



TURING

图灵程序设计丛书

Introduction to computation and programming using Python
with application to understanding data

Python编程导论

(第2版)

【美】John V. Guttag 著
陈光欣 译

- MIT热门MOOC教材
- 基于Python 3讲解计算机科学基础知识
- 掌握用计算思维解决问题的能力



 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

Introduction to computation and programming using Python
with application to understanding data

Python编程导论

(第2版)

【美】John V. Guttag 著
陈光欣 译

人民邮电出版社
北 京

图书在版编目 (C I P) 数据

Python编程导论 : 第2版 / (美) 约翰·谷泰格著 ;
陈光欣译. -- 北京 : 人民邮电出版社, 2018. 2 (2018. 5重印)
(图灵程序设计丛书)
ISBN 978-7-115-47376-9

I. ①P… II. ①约… ②陈… III. ①软件工具—程序
设计—教材 IV. ①TP311. 561

中国版本图书馆CIP数据核字 (2017) 第308830号

内 容 提 要

本书基于 MIT 编程思维培训讲义写成, 主要目标在于帮助读者掌握并熟练使用各种计算技术, 具备用计算思维解决现实问题的能力。书中以 Python 3 为例, 介绍了对中等规模程序的系统性组织、编写、调试, 帮助读者深入理解计算复杂度, 还讲解了有用的算法和问题简化技术, 并探讨各类计算工具的使用。与本书第 1 版相比, 第 2 版全面改写了后半部分, 且书中所有示例代码都从 Python 2 换成了 Python 3。

本书适合对编程知之甚少但想要使用计算方法解决问题的读者。

-
- ◆ 著 [美] John V. Guttag
译 陈光欣
责任编辑 陈 曦
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>
三河市君旺印务有限公司印刷
- ◆ 开本: 800×1000 1/16
印张: 21.25
字数: 502千字 2018年2月第1版
印数: 4 001 - 6 000册 2018年5月河北第2次印刷
著作权合同登记号 图字: 01-2017-1465号
-

定价: 69.00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: 京东工商广登字 20170147 号

站在巨人的肩膀上
Standing on Shoulders of Giants



iTuring.cn

站在巨人的肩上
Standing on Shoulders of Giants



iTuring.cn

版 权 声 明

Copyright © 2016 Massachusetts Institute of Technology. Published in the English language under the title *Introduction to computation and programming using Python : with application to understanding data*.
Simplified Chinese-language edition copyright © 2018 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由The MIT Press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

本书基于MIT的一门课程写成，这门课程始于2006年，自2012年起，成为edX和MITx上的一门“大规模在线开放课程”（Massive Online Open Courses, MOOC）。本书第1版基于一个学期的课程，但随着时间的推移，我不得不添加更多内容，再用一学期来讲述课程已经不合适了。现在的这个版本适合于两学期的计算机科学系列导论课程。

当我开始编写第2版时，本以为只要加上几章内容就可以了，但结果远超预料。我重新组织了本书的后半部分，并将整本书中的代码从Python 2换成了Python 3。

本书面向的是那些没有或只有很少编程经验，但希望掌握计算方法来解决问题的学生。书中的内容是一些学生学习更高级计算机科学课程的跳板，但对更多学生来说，则是正式学习计算机科学的一门课程。

正因如此，所以我们更强调课程的广度，而不是深度。课程的目标是为学生简述更多的主题，使他们在想用计算机完成目标时知道自己能做什么。尽管如此，这并不是一门“计算机鉴赏”课程，要求比较严格，而且有一定难度。读者需要花费大量时间和精力才能真正掌握书中内容，使计算机服从自己的调遣。

本书的主要目标是帮助学生掌握并熟练使用各种计算技术，以得到有价值的成果。他们应该学会使用计算思维表述问题，并掌握如何从数据中提取信息。学生从本书中获得的最重要的能力是，使用计算思维解决问题的艺术。

