> g1 = AndGate("G1")
>>> g1.getOutput()
Enter Pin A input for gate G1-->1
Enter Pin B input for gate G1-->0
0

```

或门和非门都能以相同的方式来构建。`OrGate` 也是 `BinaryGate` 的子类，`NotGate` 则会继承 `UnaryGate` 类。由于计算逻辑不同，这两个类都需要提供自己的 `performGateLogic` 函数。

要使用逻辑门，可以先构建这些类的实例，然后查询结果（这需要用户提供输入）。

```

>>> g2 = OrGate("G2")
>>> g2.getOutput()
Enter Pin A input for gate G2-->1
Enter Pin B input for gate G2-->1
1
>>> g2.getOutput()
Enter Pin A input for gate G2-->0
Enter Pin B input for gate G2-->0
0
>>> g3 = NotGate("G3")
>>> g3.getOutput()
Enter Pin input for gate G3-->0
1

```

有了基本的逻辑门之后，便可以开始构建电路。为此，需要将逻辑门连接在一起，前一个的输出是后一个的输入。为了做到这一点，我们要实现一个叫作 `Connector` 的新类。

Connector 类并不在逻辑门的继承层次结构中。但是，它会使用该结构，从而使每一个连接器的两端都有一个逻辑门（如图 1-12 所示）。这被称为 **HAS-A 关系**（HAS-A 意即“有一个”），它在面向对象编程中非常重要。前文用 IS-A 关系来描述子类与父类的关系，例如 UnaryGate 是一个 LogicGate。

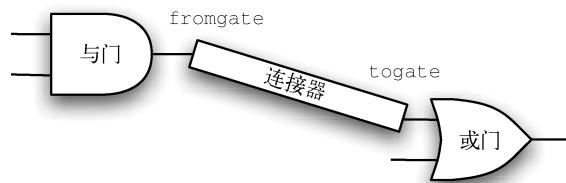


图 1-12 连接器将一个逻辑门的输出与另一个逻辑门的输入连接起来

Connector 与 LogicGate 是 HAS-A 关系。这意味着连接器内部包含 LogicGate 类的实例，但是不在继承层次结构中。在设计类时，区分 IS-A 关系（需要继承）和 HAS-A 关系（不需要继承）非常重要。

代码清单 1-14 展示了 Connector 类。每一个连接器对象都包含 fromgate 和 togate 两个逻辑门实例，数据值会从一个逻辑门的输出“流向”下一个逻辑门的输入。对 setNextPin 的调用（实现如代码清单 1-15 所示）对于建立连接来说非常重要。需要将这个方法添加到逻辑门类中，以使每一个 togate 能够选择适当的输入。

代码清单 1-14 Connector 类

```

1  class Connector:
2
3      def __init__(self, fgate, tgate):
4          self.fromgate = fgate
5          self.togate = tgate
6
7          tgate.setNextPin(self)
8
9      def getFrom(self):
10         return self.fromgate
11
12     def getTo(self):
13         return self.togate
  
```

代码清单 1-15 setNextPin 方法

```

1  def setNextPin(self, source):
2      if self.pinA == None:
3          self.pinA = source
4      else:
5          if self.pinB == None:
6              self.pinB = source
  
```

```

7         else:
8             raise RuntimeError("Error: NO EMPTY PINS")

```

在 `BinaryGate` 类中，逻辑门有两个输入，但连接器必须只连接其中一个。如果两个都能连接，那么默认选择 `pinA`。如果 `pinA` 已经有了连接，就选择 `pinB`。如果两个输入都已有连接，则无法连接逻辑门。

现在的输入来源有两个：外部以及上一个逻辑门的输出。这需要对方法 `getPinA` 和 `getPinB` 进行修改（请参考代码清单 1-16）。如果输入没有与任何逻辑门相连接（`None`），那就和之前一样要求用户输入。如果有连接，就访问该连接并且获取 `fromgate` 的输出值。这会触发 `fromgate` 处理其逻辑。该过程会一直持续，直到获取所有输入并且最终的输出值成为正在查询的逻辑门的输入。在某种意义上，这个电路反向工作，以获得所需的输入，再计算最后的结果。

代码清单 1-16 修改后的 `getPinA` 方法

```

1  def getPinA(self):
2      if self.pinA == None:
3          return input("Enter Pin A input for gate " + \
4                         self.getName() + "-->")
5      else:
6          return self.pinA.getFrom().getOutput()

```

下面的代码段构造了图 1-10 中的电路。

```

>>> g1 = AndGate("G1")
>>> g2 = AndGate("G2")
>>> g3 = OrGate("G3")
>>> g4 = NotGate("G4")
>>> c1 = Connector(g1, g3)
>>> c2 = Connector(g2, g3)
>>> c3 = Connector(g3, g4)

```

两个与门（`g1` 和 `g2`）的输出与或门（`g3`）的输入相连接，或门的输出又与非门（`g4`）的输入相连接。非门的输出就是整个电路的输出。

