TURING 图灵程序设计丛书

编程导论

[美]John V. Guttag◎著　梁杰◎译

Introduction to Computation and Programming Using Python

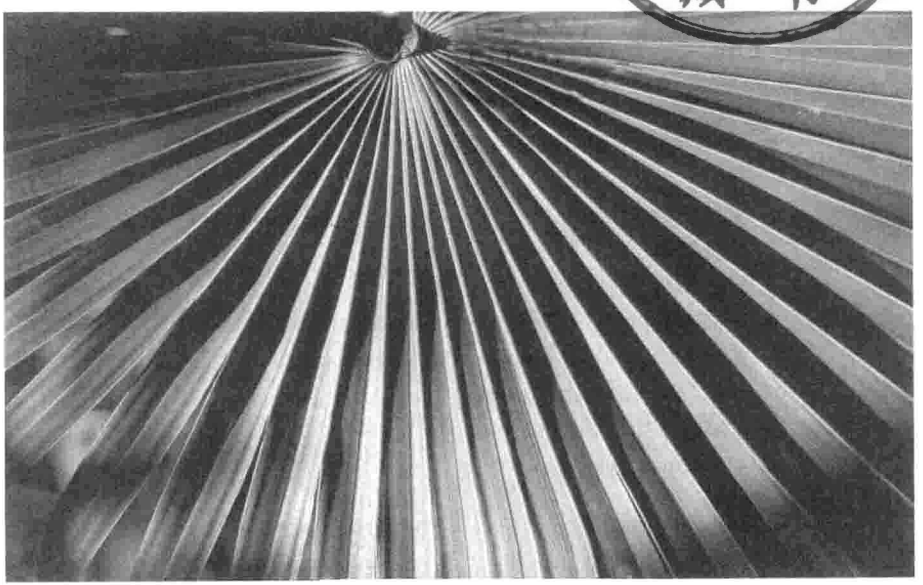
中国工信出版集团　人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

编程导论

Introduction to Computation and Programming Using Python

[美]John V. Guttag◎著　　梁杰◎译



人民邮电出版社
北　京

图书在版编目（CIP）数据

编程导论 /（美）谷泰格（Guttag, J. V.）著 ；梁杰译. -- 北京 ：人民邮电出版社，2015.4
（图灵程序设计丛书）
ISBN 978-7-115-38801-8

Ⅰ. ①编… Ⅱ. ①谷… ②梁… Ⅲ. ①软件工具－程序设计 Ⅳ. ①TP311.56

中国版本图书馆CIP数据核字(2015)第052955号

内 容 提 要

本书基于MIT（麻省理工学院）的一门课程写成，主要目标是帮助读者掌握并熟练使用各种计算技术。本书涵盖了Python的大部分特性，重点介绍如何使用Python这门语言，共包含编程基础、Python程序设计语言、理解计算的关键概念、计算问题的解决技术等四个方面。本书将Python语言特性和编程方法贯穿全书，目的是帮助读者在学习Python的同时掌握如何使用计算来解决有趣的问题。

本书适合那些对编程知之甚少却需要（或者想要）使用计算方法来解决问题的读者，是学习更高级计算机科学课程的基础。

◆ 著 [美] John V. Guttag
译 梁 杰
责任编辑 李松峰
执行编辑 李 静 李艳玲
责任印制 杨林杰

◆ 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
北京鑫正大印刷有限公司印刷

◆ 开本：800×1000 1/16
印张：17.75
字数：431千字 2015年4月第1版
印数：1-3 500册 2015年4月北京第1次印刷
著作权合同登记号 图字：01-2013-6321号

定价：59.00元

读者服务热线：(010)51095186转600 印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第0021号