这本书很难纳入传统的计算机科学课程。第1~11章是典型的针对没有或只有很少编程经验的学生的计算机科学课程；第12~14章稍微高级一些，如果想学习进阶技术，可以从这几章挑选些内容，作为导论课程的补充；第15~24章介绍如何使用计算技术来理解数据，我们认为其中的内容应该成为计算机科学课程体系中的第二门课程（代替传统的数据结构课程）。

第1~11章主要包含五个方面的内容：

- 编程基础
- Python 3编程语言
- 计算问题的解决技术
- 计算复杂度
- 使用图形表示信息

我们会介绍Python语言的大部分特性，但重点在于可以使用编程语言做什么，而不是语言本身。比如，第3章结束时虽然只介绍了Python语言的一小部分，但已经引入穷举的概念、猜测与

验证算法、二分查找和高效近似算法。纵贯本书，我们都会介绍Python的特性。同样地，本书从头至尾也会介绍编程方法。我们的理念是帮助学生们掌握Python，并成为优秀的程序员，能够使用计算技术解决自己感兴趣的问题。

书中示例都使用Python 3.5进行了测试。Python 3修正了Python 2各种发布版本（通常称为Python 2.x）在设计上的很多不一致性，但它不是向后兼容的，这意味着大多数使用Python 2编写的程序不能在Python 3中正常运行。因为这个原因，Python 2.x还在被广泛使用。第一次使用Python 3中不同于Python 2的特性时，我们都会明确指出如何在Python 2中完成相同功能。书中所有示例都有Python 3.5和Python 2.7的在线版本。

第12~13章介绍了优化，这是一个虽然重要但很少包含在导论课程中的主题。第14~16章介绍了随机规划，这也是一个虽然重要但很少包含在导论课程中的主题。我们在MIT授课的经验是，在一个学期的导论课程中，或者可以讲述第12~13章的内容，或者可以讲述第14~16章的内容，但不能二者兼顾。

第15~24章在设计上是独立成篇的，内容涉及如何使用计算技术来理解数据。其中使用的数学知识不超出高中代数的范围，但要求读者具有严谨的思维能力，且不会被数学概念吓倒。这一部分包括多数导论课程中没有的内容：数据可视化、模拟模型、概率与统计思维，以及机器学习。我们相信对大多数学生来说，这部分内容远比那些典型的第二门计算机科学课程的内容更有意义。

我们没有在每章末尾设置习题，而是在每章的适当部分插入了“实际练习”。有些练习非常简短，目的是使读者确认自己确实明白了刚刚学过的内容。有些练习增加了一点挑战性，适合在考试时使用。其余练习的难度比较大，可以作为课后作业。

本书有三个普适性主题：系统的问题解决方式、抽象能力，以及将计算思维作为思考世界的一种方式。学完本书后，你应该：

- 学会使用Python语言进行编程和计算；
- 学会系统性地组织、编写、调试中等规模的程序；
- 理解计算复杂度；
- 将模糊的问题描述转化为明确的计算方法，以此解决问题，并对这个过程有深刻的理解；
- 掌握一些有用的算法以及问题简化技术；
- 对于那些很难得到封闭解的问题，知道如何使用随机性和模拟技术进行清晰阐述；
- 学会使用计算工具（包括简单的统计、可视化以及机器学习工具）对数据进行理解与建模。

编程本身就是一项非常困难的活动。正如那句名言所说：“在几何中，没有专为国王铺设的大道。”^①对于编程来说，也没有捷径可走。如果你想真正掌握书中的内容，光阅读是不够的，还应该亲自运行书中的代码。2008年以来，本书基于的各种课程都放在MIT的开放课程网站（OpenCourseWare, OCW）上。这里有授课过程的视频录像，以及一系列习题和考题。从2012年秋季开始，我们在edX和MITx上提供了在线课程，其中包括本书的大部分内容。我们强烈推荐你做做OCW和edX上提供的习题。

