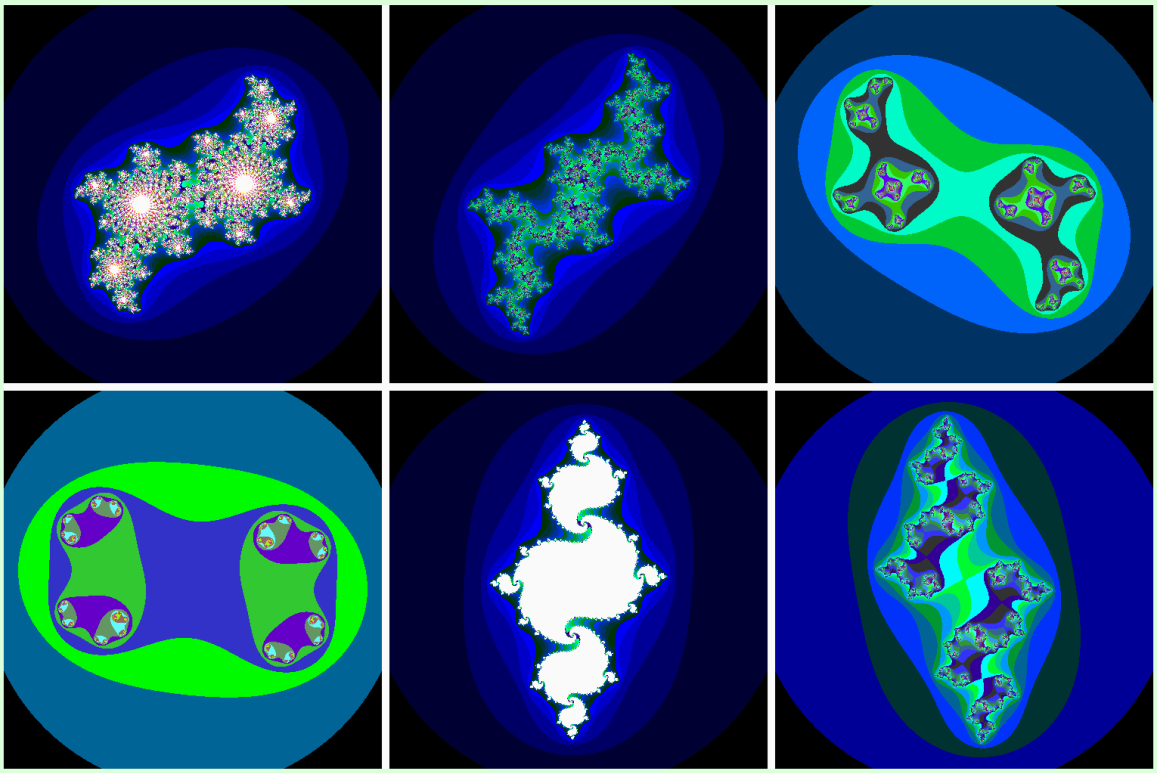
Python科学计算基础

罗奇鸣　编著



出版社

前言

本书是中国科学技术大学“Python 科学计算基础”课程的教材，已获教务处教材出版项目立项，目前正在编写和修改过程中。

本书的版权归编著者所有，仅供选课同学在校内使用，禁止对外传播。

目录

1 绪论 **1**

1.1 科学计算 1

1.2 利用计算机解决问题 1

1.3 程序设计语言的分类 2

1.4 过程式编程范式 2

1.5 面向对象编程范式 3

1.6 学习 Python 语言的理由 4

1.7 Python 语言的发展历史 4

1.8 Python 语言的特点 5

1.9 Python 科学计算环境 6

1.10 选课须知 6

1.10.1 选课要求和课程安排 6

1.10.2 学习方法 6

1.10.3 大作业 7

1.11 实验 1：安装和使用 Python 开发环境 9

2 内置数据类型及其运算 **13**

2.1 变量和类型 13

2.2 数值类型 14

2.2.1 int 类型 14

2.2.2 float 类型 15

2.2.3 complex 类型 16

2.2.4 数值类型的内置函数 17

2.2.5 math 模块和 cmath 模块 17

2.3 bool 类型 18

2.4 序列类型 19

2.4.1 list 类型 (列表) 和 tuple 类型 (元组) 19

2.4.2 str 类型 (字符串) 22

2.5 set 类型 (集合) 26

2.6 dict 类型 (字典) 27
2.7 实验 2：内置数据类型及其运算 28

3 分支和迭代 31
3.1 if 语句和 if-else 表达式 31
3.2 for 语句 33
3.3 while 语句 35
3.4 推导式 36
3.5 实验 3：分支和迭代 37

4 函数和模块 39
4.1 定义和调用函数 39
4.2 局部变量和全局变量 40
4.3 默认值形参和关键字实参 41
4.4 可变数量的实参 42
4.5 函数式编程 42
4.5.1 函数作为实参 43
4.5.2 函数作为返回值 45
4.6 递归 45
4.6.1 阶乘 46
4.6.2 最大公约数 46
4.6.3 字符串反转 47
4.6.4 快速排序 47
4.6.5 Fibonacci 数列 48
4.6.6 0-1 背包问题 49
4.6.7 矩阵链乘积问题 52
4.7 创建和使用模块 55
4.8 实验 4：函数和模块 61

5 类和继承 67
5.1 定义和使用类 67
5.1.1 二维平面上的点 67
5.1.2 复数 69
5.1.3 一元多项式 75
5.2 继承 77
5.3 迭代器和生成器 81
5.4 实验 5：类和继承 83

6 NumPy 数组和矩阵计算 89
6.1 创建数组 89
6.1.1 已有元素存储在其他类型的容器中 89
6.1.2 没有元素但已知形状 90
6.1.3 改变数组的形状 91
6.1.4 数组的堆叠 (Stacking) 92
6.1.5 数组的分割 92
6.2 数组的运算 94
6.2.1 基本运算 94
6.2.2 函数运算 95
6.3 索引、切片和迭代 96
6.4 复制和视图 102
6.5 矩阵计算 103
6.6 稀疏矩阵 106
6.7 实验 6：NumPy 数组和矩阵计算 110

7 错误处理和文件读写 113
7.1 错误处理 113
7.1.1 错误的分类 113
7.1.2 调试 114
7.1.3 异常处理 118
7.2 文件读写 121
7.2.1 打开和关闭文件 121
7.2.2 读写文本文件 121
7.2.3 读写 CSV 文件 123
7.2.4 读写 JSON 文件 124
7.2.5 读写 pickle 文件 125
7.2.6 读写 NumPy 数组的文件 126
7.3 实验 7：错误处理和文件读写 127

8 程序运行时间的分析和测量 129
8.1 算法和时间性能的分析 129
8.2 算法的时间复杂度 130
8.2.1 插入排序 130
8.2.2 归并排序 131
8.2.3 线性查找 133
8.2.4 二分查找 133
8.2.5 穷举法求解 3-sum 问题 134
8.2.6 穷举法求解 subset-sum 问题 135

8.3 程序运行时间的测量 136
8.4 Cython 139
8.5 实验 8：程序运行时间的分析和测量 142

9 Matplotlib 库简介 145
9.1 基于状态的绘图方式 145
9.2 面向对象的绘图方式 151
9.3 三维数据的图示 155
9.3.1 颜色映射图和等高线图 155
9.3.2 表面图、线框图和三维等高线图 158
9.4 二维和三维向量场 159
9.5 实验 9：Matplotlib 库简介 161

10 SymPy 库简介 163
10.1 符号表达式 163
10.2 符号表达式的变换 164
10.2.1 幂函数的变换 (表 10.4) 166
10.2.2 对数函数的变换 (表 10.5) 167
10.2.3 三角函数和双曲函数的变换 (表 10.6) 168
10.2.4 有理系数多项式函数的变换 (表 10.7) 168
10.2.5 特殊函数的变换 (表 10.8) 168
10.3 微积分 170
10.3.1 求导数 (表 10.9) 170
10.3.2 求积分 (表 10.10) 170
10.3.3 求极限 (表 10.11) 170
10.3.4 泰勒级数展开 (表 10.12) 170
10.4 矩阵 172
10.5 方程求解 176
10.5.1 代数方程求解 176
10.5.2 常微分方程求解 178
10.6 实验 10：SymPy 库简介 180

11 SciPy 库简介 181
11.1 插值和数值积分 181
11.1.1 一元插值：多项式函数和样条函数 181
11.1.2 二元插值 184
11.1.3 数值积分 186
11.2 代数方程求解 189
11.2.1 线性方程组求解 189
11.2.2 非线性方程求解 190

11.2.3 非线性方程组求解 193
11.3 常微分方程求解 195
11.3.1 常微分方程的初值问题 195
11.3.2 初值问题的数值求解方法 198
11.3.3 边值问题的数值求解方法 202
11.4 实验 11：SciPy 库简介 205

12 统计计算 207
12.1 概率分布 207
12.2 随机数和模拟 213
12.3 假设检验 217
12.4 线性回归 221

13 最优化方法 229
13.1 非线性最优化问题的解析求解方法 230
13.1.1 无约束非线性最优化问题的解析求解方法 230
13.1.2 约束非线性最优化问题的解析求解方法 232
13.2 非线性最优化问题的数值求解方法 236
13.3 线性最优化问题的求解方法 246
13.4 混合整数线性最优化问题的求解方法 248
13.4.1 分支定界法 249
13.4.2 Pyomo 251