版权声明

献给我的家人奥尔加、大卫、安德里亚、迈克尔、马克和艾迪。

中文版序

掌握多种不同的思维方式是每个人大学时代的必修课。事实上，高等院校要求学生学习多个学科的课程，其目的就是为了培养他们从多个角度思考问题的能力。在麻省理工学院，每个本科生都必须学习数学、物理、化学、生物以及人文、艺术和社会科学方面的课程，无论他的专业是什么。这不仅是为了让他们学到各学科的知识，更重要的是让他们学会像数学家、物理学家、化学家、生物学家和人文艺术学家一样思考。编程思维正迅速成为又一个重要的思维方式，因此也应该被每个毕业生掌握。如何从算法的角度思考问题？如何抽取问题的关键要素？如何将问题的解决方法模块化？如何估计解决方案的复杂度？如何分辨问题所属的类型并应用不同的技术加以解决？

在编程思维日益重要的大背景下，本书应运而生，成为想对编程思维一探究竟之人的必读之作。不管读者有什么背景，从事哪一行，打算如何应用计算技术，看这本书都没有错。在本书中，作者 John Guttag 全面介绍了编程思维，它与编程不同，是每个人都能够并且应该掌握的。之所以区分编程思维和编程有这么几个原因。编程确实是一项有价值的技能，但像计算机科学家一样思考则远远不止于掌握编程语言的语法和语义那么简单。（像计算机科学家一样思考，需要能够理解并解决科学、工程、金融以及其他领域的重大问题。）你还必须掌握通用的算法模式，面对问题时知道应该选择哪种算法工具，能够把问题抽象分解为既有方法能够解决的具体情况，明白何时以及如何忽略细节从而专注于解决问题的整体算法。这些技能源于对计算在实际中应用的观察，知道如何解决一个现实问题，就有了一个解决同类问题的模板。要具备编程思维，头脑中还必须拥有一个基础性理论性的框架，在这个框架之内，通晓种种算法分支的来龙去脉，在这个框架之上，可以变通出一般性的解决方案。

本书是掌握编程思维，拥有计算机科学家视野的理想门径。虽然它的前半部分重点介绍编程语言 Python，但着眼点却都是基础性的概念。作者通过演示编程语言在解决常见问题时的应用，通过抽象公共语言范式及分析它们的广泛影响，自然而然地讲解了编程语言的各个方面。本书后半部分全面示范了前半部分讲解的基础工具（二分查找、分治、生成与测试、穷举法等）在解决常见的、实际的科学和工程问题时的应用。特别地，读者将比较熟练地掌握分析数据、运行模拟、应用统计测试来确定结果相关性，以及将结果可视化等技能。这些技术在各自领域都有巨大的价

值。学习完本书，读者将掌握使用现代数据科学工具的基本技能，并有望将在本书中学到的编程思维应用到未来的各种挑战中。

Eric Grimson

麻省理工学院学术促进校监

计算机科学系教授

Preface to Introduction to Computation and Programming Using Python

Mastering different modes of thinking is an essential part of any college student's experience. Indeed, learning to consider a problem from different perspectives is a key motivation behind institutions of higher learning requiring that students take subjects that cover a breadth of disciplines. At MIT, every undergraduate must take classes in mathematics, physics, chemistry and biology as well as several subjects in the humanities, arts and social sciences, no matter what major they plan to pursue – not just for the knowledge they acquire, but because we want every student to be able to think like a mathematician, a physicist, a chemist, a biologist, a humanist. Computation is rapidly becoming another critical mode of thought, which should be a part of the armamentarium of any educated person – how to approach any problem from an algorithmic perspective, how to abstract key elements of a problem, how to modularize an approach to a problem, how to estimate the expected complexity of a proposed solution, how to recognize instances of common problem types and apply variants of known techniques to them.