^① 据传，公元前300年左右，托勒密国王希望有学习数学的捷径。对于这个要求，欧几里得给出了这样的回答。

电子书

如需购买本书电子版，请扫描以下二维码。



致 谢

本书第1版基于我在MIT教授的本科生课程讲义，这门课程——当然也包括这本书——得益于我的教师同事（尤其是Ana Bell、Eric Grimson、Srinivas Devadas、Fredo Durand、Ron Rivest和Chris Terman）、助教以及学习该课程的学生。David Guttag克服了他对计算机科学的厌恶，对本书的多个章节进行了校对。

和所有成功的教授一样，我要向我的研究生表示万分感激。Guha Balakrishnan、David Blalock、Joel Brooks、Ganeshapillai Gartheeban、Jen Gong、Yun Liu、Anima Singh、Jenna Wiens和Amy Zhao这些学生不但进行了非常出色的研究工作（我也由此获得了一些好评），还都对本书的草稿提出了非常宝贵的意见。

我要向Julie Sussman表示由衷的感谢。我开始和她接触后，才知道一个优秀的编辑竟然会如此出色。以前出书时，与我合作的文字编辑也非常能干，因而我觉得这本书有文字编辑也就够了，但我错了。我需要合作者可以从学生视角审视本书，并且告诉我需要做什么、应该做什么和能够做什么——如果我有时间和精力。Julie给我的好建议数不胜数，而且都切中要害，不可忽视。不论是在语言方面还是在编程方面，Julie都有极为深厚的造诣。

最后，感谢我的妻子Olga，她一直督促我完成这本书。感谢她帮助我承担了很多家庭义务，这样我才能专心于此。

目 录

第1章 启程	1	第5章 结构化类型、可变性与 高阶函数	50
第2章 Python 简介	6	5.1 元组	50
2.1 Python 基本元素	7	5.2 范围	52
2.1.1 对象、表达式和数值类型	8	5.3 列表与可变性	52
2.1.2 变量与赋值	9	5.3.1 克隆	57
2.1.3 Python IDE	11	5.3.2 列表推导	57
2.2 程序分支	12	5.4 函数对象	58
2.3 字符串和输入	14	5.5 字符串、元组、范围与列表	60
2.3.1 输入	15	5.6 字典	61
2.3.2 杂谈字符编码	16		
2.4 迭代	17	第6章 测试与调试	65
第3章 一些简单的数值程序	20	6.1 测试	65
3.1 穷举法	20	6.1.1 黑盒测试	66
3.2 for 循环	22	6.1.2 白盒测试	68
3.3 近似解和二分查找	24	6.1.3 执行测试	69
3.4 关于浮点数	27	6.2 调试	70
3.5 牛顿-拉弗森法	29	6.2.1 学习调试	72
第4章 函数、作用域与抽象	31	6.2.2 设计实验	72
4.1 函数与作用域	32	6.2.3 遇到麻烦时	75
4.1.1 函数定义	32	6.2.4 找到“目标”错误之后	76
4.1.2 关键字参数和默认值	33		
4.1.3 作用域	34	第7章 异常与断言	77
4.2 规范	37	7.1 处理异常	77
4.3 递归	39	7.2 将异常用作控制流	80
4.3.1 斐波那契数列	40	7.3 断言	82
4.3.2 回文	42		
4.4 全局变量	45	第8章 类与面向对象编程	83
4.5 模块	46	8.1 抽象数据类型与类	83
4.6 文件	47	8.1.1 使用抽象数据类型设计程序	87
		8.1.2 使用类记录学生与教师	87