参考文献 256

第 1 章　绪论

1.1　科学计算

科学计算 (Scientific Computing) 是以数学和计算机科学为基础形成的交叉学科，是利用计算机的计算能力求解科学和工程问题的数学模型所需的理论、技术和工具的集合 [1]。随着计算机的计算能力的不断提升，科学计算也得到了迅速发展。理论分析、实验和科学计算并称为当今科学发现的三个支柱。

数值分析 (Numerical Analysis) 是科学计算的重要组成部分，其特点包括：计算对象是连续数值；被求解的问题一般没有解析解或理论上无法在有限步求解。例如一元 N 次 ($N \geq 5$) 方程和大多数非线性方程不存在通用的求根公式，需要使用数值方法迭代求解。数值分析的目标是寻找计算效率高和稳定性好的迭代算法。

1.2　利用计算机解决问题

计算机是能够自动对数据进行计算的电子设备。计算机的优势是运算速度快。以下举例说明。

1. 1946 年诞生的世界上第一台通用计算机 ENIAC 每秒能进行 5000 次加法运算 (据测算人最快的运算速度仅为每秒 5 次加法运算) 和 400 次乘法运算。人工计算一条弹道需要 20 多分钟时间，ENIAC 仅需 30 秒!

2. 2018 年投入使用的派-曙光是首台应用中国国产卫星数据，运行我国自主研发的数值天气预报系统 (GRAPES) 的高性能计算机系统。该系统峰值运算速度达到每秒 8189.5 万亿次，内存总容量达到 690432GB。近年来，我国台风路径预报 24 小时误差稳定在 70 公里左右，各时效预报全面超过美国和日本，达国际领先水平。同样，降水、雷电、雾-霾、沙尘等预报预测准确率也整体得到提升。

为了利用计算机解决问题，必须使用某种程序设计语言把解决问题的详细过程编写为程序，即一组计算机能识别和执行的指令。计算机通过运行程序解决问题。

[1] https://www.scicomp.uni-kl.de/about/scientific-computing/

1.3 程序设计语言的分类

程序设计语言可分为三类：机器语言、汇编语言和高级语言。早期的计算机只能理解机器语言。机器语言用 0 和 1 组成的二进制串表示 CPU(处理器) 指令和数据。之后出现的汇编语言用易于理解和记忆的符号来代替二进制串，克服了机器语言难以理解的缺点。机器语言和汇编语言的共同缺点是依赖于 CPU，用它们编写的程序无法移植到不同的 CPU 上。1956 年投入使用的的 Fortran 语言是第一种高级语言。高级语言采用接近自然语言和数学公式的方式表达解决问题的过程，不再依赖于 CPU，实现了可移植性。

近几十年来，高级语言不断涌现，数量达到几百种。高级语言按照编程范式 (programming paradigm) 可划分为以下几个类别:

1. 命令式 (imperative): 使用命令的序列修改内存状态，例如 C、C++、Java 和 Python 等。
2. 声明式 (declarative): 仅指明求解的结果，而不说明求解的过程，例如 SQL 等。
3. 过程式 (procedural): 可进行过程调用的命令式，例如 C、C++、Java 和 Python 等。
4. 函数式 (functional): 不修改内存状态的函数互相调用，例如 Lisp、ML、Haskell 和 Scala 等。
5. 逻辑式 (logic): 基于已知事实和规则推断结果，例如 Prolog 等。
6. 面向对象 (object-oriented): 有内部状态和公开接口的对象互相发送消息，例如 Simula 67、C++、Java 和 Python 等。

这些编程范式并非互斥，一种语言可同时支持多种范式。

1.4 过程式编程范式

过程式的特点是基于输入和输出将一个较复杂的问题逐步分解成多个子问题。如果分解得到的某个子问题仍然较复杂，则继续对其分解，直至所有子问题都易于解决为止。过程式的程序由多个过程构成，每个过程解决一个子问题。这些模块形成一个树状结构。每一模块内部均是由顺序、选择和循环三种基本结构组成。在软件维护时，如果需要修改软件使用的数据，则处理数据的过程也需要进行修改。过程式的缺点是软件需求发生变化时的维护代价较高，也不易实现代码复用，因此不适于开发大型软件。

以下通过一个实例说明过程式编程范式：根据年和月输出日历。程序的运行结果见 1.1。

Listing 1.1: **运行模块 calendar.py 的示例**

```
In[2]: run month_calendar.py --year 2022 --month 10
2022  10
```

```
---------------------------
Sun  Mon  Tue  Wed  Thu  Fri  Sat
                            1
2    3    4    5    6    7    8
9    10   11   12   13   14   15
16   17   18   19   20   21   22
23   24   25   26   27   28   29
30   31
```

采用过程式编程范式进行问题分解，得到以下设计方案：

1. 读取用户输入的年和月
2. 输出日历的标题
3. 输出日历的主体
   (a) 怎样确定指定的某年某月有多少天？
      i. 如果是 2 月，怎样确定指定年是否是闰年？
   (b) 怎样确定这个月的第一天是星期几？用 $w(y,m,d)$ 表示计算 y 年 m 月 d 日是星期几的函数，函数值为 $0,1,...,6$ 依次表示周日、周一、...、周六。已知有公式可以计算指定的某年 y 的一月一日是星期几：

$$w(y,1,1) = (y + \lfloor (y-1)/4 \rfloor - \lfloor (y-1)/100 \rfloor + \lfloor (y-1)/400 \rfloor) \ \% \ 7 \tag{1.1}$$

   其中 $\lfloor x \rfloor$ 表示不大于实数 x 的最大整数，$\% \ 7$ 表示除以 7 得到的余数。
   用 $v(y_1,m_1,d_1,y_2,m_2,d_2)$ 表示计算从 y_1 年 m_1 月 d_1 日到 y_2 年 m_2 月 d_2 日经历了多少天的函数，则易知

$$w(y_2,m_2,d_2) = (w(y_1,m_1,d_1) + v(y_1,m_1,d_1,y_2,m_2,d_2)) \ \% \ 7 \tag{1.2}$$

   对于 y 年 1 月 1 日和 y 年 m 月 1 日使用以上公式：

$$w(y,m,1) = (w(y,1,1) + v(y,m,1,y,1,1)) \ \% \ 7 \tag{1.3}$$

   剩余的问题就是计算从 y 年 1 月 1 日到 y 年 m 月 1 日所经历的总天数。
      i. 怎样确定任意指定的某年某月有多少天？已由 3(a) 解决。

1.5 面向对象编程范式

面向对象编程范式的特点如下：

1. 将客观事物直接映射到软件系统的对象。对象是将数据及处理数据的过程封装在一起得到的整体，用以表示客观事物的状态和行为。从同一类型的对象中抽象出其共同的属性

和操作，形成类。类是创建对象的模板，对象是类的实例。例如在一个实现学生选课功能的软件系统中，每位学生是一个对象。从所有学生对象中提取出共同的属性 (学号、姓名、所在系等) 和操作 (选课、退课等)，形成学生类。

2. 每个类作为一个独立单元进行开发、测试和维护。如果需要修改类的实现细节，只要不改变类的接口就不会影响使用该类的外部代码，使软件系统更易于维护。

3. 通过继承可以重复利用已有类的代码，并根据需要进行扩展，从而提升了软件系统的开发效率。

4. 程序由多个类构成。程序在运行时由各个类创建一些对象，对象之间通过明确定义的接口进行交互，完成软件系统的功能

面向对象编程范式的优点是易于开发和维护大型软件，缺点是在程序的运行效率上不如过程式程序设计方法。

1.6 学习 Python 语言的理由

TIOBE 指数由荷兰 TIOBE 公司自 2001 年开始每月定期发布，用于评估程序设计语言的流行度。近几年 Python 语言的流行度快速攀升，目前已跃居榜首 (图 1.1)。

Google 公司的决策是"Python where we can, C++ where we must." 即仅在性能要求高和需要对内存进行精细管理的场合使用 C+ +，而在其他场合都使用 Python。原因是用 Python 语言开发软件的效率更高，并且易于维护和复用。

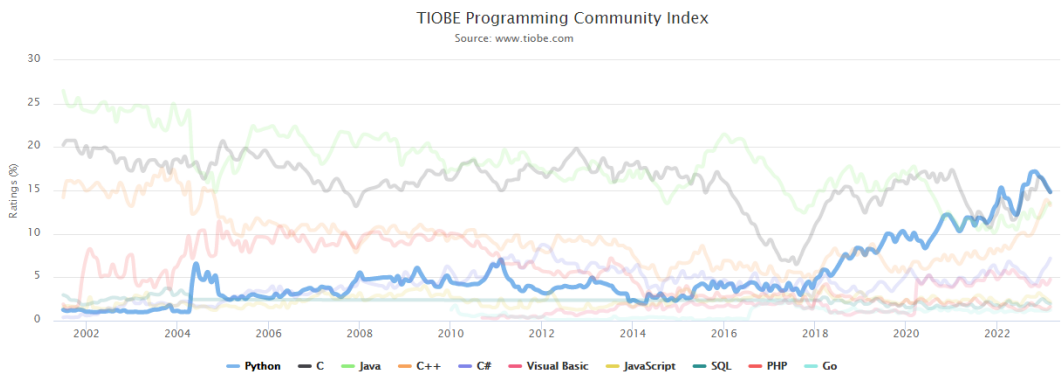


图 1.1: TIOBE 指数 2023 年 3 月

1.7 Python 语言的发展历史

Python 由荷兰程序员 Guido van Rossum 于 1989 年基于 ABC 教学语言设计和开发，其命名是源于 BBC 的喜剧节目 "Monty Python's Flying Circus"。Guido 发现像他这样熟练掌握 C 语言的人，在用 C 实现功能时也不得不耗费大量的时间。shell 作为 UNIX 系统的解释器已经