```

>>> g4.getOutput()
Pin A input for gate G1-->0
Pin B input for gate G1-->1
Pin A input for gate G2-->1
Pin B input for gate G2-->1
0
>>>

```

1.5 小结

- 计算机科学是研究如何解决问题的学科。
- 计算机科学利用抽象这一工具来表示过程和数据。
- 抽象数据类型通过隐藏数据的细节来使程序员能够管理问题的复杂度。

- Python 是一门强大、易用的面向对象编程语言。
- 列表、元组以及字符串是 Python 的内建有序集合。
- 字典和集是无序集合。
- 类使得程序员能够实现抽象数据类型。
- 程序员既可以重写标准方法，也可以构建新的方法。
- 类可以通过继承层次结构来组织。
- 类的构造方法总是先调用其父类的构造方法，然后才处理自己的数据和行为。

1.6 关键术语

HAS-A 关系	IS-A 关系	
编程	超类	<code>self</code>
抽象数据类型	独立于实现	抽象
方法	封装	对象
格式化字符串	过程抽象	格式化运算符
继承层次结构	接口	继承
可修改性	类	可计算
列表解析式	模拟	列表
深相等	数据抽象	浅相等
数据类型	算法	数据结构
信息隐藏	异常	提示符
子类	字典	真值表
		字符串

1.7 讨论题

1. 为校园里的人构建一个继承层次结构，包括教职员及学生。他们有何共同之处？又有何区别？
2. 为银行账户构建一个继承层次结构。
3. 为不同类型的计算机构建一个继承层次结构。
4. 利用本章提供的类，以交互方式构建一个电路并对其进行测试。

1.8 编程练习

1. 实现简单的方法 `getNum` 和 `getDen`，它们分别返回分数的分子和分母。
2. 如果所有分数从一开始就是最简形式会更好。修改 `Fraction` 类的构造方法，立即使用最大公因数来化简分数。注意，这意味着 `__add__` 不再需要化简结果。

3. 实现下列简单的算术运算：`__sub__`、`__mul__`和`__truediv__`。
4. 实现下列关系运算：`__gt__`、`__ge__`、`__lt__`、`__le__`和`__ne__`。
5. 修改 `Fraction` 类的构造方法，使其检查并确保分子和分母均为整数。如果任一不是整数，就抛出异常。
6. 我们假设负的分数是由负的分子和正的分母构成的。使用负的分母会导致某些关系运算符返回错误的结果。一般来说，这是多余的限制。请修改构造方法，使得用户能够传入负的分母，并且所有的运算符都能返回正确的结果。
7. 研究`__radd__`方法。它与`__add__`方法有何区别？何时应该使用它？请动手实现`__radd__`。
8. 研究`__iadd__`方法。它与`__add__`方法有何区别？何时应该使用它？请动手实现`__iadd__`。
9. 研究`__repr__`方法。它与`__str__`方法有何区别？何时应该使用它？请动手实现`__repr__`。
10. 研究其他类型的逻辑门（例如与非门、或非门、异或门）。将它们加入电路的继承层次结构。你需要额外添加多少代码？
11. 最简单的算术电路是半加器。研究简单的半加器电路并实现它。
12. 将半加器电路扩展为 8 位的全加器。
13. 本章展示的电路模拟是反向工作的。换句话说，给定一个电路，其输出结果是通过反向访问输入值来产生的，这会导致其他的输出值被反向查询。这个过程一直持续到外部输入值被找到，此时用户会被要求输入数值。修改当前的实现，使电路正向计算结果。当收到输入值的时候，电路就会生成输出结果。
14. 设计一个表示一张扑克牌的类，以及一个表示一副扑克牌的类。使用这两个类实现你最喜欢的扑克牌游戏。
15. 在报纸上找到一个数独游戏，并编写一个程序求解。



微信连接



回复“Python”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书



QQ连接

图灵读者官方群I: 218139230

图灵读者官方群II: 164939616

图灵社区
iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈

若把编写代码比作行军打仗，那么要想称霸沙场，不能仅靠手中的利刃，还需深谙兵法。Python是一把利刃，数据结构与算法则是兵法。只有熟读兵法，才能使利刃所向披靡。

本书作者在计算机科学领域深耕数十载，积累了丰富的实战经验。通过学习本书，你将掌握数据结构与算法的基本思想，从而有信心探索任何编程难题的解决方法。

- 使用Python实现栈、队列、列表等抽象数据类型
- 掌握大O记法和时间复杂度等概念
- 利用递归解决汉诺塔问题
- 实现常用的搜索算法和排序算法，并分析性能
- 掌握树与图在Python中的应用



FRANKLIN,
BEEDLE &
ASSOCIATES
INC.

图灵社区: iTuring.cn
热线: (010)51095183转600

分类建议 计算机/程序设计/Python

人民邮电出版社网址: www.ptpress.com.cn

ISBN 978-7-115-51721-0



ISBN 978-7-115-51721-0

定价: 79.00元

欢迎加入

图灵社区

最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要你有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、审读、评选等多种活动，赢取积分和银子，积累个人声望。

ituring.com.cn