With computational thinking emerging as a critical and highly relevant mode of thought, this book becomes an important addition to any inquisitive person's bookshelf, no matter what their background, or their profession, or their intended use of computational skills. In this book, John Guttag takes the thoughtful approach that computational thinking, as opposed to programming, is something that anyone can master, and should. I distinguish between computational thinking and programming for several reasons. Knowing how to program a computer is certainly a valuable skill. But there is much more to thinking like a computer scientist than simply knowing the syntax and semantics of a programming language (and being able to think like a computer scientist is essential to understanding how to tackle challenging problems in science, engineering, finance, and many other domains). One must also have mastery of common algorithmic patterns, be able to recognize when particular algorithmic tools are appropriate for tackling a problem, understand how to abstract components of a problem into instances with known solution methods, and recognize when and how to suppress details so as to focus on the overall algorithmic approach to a problem. Those skills come from seeing computation in action – applied to real world problems, and presented as a template for collections of problems. Those skills also

come from having a foundational, theoretic framework in which to understand the ramifications of algorithmic choices, and on which to generalize solutions.

This book wonderfully provides a setting for acquiring all of those skills and perspectives. While the first half of the book focuses on learning a programming language – in this instance, Python – it does so in a foundational way, by motivating different aspects of the language through application to common computational problems, and by abstracting common language paradigms and analyzing their broader impact. Equally importantly, the second half of the book then expands upon these fundamental tools (binary search, divide-and-conquer, generate-and-test, exhaustive enumeration, and others) by utilizing them in a wide range of common and practical scientific and engineering problems. In particular, readers will gain a level of comfort in analyzing data, in running simulations, in applying statistical tests to determine relevance of results, and in visualizing results – all tools of great value in many domains. At the end of this book, a reader will have a basic command of modern data science tools, and hopefully will have begun a life-long journey of using computational thinking whenever confronted with a new challenge.

Eric Grimson
Chancellor for Academic Advancement
Professor of Computer Science, MIT

前　言

本书基于 MIT 的一门课程写成。这门课从 2006 年起每年开课两次，主要针对那些对编程知之甚少却需要（或者想要）使用计算方法来解决问题的学生。每年都有少数学生在掌握这门课之后继续学习更高级的计算机科学课程，但对于大多数学生来说，这是他们唯一一门计算机科学课程。

正因为如此，相比深度我们更加注重广度。我们的目标是让学生初步了解大部分主题，当他们需要使用计算方法来实现目标时更容易想到可行的方法。也就是说，这不是一门“计算赏析”课程，而是一门充满挑战、要求严格的课程，需要学生投入很多时间和精力来让计算机按照他们的想法运行。

本书的主要目标是帮助你掌握并熟练使用各种计算技术。你要学会如何将理论计算模式应用到实际问题，以及如何使用计算方法从数据中提取信息。本书会向你展示解决计算问题的艺术。

内容结构比较独特。第一部分（第 1 章至第 8 章）介绍了如何使用 Python 编程。和其他教程不同的是，本书涉及四个方面：

- 编程基础；
- Python 程序设计语言；
- 理解计算的关键概念；
- 计算问题的解决技术。

本书涵盖了 Python 的大部分特性，但侧重点是如何使用这门语言，而不是语言本身。例如，前三章介绍了穷举的概念、猜测和验证算法、二分查找以及高效近似算法，但只涉及一小部分 Python 知识。我们将 Python 的特性贯穿全书，类似地，也将编程方法贯穿全书，目的是帮助你在学习 Python 的同时掌握如何使用计算来解决有趣的问题。

第二部分（第 9 章至第 16 章）主要介绍如何使用计算来解决问题。这部分知识不会超出高中线性代数的范围，但读者要有严谨的思维并且不会被数学概念吓到。这部分会介绍一些入门教材中常见的主题，比如计算复杂度和简单算法，同时也会介绍许多在入门教材中很少提到的主题，比如数据可视化、概率和统计思维、模拟模型以及使用计算来理解数据。