长期存在，它可以像胶水一样将 UNIX 的许多功能连接在一起。许多需要用上百行语句的 C 语言程序实现的功能只需几行 shell 语句就可以完成。然而，shell 的本质是调用命令，并不是一种通用语言。所以 Guido 希望有一种语言可以兼具 C 和 shell 的优点。Guido 总结的设计目标列举如下：

1. 一种简单直观的语言，并与主要竞争者一样强大；
2. 代码像纯英语那样容易理解；
3. 适用于短期开发的日常任务；
4. 开源，以便任何人都可以为它做贡献。

Python 软件基金会 (Python Software Foundation, https://www.python.org/psf/) 是 Python 的版权持有者，致力于推动 Python 开源技术和发布 Python 的新版本。2008 年 12 月，Python3.0 发布，这是一次重大的升级，与 Python2.x 不兼容。2019 年 10 月，Python3.8 发布。2020 年 10 月，Python3.9 发布。2021 年 10 月，Python3.10 发布。2022 年 10 月，Python3.11 发布。

1.8 Python 语言的特点

Python 是一种简单易学、动态类型、功能强大、面向对象、解释执行、易于扩展的开源通用型语言。Python 的主要特点如下。

1. 语法简单清晰，程序容易理解。
2. 使用变量之前无需声明其类型，变量的类型由运行时系统推断。
3. 标准库提供了数据结构、系统管理、网络通信、文本处理、数据库接口、图形系统、XML 处理等丰富的功能。
4. Python 社区提供了大量的第三方模块，使用方式与标准库类似。它们的功能覆盖科学计算、图形用户界面、Web 开发、系统管理等多个领域。
5. 面向对象, 适于大规模软件开发。
6. 解释器提供了一个交互式的开发环境，程序无需编译和链接即可执行。
7. 如果需要一段关键代码运行得更快或者不希望公开某些代码，可以把这部分代码用 C 或 C++编写并编译成扩展库，然后在 Python 程序中使用它们。
8. 与 C 等编译执行的语言相比，Python 程序的运行效率更低。

1.9 Python 科学计算环境

科学计算使用的程序设计语言主要包括 Fortran、C、C++、MATLAB 和 Python。前三种语言称为低层语言，后两种称为高层语言。用低层语言开发的程序比用高层语言开发的程序运行效率更高，但开发耗时更长，软件维护代价也更高。由于人力成本不断上升而硬件成本不断下降，当前趋势是用高层语言开发程序，并通过接口访问用低层语言开发的软件库。

Python 语言及其众多的扩展库 (NumPy、SciPy、SymPy 和 Matplotlib 等) 所构成的开发环境十分适合开发科学计算应用程序。NumPy 提供了 N 维数组类型以及数组常用运算的高效实现。SciPy 基于 NumPy 实现了大量的数值计算算法，包括矩阵计算、插值、数值积分、代数方程求解、最优化方法、常微分方程求解、信号和图像处理等。SymPy 实现了一种进行符号计算的计算机代数系统，可以进行矩阵计算、表达式展开和化简、微积分、代数方程求解和常微分方程求解等。Matplotlib 提供了丰富的绘图功能，可绘制多种二维和三维图示，直观地呈现科学计算的输入数据和输出结果。

和美国公司开发的付费商业软件 MATLAB 相比，Python 科学计算环境的优势是免费开源并且没有国际政治风险。

1.10 选课须知

1.10.1 选课要求和课程安排

本课程面向满足以下条件的学生：

- 已经学习过微积分和线性代数；
- 对于学习计算机科学和数学具有浓厚兴趣；
- 自备笔记本电脑。

所有学习资料 (讲义和源程序等) 将发送至选课同学的电子信箱。每次课程包括两节理论课和一节实验课。考核方式是大作业 (75%)、实验课作业 (20%) 和课堂参与 (5%)。每次实验课作业需要在 Blackboard 系统提交，如果提交时间超过截止时间则扣除 50%分数。

1.10.2 学习方法

1946 年著名学习专家爱德加 • 戴尔发现不同学习方式的学习效果按从高到低呈金字塔分布。总体上，被动学习 (听讲、阅读、视听、演示) 的效果低于主动学习 (讨论、实践、教他人)，其中听老师讲课是所有学习方式中效果最差的，最好的是“教他人”或“马上应用”。

学习本课程的最有效方法是编写和调试程序，在实践中掌握知识和提升能力。

编程中遇到自己解决不了的问题怎么办？

- 仔细阅读讲义、书籍和技术文档；
- 上网搜索：百度 (www.baidu.com), 微软 Bing 国际版 (https://cn.bing.com/?ensearch=1)，stackoverflow，...；
- 在实验课和其他同学讨论，或者在 Blackboard 平台 (https://www.bb.ustc.edu.cn/) 的讨论版 (图 1.2) 发帖；
- 在实验课向老师求助，或者在课后发送电子邮件给老师。

图 1.2: Blackboard 平台的讨论版

1.10.3 大作业

大作业的要求是每位同学设计和实现一个 Python 科学计算程序并在课堂讲解和演示。采用这种方式的目的是：

- 学以致用，利用本学期所学设计和实现科学计算程序，助力自己的专业学习和科研工作；
- 训练计算思维能力；
- 训练创新能力和自主探索能力；
- 训练口头表达能力。

大作业由三部分组成：程序、文档和课堂讲解，详细规则如下。

1. 程序的主要功能是用 Python 语言实现一个科学计算程序。程序至少包含 50 行代码。程序可以是自己完全独立设计和编写的，也可以对于书籍或互联网上已有程序做一些改进。完成程序以后应使用已知正确结果的测试数据对程序进行充分测试。所提交的程序文件应是文本格式，以 py 作为后缀。

2. 文档的格式是 HTML 或 PDF。如果选择 HTML，建议使用 PythonLinks.html 提供的 HTML Composer 编写文档。使用方法是安装 SeaMonkey 浏览器以后，在窗口菜单下选择 Composer，即可打开 HTML 编辑器。文档的主要内容包括科学技术原理，设计方案，创新性描述，运行方法和参数设置，学习心得和收获、参考文献等。其中科学技术原理部分必须引用属于科技论文或科技书籍类别的参考文献。文档的内容应以文字为主，可以包含若干图片，但所有图片文件的长度总和不超过 800K 字节。如果超过，可适当缩小图片的分辨率或删除非必要图片 (例如程序运行可以生成的)。图片文件必须采用 JPG 或 PNG 压缩格式。文档中对图片的引用要使用相对路径，例如 src="images/formula.png"。

3. 课堂讲解的主要内容是基于文档介绍程序和演示程序的运行结果。课堂讲解的日期可以在本学期的最后四次课中选择。选择在某一日期讲解的同学需要提前至少一天将打包文件通过电子邮件发送给教师，电子邮件的标题必须使用以下格式：Python+学号+姓名+上课地点 (东区或西区)。打包文件采用 ZIP 格式进行压缩，原则上长度不超过 1 兆字节。比较大的数据文件无需放在打包文件中，可以在讲解前用优盘复制到讲台电脑上。打包文件解压缩后的结构应符合以下要求：文件夹名称为学号+姓名；文件夹内包含的文件后缀可以为 py,pyx,html,css,png,jpg,csv 和 txt；文档 HTML 文件所引用的所有图片文件位于 images 子文件夹中。

评分依据有以下几个方面。

1. 创新性：体现在程序所解决的问题和设计方案等方面
2. 技术含量：程序中自己独立完成的代码的数量和质量
3. 程序的易用性：展示的运行结果是否直观？如果程序需要输入大量数据，则应从文件读取数据，而不是让用户在界面上输入。
4. 学习心得和收获：学习本课程和完成大作业的过程中有哪些学习心得和收获、经验和教训？对教学有何意见建议？
5. 对于以上规则的遵守情况。
6. 在前三次课讲解的同学在评分时有加分。