8.2 继承	90	第 13 章 动态规划	155
8.2.1 多重继承	92	13.1 又见斐波那契数列	155
8.2.2 替换原则	93	13.2 动态规划与 0/1 背包问题	157
8.3 封装与信息隐藏	94	13.3 动态规划与分治算法	162
8.4 进阶示例：抵押贷款	99	第 14 章 随机游走与数据可视化	163
第 9 章 算法复杂度简介	103	14.1 随机游走	163
9.1 思考计算复杂度	103	14.2 醉汉游走	164
9.2 渐近表示法	106	14.3 有偏随机游走	170
9.3 一些重要的复杂度	107	14.4 变幻莫测的田地	175
9.3.1 常数复杂度	107	第 15 章 随机程序、概率与分布	178
9.3.2 对数复杂度	108	15.1 随机程序	178
9.3.3 线性复杂度	108	15.2 计算简单概率	180
9.3.4 对数线性复杂度	109	15.3 统计推断	180
9.3.5 多项式复杂度	109	15.4 分布	192
9.3.6 指数复杂度	111	15.4.1 概率分布	194
9.3.7 复杂度对比	112	15.4.2 正态分布	195
第 10 章 一些简单算法和数据结构	114	15.4.3 连续型和离散型均匀分布	199
10.1 搜索算法	115	15.4.4 二项式分布与多项式分布	200
10.1.1 线性搜索与间接引用元素	115	15.4.5 指数分布和几何分布	201
10.1.2 二分查找和利用假设	116	15.4.6 本福德分布	203
10.2 排序算法	119	15.5 散列与碰撞	204
10.2.1 归并排序	120	15.6 强队的获胜概率	206
10.2.2 将函数用作参数	122	第 16 章 蒙特卡罗模拟	208
10.2.3 Python 中的排序	123	16.1 帕斯卡的问题	209
10.3 散列表	124	16.2 过线还是不过线	210
第 11 章 绘图以及类的进一步扩展	128	16.3 使用查表法提高性能	213
11.1 使用 PyLab 绘图	128	16.4 求 π 的值	214
11.2 进阶示例：绘制抵押贷款	133	16.5 模拟模型结束语	218
第 12 章 背包与图的最优化问题	139	第 17 章 抽样与置信区间	220
12.1 背包问题	139	17.1 对波士顿马拉松比赛进行抽样	220
12.1.1 贪婪算法	140	17.2 中心极限定理	225
12.1.2 0/1 背包问题的最优解	143	17.3 均值的标准误差	228
12.2 图的最优化问题	145	第 18 章 理解实验数据	231
12.2.1 一些典型的图论问题	149	18.1 弹簧的行为	231
12.2.2 最短路径：深度优先搜索和 广度优先搜索	149	18.2 弹丸的行为	238

18.2.1 可决系数	240	21.8 慎用外推法	273
18.2.2 使用计算模型	241	21.9 得克萨斯神枪手谬误	274
18.3 拟合指数分布数据	242	21.10 莫名其妙的百分比	276
18.4 当理论缺失时	245	21.11 不显著的显著统计差别	276
第 19 章 随机试验与假设检验	247	21.12 回归假象	277
19.1 检验显著性	248	21.13 小心为上	278
19.2 当心 P-值	252	第 22 章 机器学习简介	279
19.3 单尾单样本检验	254	22.1 特征向量	281
19.4 是否显著	255	22.2 距离度量	283
19.5 哪个 N	257	第 23 章 聚类	288
19.6 多重假设	258	23.1 Cluster 类	289
第 20 章 条件概率与贝叶斯统计	261	23.2 K 均值聚类	291
20.1 条件概率	262	23.3 虚构示例	292
20.2 贝叶斯定理	263	23.4 更真实的示例	297
20.3 贝叶斯更新	264	第 24 章 分类方法	303
第 21 章 谎言、该死的谎言与统计学	267	24.1 分类器评价	303
21.1 垃圾输入, 垃圾输出	267	24.2 预测跑步者的性别	306
21.2 检验是有缺陷的	268	24.3 K 最近邻方法	308
21.3 图形会骗人	268	24.4 基于回归的分类器	312
21.4 Cum Hoc Ergo Propter Hoc	270	24.5 从“泰坦尼克”号生还	320
21.5 统计测量不能说明所有问题	271	24.6 总结	325
21.6 抽样偏差	272	Python 3.5 速查表	326
21.7 上下文很重要	273		