第三部分（第 17 章至第 19 章）介绍三个进阶内容——最优化问题、动态规划以及集群计算。

第一部分可以作为独立的基础课程讲授，大约需要 1/4 到 1/2 学期。我建议用整个学期来讲授本书的第一部分和第二部分。如果要包含第三部分，学生可能会感到吃力。

有两个主题贯穿全书：系统性地解决问题以及抽象能力。学完本书之后你能：

- ❑ 学会用 Python 这门语言来表示计算；
- ❑ 学会系统性地组织、编写和调试中等规模的程序；
- ❑ 理解计算复杂度；
- ❑ 将模棱两可的问题陈述转换为可计算的形式，从而解决问题；
- ❑ 学会一些实用的算法和问题简化技术；
- ❑ 学会用随机化和模拟来处理棘手的问题；
- ❑ 学会使用计算工具，包括用简单的统计和可视化工具对数据建模，以及理解数据的意义。

编程从本质上来说是一件很难的事。就像“几何无坦途”[①]一样，编程同样无坦途。你可以让学生完成一系列限制严格的“填空”编程题，让他们误以为自己已经学会了编程。但是，这并不能让学生掌握解决问题所必需的计算思维。

如果你真的想学懂这本书，只靠阅读是不够的，还要亲自动手运行书中的代码。书中的所有代码都可以在 http://mitpress.mit.edu/ICPPRE 上找到。从 2008 年开始，MIT 的开源课程网站上就有了这门课的多个版本。网站收录了课程的视频录像、一系列练习题以及考试。从 2012 年秋季开始，edX 和 MITx 提供了这门课的在线版。如果你真的想学懂这本书，我们强烈建议你完成这些网站上的练习题。

① 这是欧几里得在大约公元前 300 年对国王托勒密（Ptolemy）的回答，当时托勒密想找一条学习数学的捷径。

致　　谢

在 MIT 教授一门本科生课程时，我记了很多笔记，本书就脱胎于这些笔记。这门课以及这本书得益于我的教师同事（尤其是 Eric Grimson、Srinivas Devadas 和 Fredo Durand）、助教以及选修这门课的学生。

将课程笔记整理成书远比我想象的复杂。幸好，我的乐观态度一直支撑着我，因此我才没有放弃。家人和同事的鼓励也起到了非常大的作用。

Eric Grimson、Chris Terman 和 David Guttag 提供了非常重要的帮助。Eric 是 MIT 的执行校长，他挤出时间仔细读完了几乎整本书。他发现了很多错误（包括很多让我非常尴尬的技术性错误）并指出需要添加说明的地方。Chris 读了部分手稿并发现了一些错误，还帮我对付该死的 Microsoft Word，我们不得不用它来完成大部分工作。David 忍着对计算机科学的厌恶帮忙校对了后面的章节。他检查了我的数学内容并删改了我的废话。

本书的初稿用于 MIT 课程 6.00 以及 MITx 课程 6.00x。很多参与课程的学生指出了错误。一名 6.00x 的学生 J.C. Cabrejas 帮了大忙，他指出了许多拼写错误以及很多技术性错误。Cabrejas，谢谢你。

就像其他成功的教授一样，我很感谢我带的研究生。本书封底的照片是我在指导我现在的学生。然而，在研究室中，实际上是他们在帮助我。除了优秀的科研（这让我占了一些功劳）之外，Guha Balakrishnan、Joel Brooks、Ganeshapillai Gartheeban、Jen Gong、Yun Liu、 Anima Singh、Jenna Wiens 和 Amy Zhao 还对本书的手稿提出了许多有用的建议。

我非常感激编辑 Julie Sussman。和 Julie 合作之前，我根本不明白编辑的重要性。之前我也和很多能干的文字编辑一起合作过，我本以为那就是我需要的人，但是我错了。我需要我的合作者从学生的角度来阅读本书并告诉我，我必须做什么、我应该做什么以及如果我有时间和精力的话可以做什么。Julie 提出了很多有用的建议，不容忽视。她既精通英语又懂编程真是太了不起了。