以下任何一种情形将导致不及格成绩：

- 未按要求提交程序和文档，或提交的程序无法运行；
- 未在课堂讲解程序；
- 完全抄袭已有代码 (举报者将获得加分)。

其他未遵守以上规则的行为将导致扣分。例如：

- 在讲解日期当天才提交打包文件；

- 打包文件包含一些非必要文件；
- 文档内容不完整；
- 打包文件的长度超过 1 兆字节；

1.11 实验 1：安装和使用 Python 开发环境

实验目的

本实验的目的是安装 Python 开发环境，熟悉其基本功能。

实验内容

1. 安装 Python 开发环境
安装 Python 开发环境的方式有两种：

- 下载和安装 Anaconda：Anaconda 是一个开源的 Python 发行版本，包含 Python 解释器、集成开发环境 spyder、包管理器 conda 和多个科学计算库（numpy、scipy 等），可运行在 Windows、Linux 和 Mac OS 系统上。可从国内镜像网站 [2] 或 Anaconda 官网 [3] 下载 Anaconda。安装过程中可指定安装路径，路径中不可包含中文字符。
- 从 Python 官网 [4] 下载 Python 解释器并安装。然后使用 pip 包管理工具根据自己的需要安装 Python 包。本课程需要安装的包有：numpy、scipy、sympy、matplotlib 和 spyder 等。pip 提供了对 Python 包的查找、安装和卸载等功能，在命令行运行。以下举例说明：
  - 查找所安装的所有包: pip list
  - 安装 numpy: pip install numpy
  - 升级 numpy: pip install –upgrade numpy
  - 卸载 numpy: pip uninstall numpy

后三个命令以 numpy 为例，可将命令中的 numpy 替换成任何需要按照的 Python 包。
2. 设置开发环境参数
安装完成以后，在 Windows 系统中从已安装程序的列表中可以找到 Anaconda 文件夹下的 spyder 的图标，点击此图标即可运行 spyder。也可以通过在命令行 (控制台) 输入命令“spyder”运行 spyder。如果需要设置 spyder 开发环境的参数，可以点击 Tools 菜单的 Preference 菜单项，此时出现一个对话框 (图 1.3)。对话框左边的列表列举了可以修改的参数的所属类别。其中 Appearance 表示界面的外观。选中 Appearance，此时对话框中间的“Syntax highlighting theme”部分有一个下拉列表，其中的每个选项对应一种背景和语法高亮的颜色方案；“Fonts”部分可以设置字体类型和大小。

[2] https://mirrors.tuna.tsinghua.edu.cn/anaconda/archive/
[3] https://www.anaconda.com/products/individual
[4] https://www.python.org/

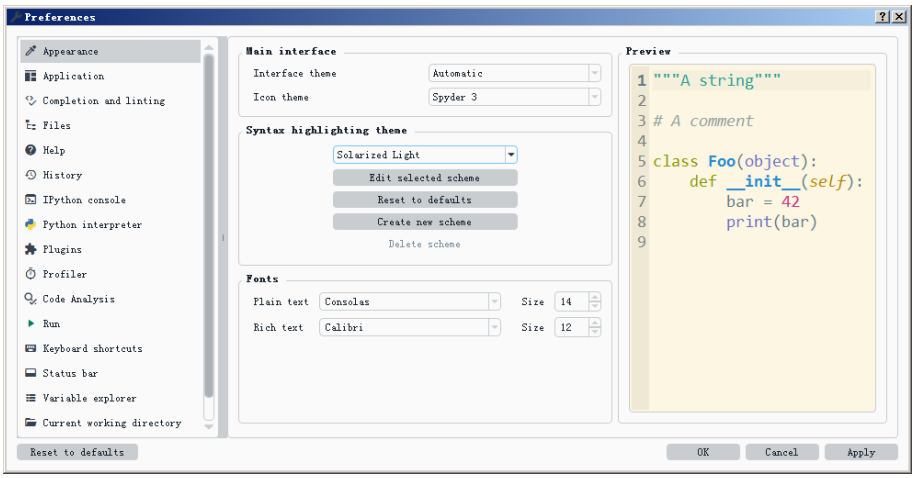


图 1.3: 设置 spyder 参数

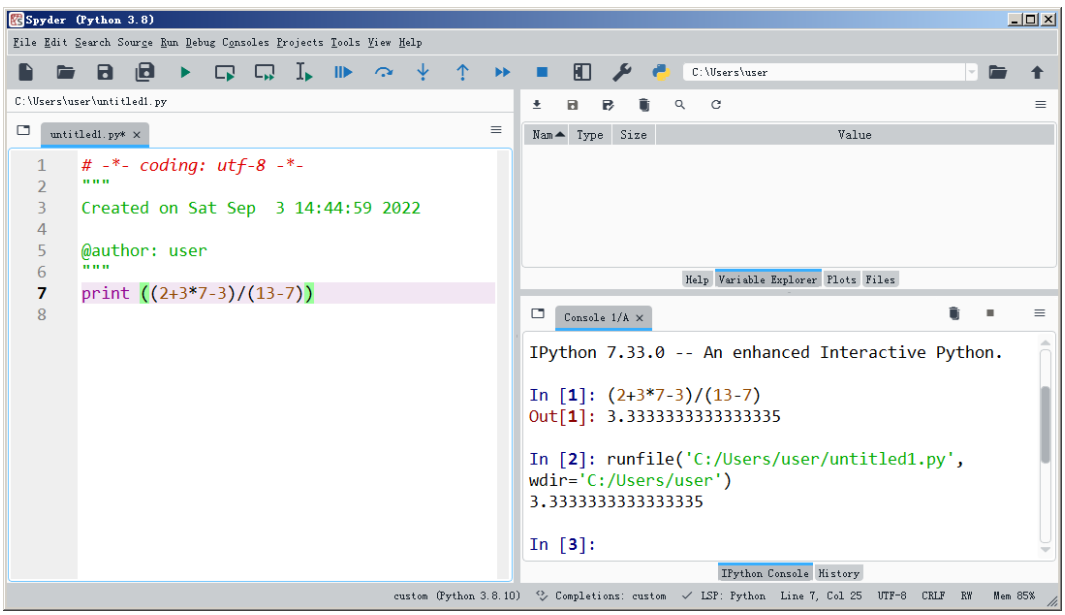


图 1.4: spyder 界面

3. 运行 Python 程序

在 spyder(图 1.4) 中运行 Python 代码的方式可以有两种, 分布适用于简短和较长的程序：

1. 在右下角的 IPython 窗口中输入一条或多条语句，然后回车；
2. 在左边的编辑窗口中输入一个完整的程序，点击 Run 菜单的 Run 菜单项执行。运行结果显示在 IPython 窗口中。

IPython 可以作为一个计算器使用，例如 1.2。其中 In[1] 表示用户输入的第一条语句，Out[1] 表示用户输入的第一条语句的执行结果。

Listing 1.2: **使用 IPython**

```
In[1]: (2+3*7-3)/(13-7)
Out[1]: 3.3333333333333335
```

第 2 章　内置数据类型及其运算

Python 语言支持面向对象编程范式，Python 程序中的所有数据都是由对象或对象之间的关系所表示。Python 程序中的每个对象包括身份 (identity，例如内存地址)，类型 (type) 和值 (value)。类型规定了数据的存储形式和可以进行的运算。例如整数类型可以进行四则运算和位运算，而浮点类型可以进行四则运算但不能进行位运算。

和 C、Java 等静态类型语言不同，Python 是一种动态类型语言，变量的类型在使用前无需声明，而是在程序运行的过程中根据变量存储的数据自动推断。动态类型的益处是程序的语法更简单，付出的代价是程序的运行效率不如静态类型语言 (如 C 和 Java 等) 程序。原因包括无法进行编译时的优化和类型推断占用了程序运行的时间等。

Python 赋值语句的一般语法形式是“变量=表达式”，其中的等号是赋值运算符，而非数学中的相等运算符。例如赋值语句 x=1 运行完成以后，变量 x 的类型即为整数类型 int，因为 x 存储的数据 1 是整数。

本章介绍了 Python 语言的常用内置 (built-in) 数据类型及其运算。Python 语言的官方文档 [PythonDoc] 详细说明了所有内置数据类型及其运算。

2.1　变量和类型

变量是计算机内存中存储数据的标识符，根据变量名称可以获取内存中存储的数据。变量的类型由其存储的数据决定。变量名只能是字母、数字或下划线的任意组合。变量名的第一个字符不能是数字。以下 Python 关键字不能声明为变量名：

Listing 2.1: **Python 关键字**

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

2.2 数值类型

数值类型包括 int(整数)、float(浮点数) 和 complex(复数)。

2.2.1 int 类型

int 类型表示任意精度的整数，可以进行的运算包括相反 (-)、加 (+)、减 (-)、乘 (*)、除 (/)、整数商 (//)、取余 (%)、乘方 (**) 和各种位运算。两个整数进行的位运算包括按位取或 (|)、按位取与 (&) 和按位取异或 (^)。单个整数进行的位运算包括按位取反 (~)、左移 (<<) 和右移 (>>)。

2.2 演示了 int 类型的运算，其中 x 和 y 通过赋值运算符 (=) 分别被赋予了整数值，因此都是 int 类型的变量。表达式是对数据进行运算的语法形式。当一个表达式中出现多种运算时，这些运算的运行次序由运算符的优先级 (precedence) 和结合性 (associativity) 确定。优先级高的运算先运行。若多个运算的优先级相同，则根据结合性是从左到右还是从右到左确定次序。括号的优先级最高，其次是乘方，再次是乘除，之后是加减。以上这些运算中，除了乘方的结合性是从右到左以外，其余运算的结合性都是从左到右。