第1章

启 程



计算机能且只能做两件事，执行计算与保存计算结果，但它把这两件事都做到了极致。常见的台式机或笔记本电脑每秒钟大概可以执行10亿次计算，快得难以置信。想象一下，让一个离地面1米高的球自由落体到地面上，这么短暂的时间，你的计算机已经执行了超过10亿条指令。至于存储，一台普通计算机可以有几千亿字节的存储空间。这是什么概念呢？打个比方，如果1字节（byte，表示一个字符所需的位数，通常是8位）的重量是1克（实际上当然不是），那么100 GB的重量就相当于10 000吨，这几乎是15 000头非洲象的重量。

在人类历史的大部分时间里，计算受限于人类大脑的计算速度以及人类双手记录计算结果的能力，这意味着通过计算只能解决一些最简单的问题。即使现代计算机具备如此快的速度，它在很多问题上依然无能为力（例如，搞清楚气候变化），但越来越多的问题已经被证明可以通过计算解决。我们希望你学习完本书之后，可以熟练地将计算思维应用到解决工作、学习、生活问题的过程中。

那么，计算思维到底指什么呢？

所有知识都可以归结为两类：陈述性知识和程序性知识。陈述性知识由对事实的描述组成。例如：“如果满足 $y \times y = x$ ，那么 x 的平方根就是数值 y 。”这就是对事实的描述。遗憾的是，它没有告诉我们怎样求出平方根。

程序性知识说明“如何做”，描述的是信息演绎的过程。亚历山大的海伦（Heron of Alexandria）第一次提出如何计算一个数的平方根^①。他的方法可以总结如下：

- (1) 随机选择一个数 g ；
- (2) 如果 $g \times g$ 足够接近 x ，那么停止计算，将 g 作为答案；
- (3) 否则，将 g 和 x/g 的平均数作为新数，也就是 $(g + x/g)/2$ ；
- (4) 使用新选择的数——还是称其为 g ——重复这个过程，直到 $g \times g$ 足够接近 x 。

思考下面这个例子，求出25的平方根。

- (1) 为 g 设置一个任意值，例如3；
- (2) 我们确定 $3 \times 3 = 9$ ，没有足够接近25；

^① 很多人认为海伦不是这种方法的发明者，确实有一些证据表明，是古巴比伦人发明了这种方法。

(3) 设置 g 为 $(3 + 25/3)/2 = 5.67$;^①

(4) 我们确定 $5.67 \times 5.67 = 32.15$ 还是不够接近25;

(5) 设置 g 为 $(5.67 + 25/5.67)/2 = 5.04$;

(6) 我们确定 $5.04 \times 5.04 = 25.4$ 已经足够接近25了, 所以停止计算, 宣布5.04就是25的平方根的一个合适的近似值。

请注意, 这个方法描述的是一系列简单的步骤, 以及一个控制流, 用来确定某个步骤在什么情况下得以执行。这种描述称为算法^②。这个例子是一个猜测与检验算法。它基于这样一个事实: 我们很容易验证一个猜测是否合理。

更加正式的定义是: 算法是一个有穷指令序列, 描述了这样一种计算过程, 即在给定的输入集合中执行时, 会按照一系列定义明确的状态进行, 最终产生一个输出结果。

算法有点像菜谱:

(1) 将蛋奶糊加热;

(2) 搅拌;

(3) 将调羹浸入蛋奶糊;

(4) 拿出调羹, 用手指划一下调羹背面;

(5) 如果留下明显痕迹, 则停止加热并冷却;

(6) 否则重复以上步骤。

算法包含一些测试指令, 用来确定整个过程何时结束; 还包含一些顺序指令, 用来确定指令执行的顺序。有些时候, 还会根据测试结果跳转到某些指令。

那么, 如何将菜谱的思想应用到机械过程中呢? 一种方法就是, 专门设计一台用来计算平方根的机器。这听起来有点奇怪, 但实际上, 最早用来计算的机器就是这种固定程序计算机。顾名思义, 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, 如计算炮弹的弹道。阿塔纳索夫和贝里在1941年建造的机器算是最早的计算机之一, 它可以用来解线性方程组, 但其他什么都不会。阿兰·图灵在二战期间设计Bombe计算机器的目的就是, 专门破解纳粹德国的Enigma密码。现在, 一些非常简单的计算机还在使用这种方法。例如, 四功能计算器就是一种固定程序计算机, 它只能做简单的算术, 不能用作文字处理器, 也不能运行视频游戏。要改变这种机器的程序, 只能更换电路。