最后，感谢我的妻子 Olga，她一直督促着我并在关键时刻提供帮助。

目　录

第1章 起　步

计算机只能做两件事，执行计算并记录计算的结果，但是它把这两件事完成得非常漂亮。常见的台式机和笔记本电脑每秒钟可以执行近10亿次计算，快得难以置信。想象一下，让一个球从1米高自由落体掉到地板上，就在这段时间内，你的计算机可能已经执行了10亿条指令。从内存角度来说，一台普通的计算机可能有几百GB（gigabyte，1 gigabyte是10亿字节）。几百GB到底有多大呢？如果一字节（byte，1字节等于8比特，用来表示一个字符）重一盎司（实际上当然没有这么重），那么100 GB总重量会超过300万吨，几乎和美国一年的煤炭产量相同。

在漫长的人类历史上，计算的速度受限于人脑的计算速度以及书写结果的速度。这也就是说人们只能计算最小的问题。即使现代计算机的速度已经如此之快，仍然有很多问题无法计算（例如，气候变化），但是越来越多的问题是可以通过计算来解决的。我们希望当你学完本书时，可以使用计算思维来解决工作、学习或者日常生活中遇到的大部分问题。

那么，到底什么是计算思维呢？

所有的知识都可以归结为两类：陈述性知识和指令性知识。*陈述性知识*是由事实陈述组成的。例如，“当$y*y=x$时，x的平方根是数字y”，这是一个事实的陈述。可惜，我们无法得知如何计算平方根。

*指令性知识*表达的是“怎么做”，或者说演绎信息的方法。亚历山大里亚的希罗（Heron of Alexandria）是第一个记录如何计算一个数的平方根的人。[①]他的方法可以总结成这样。

随机选择一个数g。如果$g*g$和x足够接近，停止并宣布g是答案；否则选择一个新的数，取g和x/g的平均值，也就是$(g+x/g)/2$。使用新选择的数来代替g并重复这个过程，直到$g*g$和x足够接近。

思考下面这个例子，计算25的平方根：

(1) 随意选择一个数作为g，比如3；
(2) 3*3=9，通过判断可知这个数不够接近25；

① 很多人认为并不是希罗发明了这个方法，确实有些证据表明很可能是古巴比伦人。

(3) 将g替换为(3+25/3)/2=5.67[①]；

(4) 5.67*5.67=32.15，仍然不够接近25；

(5) 将g替换为(5.67+25/5.67)/2=5.04；

(6) 5.04*5.04=25.4和25足够接近，所以我们停止计算并宣布5.04是25平方根的合适近似值。

注意，这个方法描述的是一系列简单的步骤，结合一个控制流程来决定步骤的执行顺序。这样的描述称为*算法*[②]。这个算法就是猜测和验证算法的一个例子。它基于这样一个事实：我们很容易验证一个猜测是否足够好。

更严谨的说法是，一个算法就是一个有穷指令序列，描述了在给定输入上的一个*计算过程*，这个计算过程经过一系列事先定义好的状态，最终产生一个输出。

算法有点像菜谱：

① 加热蛋奶混合液；

② 搅拌；

③ 将勺子放入蛋奶混合液中；

④ 拿出勺子并用手指在勺子背面抹一下；

⑤ 如果勺子背面出现明显的痕迹，停止加热，并冷却蛋奶混合液；

⑥ 否则重复整个过程。

这个算法包含一些测试，用来判断过程是否完成，还包含一些和执行顺序相关的指令，有时会基于一个测试来跳转到其他指令。

那么如何把菜谱的思想应用到机械过程中呢？一种方法是设计一个专门计算平方根的机器。听起来可能很奇怪，但是最早的计算机器就是*固定程序计算机*，也就是说，它们的设计目的就是做非常具体的事情，而且大多数情况下用来解决具体的数学问题，例如计算炮弹的弹道。最早的计算机之一（1941年由阿塔纳索夫和贝里制造）被设计用来解线性方程组，除此之外什么都不能做。二战期间，阿兰·图灵设计的名为炸弹（bombe）的机器，仅仅用来破解德国的Enigma编码。一些非常简单的计算机仍然使用这种方法。例如，四功能计算器就是一个固定程序计算机，只能做基本的数学运算，不能用来处理文本或者玩电子游戏。要改变这类机器的程序，只能更换电路。