Listing 2.2: **int 类型的运算**

```
In[1]: (2**2**3+4)/(7*(9-5))
Out[1]: 9.285714285714286
In[2]: (2**2**3+4)//(7*(9-5))
Out[2]: 9
In[3]: (2**2**3+4)%(7*(9-5))
Out[3]: 8
In[1]: (32+4)/(2*(9-5))
Out[1]: 4.5
In[2]: (32+4)//(23-13)
Out[2]: 4
In[3]: (32+4)%(23-13)
Out[3]: 6
In[4]: x=37951640090681193063801489608 0
In[5]: x**2
Out[5]: 1440326985572599996078861105607553629731714764199731993664 00
In[6]: y=120557357903313594474425387 67
In[7]: 991*y**2
Out[7]: 1440326985572599996078861105607553629731714764199731993663 99
In[8]: Out[7]-Out[5]
Out[8]: -1
In[9]: x**2-991*y**2-1
```

```
Out[9]: 0
In[10]: bin(367), bin(1981)  # 内置函数bin显示一个整数的二进制形式
Out[10]: ('0b101101111', '0b11110111101')
In[11]: bin(367 | 1981), bin(367 & 1981), bin(367 ^ 1981)
Out[11]: ('0b11111111111', '0b100101101', '0b11011010010')
In[12]: bin(~1981), bin(1981 << 3), bin(1981 >> 3)
Out[12]: ('-0b11110111110', '0b11110111101000', '0b11110111')
```

2.2.2 float 类型

float 类型根据 IEEE754 标准定义了十进制实数在计算机中如何表示为二进制浮点数。64 位二进制浮点数表示为 $(-1)^s(1+f)2^{e-1023}$。s 占 1 位表示正数和负数的符号。f 占 52 位，$1+f$ 为小数部分。e 占 11 位，$e-1023$ 为指数部分。$e=2047, f=0, s=\pm1$ 表示正无穷和负无穷。$e=2047, f\neq 0$ 表示非数值 (如 0/0)。四舍五入的相对误差为 $\frac{1}{2}2^{-52}=2^{-53}\approx 1.11\times 10^{-16}$，因此 64 位二进制浮点数对应的十进制实数的有效数字的位数为 15。

标准库以模块作为组成单位，使用某一模块之前需要用 import 语句将其导入。标准库的 sys 模块的 float_info 属性提供了 float 类型的取值范围 (max，min) 和有效数字位数 (dig) 等信息。绝对值超过 max 的数值表示为 inf(正无穷) 或-inf(负无穷)。In[1] 行用 import 语句导入 sys 模块。

Listing 2.3: **float 类型**

```
In[1]: import sys
In[2]: sys.float_info
Out[2]: sys.float_info(max=1.7976931348623157e+308, max_exp=1024,
        max_10_exp=308, min=2.2250738585072014e-308,
        min_exp=-1021, min_10_exp=-307, dig=15, mant_dig=53,
        epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

float 类型可以进行相反 (-)、加 (+)、减 (-)、乘 (*)、除 (/)、整数商 (//)、取余 (%) 和乘方 (**) 等运算。由于需要进行进制转换和表示位数的限制，实数在计算机中的表示可能存在误差，称为舍入误差 (rounding error)。例如十进制的 0.1 表示为二进制无限循环小数 $0.000\overline{1100}$，在截断后导致舍入误差。舍入误差在计算过程中可能不断积累，导致最终计算结果出现较大的误差。例如：1991 年 2 月 25 日海湾战争期间，爱国者导弹防御系统运行 100 个小时以后积累了 0.3422 秒的误差，导致其未能拦截来袭导弹，造成 28 名美军士兵死亡 [1]。因此，在需要高精度计算结果的场合，进行浮点数计算时必须对误差的产生和积累进行严密的分析和控制。由于误差的存在，在计算过程中比较两个浮点数是否相等的方法是判断这两个数的差的绝对值是否小于一个预先确定的值较小的正数 (例如 10^{-10})。

[1] https://www-users.cse.umn.edu/~arnold/disasters/patriot.html

2.4 演示了 float 类型的运算，其中 max 通过赋值运算符 (=) 被赋予了浮点数值，因此是 float 类型的变量。“In[4]” 行包含了多条语句，语句之间用分号 (;) 分隔，最后一条语句用来输出 max 的值。float 类型的数值的表示形式有两种：十进制和科学计数法。科学计数法用 e 或 E 表示指数部分，例如 1.2345678909876543e38 表示 $1.2345678909876543 \times 10^{38}$。

Listing 2.4: **float 类型的运算**

```
In[1]: 10000*(1.03)**5
Out[1]:  11592.740743
In[2]: (327.6-78.65)/(2.3+0.13)**6
Out[2]: 1.2091341548676164
In[3]: 4.5-4.4
Out[3]: 0.09999999999999964    # 精确值应当是0.1
In[4]: import sys; max = sys.float_info.max; max
Out[4]: 1.7976931348623157e+308
In[5]: max*1.001
Out[5]: inf    # 向上溢出(overflow)导致结果是无穷大
In[6]: sys.float_info.min*0.0000000000000001
Out[6]: 0    # 向下溢出(underflow)导致结果是0
In[6]: 1.2345678909876543 21e38
Out[6]: 1.2345678909876543e+38    # 有效数字的位数不能超过15
In[7]: import numpy as np
In[8]: (2**(2046-1023))*((1 + sum(0.5**np.arange(1, 53))))
Out[8]: 1.7976931348623157e+308
In[9]: (2**(1-1023))*(1+0)
Out[9]: 2.2250738585072014e-308
```

2.2.3 complex 类型

complex 类型表示实部和虚部为 float 类型的复数，可以进行相反 (-)、加 (+)、减 (-)、乘 (*)、除 (/)、和乘方 (**) 等运算。

2.5 演示了 complex 类型的运算，其中 x 和 y 通过赋值运算符 (=) 分别被赋予了复数值，因此是 complex 类型的变量。

Listing 2.5: **complex 类型的运算**

```
In[1]: x = 3 - 5j;
In[2]: y = -(6 - 21j);
In[3]: (x+y)/(x - y**2)*(x**3 + y - 3j)
Out[3]: (-2.7021404738144748-6.422968879823101j)
In[4]: x.real
```

```
Out[4]: 3.0
In[5]: x.imag
Out[5]: -5.0
In[6]: x.conjugate()
Out[6]: (3+5j)
```

2.2.4 数值类型的内置函数

对于一个整数或实数，abs 函数获取其绝对值。对于一个复数，abs 函数获取其模。int 和 float 函数可以进行这两种类型的相互转换。complex 函数从两个 float 类型的数值生成一个 complex 类型的数值。pow 计算乘方，等同于**运算符。

2.6 演示了这些函数的使用。

Listing 2.6: 数值类型的内置函数

```
In[1]: x = -15.6;
In[2]: y = int(x); y
Out[2]: -15
In[3]: type(y)
Out[3]: int
In[4]: x=float(y); x
Out[4]: -15.0
In[5]: type(x)
Out[5]: float
In[6]: z = complex(abs(x),(2 - y)); z
Out[6]: (15+17j)
In[7]: abs(z)
Out[7]: 22.671568097509265
In[8]: pow(z, 1.28)
Out[8]: (25.35612170271214+48.0468434395756j)
In[9]: pow(1.28, z)
Out[9]: (-20.006681963602528-35.28791909603722j)
```

2.2.5 math 模块和 cmath 模块

math 模块定义了圆周率 math.pi、自然常数 math.e 和以实数作为自变量和因变量的常用数学函数。cmath 模块定义了以复数作为自变量和因变量的常用数学函数。

表 2.1 列出了 math 模块的部分函数。2.7 演示了求解一元二次方程 $ax^2 + bx + c = 0$ $(a \neq 0)$

函数名称	函数定义和示例
math.ceil(x)	大于等于 x 的最小整数: math.ceil(-5.3) 值为-5
math.floor(x)	小于等于 x 的最大整数: math.floor(-5.3) 值为-6
math.factorial(x)	x 的阶乘: math.factorial(5) 值为 120
math.sqrt(x)	x 的平方根: math.sqrt(3) 值为 1.7320508075688772
math.exp(x)	以自然常数 e 为底的指数函数: math.exp(2) 值为 7.38905609893065
math.log(x)	以自然常数 e 为底的对数函数: math.log(7.38905609893065) 值为 2.0
math.log(x, base)	以 base 为底的对数函数: math.log(7.38905609893065, math.e) 值为 2.0
math.log2(x)	以 2 为底的对数函数: math.log2(65536) 值为 16.0
math.log10(x)	以 10 为底的对数函数: math.log10(1e-19) 值为-19.0
三角函数	math.sin(x) math.cos(x) math.tan(x)
反三角函数	math.asin(x) math.acos(x) math.atan(x)
双曲函数	math.sinh(x) math.cosh(x) math.tanh(x)
反双曲函数	math.asinh(x) math.acosh(x) math.atanh(x)

表 2.1: math 模块的部分函数

的根。当判别式 $\Delta = b^2 - 4ac$ 为负时两个根为复数，使用 math 模块的求平方根函数 (sqrt) 会报错 (ValueError)。此时需要使用 cmath 模块的求复数平方根的函数。

Listing 2.7: **求解一元二次方程**