第一台真正的现代计算机是Manchester Mark 1^③。相比于那些先驱者, 它有本质上的改进, 即它是一台存储程序计算机。这种计算机存储(并操作)一个指令序列, 并具有一个可以执行序列中任何指令的元素集合。通过创建指令集结构, 并将计算过程详细划分为一个指令序列(也就是一个程序), 我们可以制造高度灵活的机器。存储程序计算机可以对指令进行处理, 就像处理数据一样, 这样就能够轻松修改程序, 并且可以在程序的控制之下做这些事情。实际上, 计算机

① 为简单起见, 我们对结果进行了四舍五入。

② 这个词源于波斯数学家穆罕默德·伊本·穆萨·阿尔·花刺子模(Muhammad ibn Musa al-Khwarizmi)。

③ 这台计算机建造于曼彻斯特大学, 1949年运行了第一个程序。它将先前约翰·冯·诺依曼提出的设想变成了现实, 也验证了阿兰·图灵1936年提出的通用图灵机理论。

的核心变成了可以执行任意合法指令集的程序（称为解释器），这样计算机就能够计算任何可以使用基本指令集描述的问题。

计算机操作的程序和数据都存储在内存中。通常都有一个程序计数器指向内存中的特定位置，通过执行这个位置上的指令，计算过程得以开始。大多数情况下，解释器只是简单地按照顺序执行序列中的下一条指令，但并不总是如此。在某些情况下，解释器执行一个测试，然后根据测试结果可能跳到指令序列的其他位置继续执行。这称为控制流，是允许我们编写可执行复杂任务的程序的必备条件。

再回到菜谱这个比喻，如果给定一组固定原料，那么一位优秀的厨师通过各种方式的搭配，可以做出非常多的美味佳肴。同样地，如果给定一个小规模固定初始元素组合，那么一位优秀的程序员也可以创造出非常多的有用程序。这就是编程如此引人入胜的原因。

要创建“菜谱”，即指令序列，我们需要能够描述这些指令的编程语言，从而向计算机发号施令。

1936年，英国数学家阿兰·图灵提出了一种抽象的计算设备，后来称为通用图灵机。这种机器具有无限的存储容量，即一条无限长的纸带。可以在纸带上面写入0和1，以及一些非常简单的初始指令，从而对纸带进行移动、读出和写入等操作。邱奇-图灵论题表明，如果一个函数是可计算的，那么一定可以通过对图灵机进行编程实现这种计算。

邱奇-图灵论题中的“如果”非常重要，并非所有问题都可以通过计算求解。举例来说，图灵就曾经证明，不存在这样一个程序：对于给定的任意程序P，当且仅当P永远运行时输出true。这就是著名的停机问题。

邱奇-图灵论题可以直接推导出图灵完备性这个概念。如果一门编程语言可以模拟通用图灵机，才可以说它是图灵完备的。所有现代编程语言都是图灵完备的。因此，任何可以被一门编程语言（例如Python）实现的程序，都可以被另一门编程语言（例如Java）实现。当然，有些事情用某种语言实现起来更容易，但就计算能力而言，所有语言从根本上说都是相等的。

幸运的是，程序员不必在图灵初始指令集的基础上构建程序，现代编程语言提供了更大、更方便的初始指令集。但是，编程基本思想的核心仍然是组装操作序列的过程。

不管具有什么样的初始指令集，也不管如何使用初始指令集，关于编程的最美妙的事情同时也是最糟糕的事情：计算机严格按照你的指令去做。这很美妙，因为这意味着你可以使用计算机做各种各样既有趣又有用的事情；这很糟糕，因为如果计算机没有按照你的期望去做，那么你不能怨天尤人，只能自怨自艾。