第一台真正的现代计算机是Manchester Mark 1。和它的前辈不同，Manchester Mark 1[③]是*程序存储计算机*。这样的计算机存储（并操作）一个指令序列，且有一个元素集合，这些元素会执行序列中的指令。通过创建指令集结构并将计算分解为一个指令序列（也就是一个程序），我们可

① 为简单起见，我们四舍五入了结果。

② “算法”这个词是波斯数学家穆罕默德·阿尔·花拉子模（Muhammad ibn Musa al-Khwarizmi）发明的。

③ 这台计算机在曼彻斯特大学建造，1949年运行了第一个程序。它实现了约翰·冯·诺依曼之前提出的想法，并被阿兰·图灵在1936年提出的通用图灵机理论所预测。

以制造高度可扩展的机器。程序存储计算机可以轻易地改变程序，因为它可以在程序的控制下用处理数据的方式来处理这些指令。实际上，计算机的核心变成了一个可以运行任意合法指令集的程序（叫作*解释器*），它可以计算任何能用基本指令集描述的东西。

需要操作的程序和数据都存储在内存中。一般来说，会有一个程序计数器指向内存中的特定位置，这个位置上的指令会被当作计算的开始。大多数情况下，解释器仅仅是继续执行序列中的下一条指令。但在有些情况下，它会执行一个测试，根据测试结果的不同，可能会跳转到序列中的其他位置，这叫作*控制流*。控制流非常重要，有了它，我们就可以编写完成复杂任务的程序。

再回到菜谱这个比喻上来，如果给一个优秀厨师一组固定的原料，他可以通过不同的组合方式创造出无限种美食。同理，给一个优秀程序员一组固定的初始元素，他可以创造出无限种有用的程序。这就是编程如此让人吃惊的原因。

要创造菜谱，或者说指令序列，我们需要一门*编程语言*来描述它们，从而向计算机发号施令。

1936年，英国数学家阿兰·图灵提出了一种理论计算设备，叫作*通用图灵机*。这个机器有一条无限长的纸带作为内存，纸带上可以记录0、1以及一些非常简单原始的指令，用来移动、读取和输出内容到纸带。*邱奇–图灵论题*宣称，如果一个函数是可计算的，那么图灵机一定可以通过编程来计算它。

邱奇–图灵论题中的“如果”非常重要。并不是所有问题都有可计算的解。例如，图灵提出，不可能写这样一个程序：给定一个任意的程序P，当且仅当P永不停止时输出`true`。这就是著名的*停机问题*。

邱奇–图灵论题直接导致了*图灵完备性*这个概念的提出。当一门编程语言可以用于模拟一个通用图灵机时，我们才说它是图灵完备的。因此，任何可以用一门编程语言（比如Python）编写的程序都可以用另一门编程语言（比如Java）来编写。当然，有的东西使用特定的语言更容易编写，但是所有语言的计算能力都是等价的。

幸好，程序员并不需要使用图灵的原始指令来编写程序，现代编程语言提供了一个更大并且更实用的原始指令集。然而，编程的基本思想仍然是组装一个操作的序列。

无论使用什么原始指令集，无论用何种方法来使用它们，编程的优势和劣势都是一样的：计算机只能做你让它做的事。优势在于，你可以随意创造有趣和有用的东西。劣势在于，当它没有按照你的想法来运行时，你只能从自身找原因。

世界上有成百上千种编程语言，并没有哪个是最好的（不过或许你可以找出一些最坏的）。对于不同类型的应用程序来说，不同的语言可能有好有坏。比如MATLAB非常适合操作向量和矩阵，C语言非常适合用来开发控制数据网络的程序，PHP非常适合用来开发网站，Python则是一个多面手，可以胜任很多种工作。

每种编程语言都有基本结构、语法、静态语义和语义。可以用一门自然语言类比，拿英语来