```
In[1]: import math; a=2; b=6; c=1
In[2]: r1 = (-b + math.sqrt(b**2 - 4*a*c))/(2*a); r1
Out[2]: -0.17712434446770464
In[3]: r2 = (-b - math.sqrt(b**2 - 4*a*c))/(2*a); r2
Out[3]: -2.8228756555322954
In[4]: a=2; b=6; c=8
In[5]: r1 = (-b + math.sqrt(b**2 - 4*a*c))/(2*a); r1
Out[5]: ValueError: math domain error
In[6]: import cmath
In[7]: r1 = (-b + cmath.sqrt(b**2 - 4*a*c))/(2*a); r1
Out[7]: (-1.5+1.3228756555322954j)
In[8]: r2 = (-b - cmath.sqrt(b**2 - 4*a*c))/(2*a); r2
Out[8]: (-1.5-1.3228756555322954j)
```

2.3 bool 类型

int 类型和 float 类型的数据可以使用以下这些关系运算符进行比较: >(大于)、<(小于)、>=(大于等于)、<=(小于等于)、==(等于)、!=(不等于)。比较的结果属于 bool 类型，只有

两种取值：True 和 False。bool 类型的数据可以进行三种逻辑运算，按优先级从高到低依次为 not(非)、and(与) 和 or(或)。表 2.2 列出了逻辑运算的规则。

x	y	x and y	x or y	not x
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

表 2.2: 逻辑运算的规则

2.8 演示了使用比较运算符和逻辑运算符判断一个年份是否闰年。

Listing 2.8: **bool 类型的运算**

```
In[1]: year = 1900
In[2]: (year % 4 == 0 and year % 100 != 0) or year % 400 == 0
Out[2]: False
In[3]: year = 2020
In[4]: (year % 4 == 0 and year % 100 != 0) or year % 400 == 0
Out[4]: True
In[5]: year = 2022
In[6]: (year % 4 == 0 and year % 100 != 0) or year % 400 == 0
Out[6]: False
```

2.4 序列类型

序列类型 (Sequence Types) 可以看成一个存储数据元素的容器，这些元素是有序的，每个元素对应一个索引值。用 n 表示序列的长度，则索引值的取值范围是区间 $[-n-1,\ n-1]$ 内的所有整数。序列中的第 k 个元素 $(1 \le k \le n)$ 对应的索引值有两个: $k-1$ 和 $k-n-1$。例如，索引值 0 和 $-n$ 都对应第 1 个元素，索引值 $n-1$ 和 -1 都对应第 n 个元素。

序列类型包括 list(列表)、tuple(元组)、range(范围)、str(字符串)、bytes、bytearray、和 memoryview 等 [PythonDoc]。range 通常用于 for 语句 (第三章)。bytes、bytearray 和 memoryview 是存储二进制数据的序列类型。

2.4.1 list 类型 (列表) 和 tuple 类型 (元组)

list 和 tuple 通常用来存储若干同一类型的元素。list 类型的语法是用方括号括起的用逗号分隔的若干元素，例如 [2,3,5,7,11] 是一个存储了五个质数的列表。tuple 类型的语法是用圆括号

括起的用逗号分隔的若干元素，例如 (2,3,5,7,11) 是一个存储了五个质数的元组。list 是可变的，即其中存储的元素可以被修改。tuple 和 range 是不可变的。

表 2.3 列出了 list 类型和 tuple 类型的共有运算。示例中参数 s 的值为 [2,3,5,7,11]，参数 t 的值为 [13,17]。

运算名称	运算定义和示例
x in s	若 s 中存在等于 x 的元素，则返回 True，否则返回 False。 例: 5 in s 值为 True
x not in s	若 s 中不存在等于 x 的元素，则返回 True，否则返回 False。 例: 7 not in s 值为 False
s + t	返回将 s 和 t 连接在一起得到的序列。例: s + t 值为 [2,3,5,7,11,13,17]
s*n 或 n*s	返回将 s 重复 n 次得到的序列。 例: s*3 值为 [2,3,5,7,11,2,3,5,7,11,2,3,5,7,11]
s[n]	返回 s 中索引值为 n 的元素。例: s[4] 值为 11
s[i:j]	返回 s 中索引值从 i 到 j(不包含 j) 的所有元素构成的序列。 例: s[1:4] 值为 [3, 5, 7]
s[i:]	返回 s 中索引值从 i 开始到最后的所有元素构成的序列。 例: s[1:] 值为 [3, 5, 7, 11]
s[:j]	返回 s 中索引值从 0 开始到 j(不包含 j) 的所有元素构成的序列。 例: s[:3] 值为 [2, 3, 5]
s[i:j:k]	返回 s 中索引值属于从 i 到 j(不包含 j) 且等差为 k 的等差数列的所有元素构成的序列。例: s[1:4:2] 值为 [3, 7]
len(s)	返回 s 的长度。例: len(s) 值为 5
min(s)	返回 s 中的最小元素。例: min(s) 值为 2
max(s)	返回 s 中的最大元素。例: max(s) 值为 11
s.index(x)	若 s 中存在元素 x，则返回 x 在 s 中第一次出现的索引值，否则报错。 例: s.index(7) 值为 3
s.index(x, i)	若 s[i:] 中存在元素 x，则返回 x 在 s 中第一次出现的索引值，否则报错。 例: s.index(7,1) 值为 3
s.index(x, i, j)	若 s[i:j] 中存在元素 x，则返回 x 在 s 中第一次出现的索引值，否则报错。 例: s.index(7,1,4) 值为 3
s.count(x)	返回 x 在 s 中出现的次数。例: s.count(8) 值为 0

表 2.3: list 类型和 tuple 类型的共有运算

表 2.4 列出了 list 类型的特有运算，每种运算运行之前参数 s 的初始值为 [2,7,5,3,11]，参数 t 的值为 [13,21,17]。

运算名称	运算定义和示例
s[i] = x	将 s 中索引值为 i 的元素修改为 x。例: 运行 s[3] = 8 后 s 值为 [2,3,5,8,11]
s[i:j] = t	将 s 中索引值从 i 到 j(不包含 j) 的所有元素构成的序列修改为 t。 例: 运行 s[1:4] = t 后 s 值为 [2, 13, 21, 17, 11]
del s[i:j]	删除 s 中索引值从 i 到 j(不包含 j) 的所有元素构成的序列。 例: 运行 del s[2:4] 后 s 值为 [2, 7, 11]
s[i:j:k] = t	将 s 中索引值属于从 i 到 j(不包含 j) 且等差为 k 的等差数列的所有元素构成的序列修改为 t。例: 运行 s[0:5:2] = t 后 s 值为 [13, 7, 21, 3, 17]
del s[i:j:k]	删除 s 中索引值属于从 i 到 j(不包含 j) 且等差为 k 的等差数列的所有元素构成的序列。例: 运行 del s[0:5:2] 后 s 值为 [7, 3]
s.append(x)	添加元素 x 到列表 s 中使其成为最后一个元素，即追加元素 x。 例: 运行 s.append(13) 后 s 值为 [2, 7, 5, 3, 11, 13]
s.clear()	删除 s 中所有元素。例: 运行 s.clear() 后 s 值为 []
s.copy()	返回 s 的一个副本。例: s.copy() 返回 [2,7,5,3,11]
s.extend(t)	将序列 t 添加到 s 的后面，即追加序列 t。 例: 运行 s.extend(t) 后 s 值为 [2, 7, 5, 3, 11, 13, 21, 17]
s += t	同上
s *= n	修改 s 为将 s 重复 n 次得到的序列 例: 运行 s *= 3 后 s 值为 [2, 7, 5, 3, 11, 2, 7, 5, 3, 11, 2, 7, 5, 3, 11]
s.insert(i, x)	添加元素 x 到列表 s 中使其成为索引值为 i 的元素。 例: 运行 s.insert(2, 19) 后 s 值为 [2, 7, 19, 5, 3, 11]
s.pop(i)	删除 s 中索引值为 i 的元素并将其返回。 例: 运行 s.pop(3) 返回 3，且 s 的值为 [2, 7, 5, 11]
s.remove(x)	删除 s 中第一次出现的元素 x。例: 运行 s.remove(11) 后 s 值为 [2, 7, 5, 3]
s.reverse()	将 s 中所有元素的次序反转。例: 运行 s.reverse() 后 s 值为 [11, 3, 5, 7, 2]
s.sort()	将 s 中所有元素按从小到大的次序排序。 例: 运行 s.sort() 后 s 值为 [2, 3, 5, 7, 11]
s.sort(reverse =True)	将 s 中所有元素按从大到小的次序排序。 例: 运行 s.sort(reverse=True) 后 s 值为 [11, 7, 5, 3, 2]

表 2.4: list 类型的特有运算

2.9 演示了 list 类型和 tuple 类型的一些运算。其中组成列表 d 的每个元素 (a 和 b) 本身也是一个列表，类似 d 这样的列表称为嵌套列表。获取 d 中的数据需要使用两层索引。对于 d 而言，d[0] 等同于 a，d[1] 等同于 b。因此，d[0][4] 等同于 a[4]，d[1][0:5:2] 等同于 b[0:5:2]。

Listing 2.9: **list 类型和 tuple 类型的运算**