当今世界中存在着几百种编程语言，没有哪一门语言是最好的（尽管你可以数出一些最差的）。术业有专攻，每种语言都有自己的用武之地。举例来说，MATLAB是一门非常优秀的语言，适合操作向量和矩阵；C也是一门优秀的语言，适合开发控制数据网络的程序；PHP是建立网站的理想选择；Python则以良好的通用性著称。

每种编程语言都有基本结构、语法、静态语义和语义。如果用一门自然语言类比，例如英语，那么基本结构就是单词，语法则用来描述哪些单词放在一起可以组成通顺的句子，静态语义定义了哪些句子是有意义的，语义则定义了句子的实际含义。Python语言中的基本结构包括字面量（例

如,数值3.2和字符串'abc')和中缀操作符(例如,+和/)。

语言中的语法定义了字符和符号组成句子的正确形式。例如,英语中的Cat dog boy这个句子在语法上是错误的,因为英语语法不接受<名词><名词><名词>这样的句子形式。在Python中,3.2 + 3.2这样的基本结构序列在语法上是良好的,但3.2 3.2这个序列则不是。

静态语义定义了哪些语法有效的句子是有意义的。例如,英语中I are big这个句子是<代词><系动词><形容词>的形式,在语法上是有效的。然而它不是有效的英语句子,因为代词I是单数,而系动词are是复数。这就是典型的静态语义错误。在Python中,序列3.2/'abc'在语法上没有问题(<字面量><操作符><字面量>),但会产生一个静态语义错误,因为数值除以字符串是没有意义的。

在一门语言中,语义为每个语法正确又没有静态语义错误的句子关联一个含义。在自然语言中,句子的语义可以是模棱两可的。例如,句子I cannot praise this student too highly可以是一种恭维,也可以是一种批评。编程语言是被精心设计过的,所以每个程序都只有一种确切的含义。

尽管语法错误是最常见的错误(特别是学习一门新的编程语言时),它的危害性却最小。每种严谨的编程语言都会尽力检查语法错误,绝不允许用户运行有语法错误的程序。而且,大多数情况下,语言系统都会给出足够明确的提示,指出错误的位置,让用户明确得知如何修复错误。

至于静态语义错误,情况就有点复杂了。有些语言,比如Java,运行程序之前会做很多静态语义检查。但其他一些语言,比如C和Python,静态语义检查就比较少。Python在运行程序时确实会做相当数量的静态语义检查,但不会捕获所有静态语义错误。如果这些错误没有被检测出,程序的行为往往将是不可预知的。后面会看到一些这样的例子。

通常我们并不会说程序具有语义错误。如果一个程序没有语法错误,也没有静态语义错误,那么它就有某种含义。也就是说,它具有语义。当然,这并不是说它具有的语义就是程序员想表达的含义。如果程序的含义与程序员想表达的含义不同,那就糟糕了。

如果程序有错误且没有按照你的期望运行,那会发生什么呢?

- ❑ 它可能崩溃,也就是说,停止运行并表现出某种明显的崩溃迹象。在设计合理的计算系统中,一个程序的崩溃不会殃及整个系统。当然,某些非常流行的计算机系统并没有这种良好的特性。几乎所有的个人计算机用户都有过这种体验:某个程序出现问题时,必须重启计算机才能解决。
- ❑ 它也可能继续运行、运行、运行,永不停止。如果你不清楚程序完成任务大概需要多少时间,那么就很难识别这种异常情况。
- ❑ 它也可能运行结束,并产生一个可能正确也可能不正确的结果。

以上每种情况都不是我们想要的,特别是最后一种。当一个程序看上去正确运行而实际上没有时,接下来的事情就很糟糕了。财产可能损失,患者可能受到致命剂量的放射治疗,飞机可能会坠毁,等等。

程序如果没有正确运行,就应该表现出明显的错误。只要有可能,我们都应该以这种方式编写程序。如何实现这一点将贯穿本书始终。