```
In[1]: a=[1,3,5,7,9]; b=[2,4,6,8,10]
In[2]: c=a+b; c
Out[2]: [1, 3, 5, 7, 9, 2, 4, 6, 8, 10]
In[3]: c.sort(); c
Out[3]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
In[4]: d=[a,b]; d
Out[4]: [[1, 3, 5, 7, 9], [2, 4, 6, 8, 10]]
In[5]: c[2:10:3]
Out[5]: [3, 6, 9]
In[6]: c[-1:-9:-4]
Out[6]: [10, 6]
In[7]: d[0][4]
Out[7]: 9
In[8]: d[1][0:5:2]=d[0][2:5]; d
Out[8]: [[1, 3, 5, 7, 9], [5, 4, 7, 8, 9]]
In[9]: b
Out[9]: [5, 4, 7, 8, 9]
In[10]: b.clear(); b
Out[10]: []            # b是空列表
In[11]: f = (); (type(f), id(f))  # f是空元组
Out[11]: (tuple, 8912936)
In[12]: f = (3,); (f, id(f))
Out[12]: ((3,), 182697528)        # f是含有一个元素的元组
In[13]: f += (4, 5); (f, id(f))
Out[13]: ((3, 4, 5), 187247080)
In[14]: f[2] = 6          # 元组中存储的元素不可以被修改，所以报错
TypeError: 'tuple' object does not support item assignment
```

2.4.2 str 类型 (字符串)

str 类型表示不可变的使用 Unicode 编码的字符构成的序列，即字符串。Unicode 是一个字符编码的国际标准，为人类语言中的每个字符设定了统一并且唯一的二进制编码。ord 函数可获取一个字符对应的 Unicode 编码值，例如 ord('A') 值为 65。chr 函数可获取一个 Unicode 编码值对应的字符，例如 chr(90) 值为'Z'。UTF-8 是一种广泛使用的 Unicode 实现方式，即用

一个或多个字节存储二进制编码。字符串可用单引号、双引号或三个相同的单引号或双引号括起。单引号和双引号可以相互嵌套。用三引号括起的字符串可以包含程序中的多行语句 (包括其中的换行符和空格), 常作为函数和模块的注释。

str 类型除了提供表 2.3 列出的运算以外，还提供了一些特有运算。它们分为两类：返回字符串的运算和不返回字符串的运算。由于 str 类型是不可变的，第一类运算不会修改作为参数输入的字符串。如果一种运算返回的字符串的字符序列和参数不同，则返回的字符串是该运算新创建的。

表 2.5 列出了返回字符串的部分特有运算，示例中参数 s 的值为'abc123abc'，参数 t 的值为'bc'，参数 u 的值为'BC'。

运算名称	运算定义和示例
s.lstrip(t)	若 s 有某个完全由 t 中的字符组成的最长前缀，则返回一个字符串，其字符序列等于从 s 中删除此前缀得到的字符序列；否则返回 s。 若 t 缺失或值为 None，则在以上操作中将 t 解释为空白字符。 例: s.lstrip(t) 的值为'abc123abc'。
s.rstrip(t)	与 s.lstrip(t) 类似，区别在于前缀改为后缀。 例: s.rstrip(t) 的值为'abc123a'。
s.strip(t)	等同于先运行 s.lstrip(t)，再运行 s.rstrip(t)。
s.replace(t, u)	若 s 中存在子串 t，则返回一个字符串，其字符序列等于 将 s 中出现的所有子串 t 替换成 u 得到的字符序列；否则返回 s。 例: s.replace(t, u) 的值为'aBC123aBC'。
s.replace(t, u, k)	与 s.replace(t, u) 的区别是仅替换 s 的前 k 个子串 t。 例: s.replace(t, u, 1) 的值为'aBC123abc'。
s.split(t)	以 t 作为分隔符，将 s 分割成若干子串，再返回由这些子串 构成的列表。若 t 缺失或值为 None，则将 t 解释为空白字符。 例: s.split(t) 的值为 ['a', '123a', ''']，其中'' 表示空串。
t.join(ss)	将列表 ss 中包含的所有字符串以 t 作为分隔符连接在一起， 再返回连接的结果。t 可以是空串。 例: t.join(['a', '123a', '']) 的值为 s。

表 2.5: str 类型的返回字符串的部分特有运算

str 类型的%运算符可以使用多种转换说明符为各种类型的数值生成格式化输出。%左边是一个字符串，其中可包含一个或多个转换说明符。%右边包含一个或多个数值，这些数值必须和转换说明符一一对应。如果有多个数值，这些数值必须置入一个元组中。

2.10 中的"%-16.8f" 中的负号表示左对齐 (无负号表示右对齐)，16 表示所生成的字符串的长度，在点以后出现的 8 表示输出结果在小数点以后保留 8 位数字。

Listing 2.10: **格式化输出**

```
In[1]: "%-16.8f" % 345.678987654321012
Out[1]: '345.67898765␣␣␣␣'
In[2]: "%16.8g" % 3.45678987654321012e34
Out[2]: '␣␣␣3.4567899e+34'
In[3]: "%16X␣%-16d" % (345678987654, 987654321012)
Out[3]: '␣␣␣␣␣␣507C12C186␣987654321012␣␣␣␣'
```

表 2.6 列出了常用的转换说明符及其定义。

转换说明符名称	转换说明符定义
%d、%i	转换为带符号的十进制整数
%o	转换为带符号的八进制整数
%x、%X	转换为带符号的十六进制整数
%e、%E	转换为科学计数法表示的浮点数（e 小写、E 大写）
%f、%F	转换为十进制浮点数
%g、%G	智能选择使用 %f 或%e 格式 (%F 或 %E 格式)
%c	将整数转换为单个字符的字符串
%r	使用 repr() 函数将表达式转换为字符串
%s	使用 str() 函数将表达式转换为字符串

表 2.6: 常用的转换说明符

表 2.7 和 2.8 列出了不返回字符串的部分特有运算，示例中参数 s 的值为'abc123abc'，参数 t 的值为'bc'。

运算名称	运算定义和示例
s.startswith(t)	若 s 以 t 为前缀，则返回 True，否则返回 False。 例: s.startswith(t) 的值为 False。
s.startswith(t, start)	若 s[start:] 以 t 为前缀，则返回 True，否则返回 False。 例: s.startswith(t, 7) 的值为 True。
s.startswith(t, start, end)	若 s[start:end] 以 t 为前缀，则返回 True，否则返回 False。 例: s.startswith(t, 7, 9) 的值为 True。
s.endswith(t)	若 s 以 t 为后缀，则返回 True，否则返回 False。 例: s.endswith(t) 的值为 True。

表 2.7: str 类型的不返回字符串的部分特有运算

运算名称	运算定义和示例
s.endswith(t, start)	若 s[start:] 以 t 为后缀，则返回 True，否则返回 False。 例: s.endswith(t, 4) 的值为 True。
s.endswith(t, start, end)	若 s[start:end] 以 t 为后缀，则返回 True，否则返回 False。 例: s.endswith(t, 4, 8) 的值为 False。
s.find(t)	若 s 中存在子串 t，则返回 t 在 s 中第一次出现的索引值，否则返回-1。 例: s.find(t) 的值为 1。
s.find(t, start)	若 s[start:] 中存在子串 t，则返回 t 在 s 中第一次出现的索引值， 否则返回-1。例: s.find(t, 2) 的值为 7。
s.find(t, start, end)	若 s[start:end] 中存在子串 t，则返回 t 在 s 中第一次出现的索引值， 否则返回-1。例: s.find(t, 2, 8) 的值为-1。
s.rfind(t[, start[, end]])	若 s(或 s[start:] 或 [start:end]) 中存在子串 t， 则返回 t 在 s 中最后一次出现的索引值，否则返回-1。
str.isalpha(s)	若 s 中包含至少一个字符并且所有字符都是字母，则返回 True， 否则返回 False。例: str.isalpha(s) 的值为 False。
str.isdecimal(s)	若 s 中包含至少一个字符并且所有字符都是数字，则返回 True， 否则返回 False。例: str.isdecimal(s) 的值为 False。
str.isalnum(s)	若 s 中包含至少一个字符并且所有字符都是字母或数字， 则返回 True，否则返回 False。例: str.isalnum(s) 的值为 True。
str.islower(s)	若 s 中包含至少一个字母并且所有字母都是小写，则返回 True， 否则返回 False。例: str.islower(s) 的值为 True。
str.isupper(s)	与 str.islower(s) 类似，区别在于判断是否所有字母都是大写。

表 2.8: str 类型的不返回字符串的部分特有运算 (续)

2.11 演示了 str 类型的一些运算。

Listing 2.11: **str 类型的运算**

```
In[1]: a = 'allows␣embedded␣"double"␣quotes'; a
Out[1]: 'allows␣embedded␣"double"␣quotes'
In[2]: b = "allows␣embedded␣'single'␣quotes"; b
Out[2]: "allows␣embedded␣'single'␣quotes"
In[3]: c = """a=[1,3,5,7,9]; b=[2,4,6,8,10]
  ...: c=a+b
  ...: c.sort(); c"""
In[4]: c
Out[4]: 'a=[1,3,5,7,9];␣b=[2,4,6,8,10]\nc=a+b\nc.sort();␣c'
In[5]: d = [a.startswith('allow'), a.startswith('allou')]; d
Out[5]: [True, False]
```

```
In[6]: e = [a.startswith('embee', 7), a.endswith('quo', 3, -3)]
Out[6]: [False, True]
In[7]: f = [a.find('em', 3, 6), a.find('em', 7)]; f
Out[7]: [-1, 7]
In[8]: g = '␣␣hello?!!'
In[9]: h = [g.lstrip(), g.rstrip('!?'), g.strip('␣!')]; h
Out[9]: ['hello?!!', '␣␣hello', 'hello?']
In[10]: a.replace('e', 'x', 1)
Out[10]: 'allows␣xmbedded␣"double"␣quotes'
In[11]: a.replace('e', 'yy', 3)
Out[11]: 'allows␣yymbyyddyyd␣"double"␣quotes'
In[12]: a.split('e')
Out[12]: ['allows␣', 'mb', 'dd', 'd␣"doubl', '"␣quot', 's']
```

2.5 set 类型 (集合)

set 类型可以看成一个存储数据元素的容器，存储的元素是无序且不可重复的，类似于数学中定义的集合。set 类型的语法是用大括号括起的用逗号分隔的若干数据。例如{2,3,5,7,11}是一个存储了五个质数的集合。frozenset 类型是不可变的 set 类型。表 2.9 列出了 set 类型的常用运算，frozenset 类型可进行表中除了 add 和 remove 以外的运算。

运算名称	运算定义
len(s)	返回 s 的长度，即其中存储了多少个元素。
x in s	若 s 中存在等于 x 的元素，则返回 True，否则返回 False。
x not in s	若 s 中存在等于 x 的元素，则返回 False，否则返回 True。
s.isdisjoint(t)	若 s 和 t 的交集非空，则返回 True，否则返回 False。
s.issubset(t)	若 s 是 t 的子集，则返回 True，否则返回 False。
s<=t、s<t	若 s 是 t 的子集 (真子集)，则返回 True，否则返回 False。
s.issuperset(t)	若 s 是 t 的超集，则返回 True，否则返回 False。
s>=t、s>t	若 s 是 t 的超集 (真超集)，则返回 True，否则返回 False。
s.union(t)	返回 s 和 t 的并集。
s.intersection(t)	返回 s 和 t 的交集。
s.difference(t)	返回 s 和 t 的差集。
s.add(x)	向 s 中添加元素 x。若 s 中存在 x，则 s 不发生变化。
s.remove(x)	从 s 中移除元素 x。若 s 中不存在 x，则报错。

表 2.9: set 类型的常用运算

2.12 演示了 set 类型的一些运算。

Listing 2.12: **set 类型的运算**

```
In[1]: l = [2,3,5,3,9,2,7,8,6,3]; (l, type(l))
Out[1]: ([2, 3, 5, 3, 9, 2, 7, 8, 6, 3], list)
In[2]: s = set(l); (s, type(s))
Out[2]: ({2, 3, 5, 6, 7, 8, 9}, set)
In[3]: t = set([11, 2, 7, 3, 5, 13])
In[4]: s.union(t)
Out[4]: {2, 3, 5, 6, 7, 8, 9, 11, 13}
In[5]: s.intersection(t)
Out[5]: {2, 3, 5, 7}
In[6]: s.difference(t)
Out[6]: {6, 8, 9}
In[7]: t.clear(); t
Out[7]: set()          # t是空集合
```

2.6 dict 类型 (字典)

dict 类型可以看成一个存储数据元素的容器，存储的每个元素由两部分组成: 键 (key) 和其映射到的值 (value)。键的类型必须是可以求哈希值的 (包括但不限于 int、str、tuple、frozenset 和用户自定义的类等)，但不能是可变的 (包括但不限于 list、set 和 dict 等)。dict 类型的语法是用大括号括起的多个元素，每个元素内部用冒号分隔键和值，元素之间用逗号分隔。例如 {2:4, 3:9, 5:25, 7:49 , 11:121}是一个存储了五个元素的字典，每个元素的键是一个整数，其映射到的值是键的平方。

表 2.10 列出了 dict 类型的常用运算。
2.13 以一个通讯录为实例演示了 dict 类型的一些运算。通讯录的每个元素的键是一个联系人的姓名，其映射到的值是该联系人的电话号码。

Listing 2.13: **dict 类型的运算**

```
In[1]: contacts={"Tom":12345, "Jerry":54321, "Mary":23415}
In[2]: contacts
Out[2]: {'Tom': 12345, 'Jerry': 54321, 'Mary': 23415}
In[3]: contacts["Jerry"]=54123; contacts["Betty"]=35421; contacts
Out[3]: {'Tom': 12345, 'Jerry': 54123, 'Mary': 23415,
         'Betty': 35421}
In[4]: contacts.keys()
Out[4]: dict_keys(['Tom', 'Jerry', 'Mary', 'Betty'])
In[5]: ['Tommy' in contacts, 'Betty' in contacts]
Out[5]: [False, True]
```

运算名称	运算定义
len(d)	返回 d 的长度，即其中存储了多少个元素。
key in d	若 s 中存在键等于 key 的元素，则返回 True，否则返回 False。
key not in d	若 s 中存在键等于 key 的元素，则返回 False，否则返回 True。
d[key] = value	若 d 中存在键等于 key 的元素，则将其对应的值修改为 value。 否则在 d 中添加一个元素，其键和值分别为 key 和 value。
del d[key]	若 d 中存在键等于 key 的元素则将其删除，否则报错。
clear()	删除所有元素。
get(key[, default])	若 d 中存在键等于 key 的元素则返回其对应的值，否则返回 default。 若未提供 default，则返回 None。
pop(key[, default])	若 d 中存在键等于 key 的元素则将其删除，然后返回其对应的值， 否则返回 default。若未提供 default，则返回 None。
items()	返回 d 中所有元素，对于每个元素返回一个由键和值组成的元组。
keys()	返回 d 中所有元素的键。
values()	返回 d 中所有元素的值。

表 2.10: dict 类型的常用运算

```
In[6]: (contacts.pop('Jerry'), contacts)
Out[6]: (54123, {'Tom': 12345, 'Mary': 23415, 'Betty': 35421})
In[7]: (contacts.pop('Tommy', None), contacts)
Out[7]: (None, {'Tom': 12345, 'Mary': 23415, 'Betty': 35421})
In[8]: contacts.clear(); contacts
Out[8]: {}          # contacts是空字典
```

2.7 实验 2：内置数据类型及其运算

实验目的

本实验的目的是掌握常用的类型 (float 和 list) 的运算。

提交方式

在 Blackboard 提交一个文本文件 (txt 后缀)，文件中记录每道题的源程序和运行结果。可以使用 spyder 或者其他 Python IDE 完成。

实验内容

1. float 类型的计算

等额本息是一种分期偿还贷款的方式，即借款人每月按相等的金额偿还贷款本息，每月还款

金额 P 可根据贷款总额 A、年利率 r 和贷款月数 n 计算得到，公式为

$$P = \frac{\frac{r}{12}A}{1-(1+\frac{r}{12})^{-n}}$$

计算当贷款金额为 1000000，贷款时间为 30 年，年利率分别为 4% 、5%和 6%时的每月还款金额和还款总额。

答案：

(4774.152954654538, 1718695.0636756336)

(5368.216230121398, 1932557.8428437035)

(5995.505251527569, 2158381.890549925)

2. math 模块

定义三个变量“a=3; b=6; c=7”表示一个三角形的三个边的长度，使用公式

$$a^2 = b^2 + c^2 - 2bc\cos\alpha$$

$$b^2 = a^2 + c^2 - 2ac\cos\beta$$

$$c^2 = a^2 + b^2 - 2ab\cos\gamma$$

分别计算三个内角 (α,β,γ) 的度数，然后检验等式 $\alpha+\beta+\gamma=180$ 是否成立。

答案：

25.208765296758365 58.41186449479884 96.37937020844281

3. list 类型的运算

定义两个列表“s=[2,4,0,1,3,9,5,8,6,7]; t=[2,6,8,4]”，对于表 2.4 中的每种运算，先手工计算其结果，然后在 Ipython 中运行并记录输出结果。若某个运算修改了 s，在运行下一个运算之前需要再设置 s=[2,4,0,1,3,9,5,8,6,7]。若某个运算需要一些参数 (如 i, j 和 x 等)，可自行设定。

第 3 章　分支和迭代

分支和迭代是程序中常用的流程控制结构。分支的语义是根据若干条件是否满足从多个分支中选择一个运行，由 if 语句和 if-else 表达式实现。迭代的语义是当某一条件满足时反复运行一个语句块，由 for 语句、while 语句和推导式实现。

3.1　if 语句和 if-else 表达式

if 语句可以根据若干条件是否满足从多个分支中选择一个运行。if 语句可以有多种形式，以计算整数的绝对值为例说明。根据绝对值的定义，可以有以下三种实现方式：3.1，3.2 和 3.3。这三个程序的第 1 行使用内建函数 input 提示用户输入一个整数，提示信息是“Please enter an integer: ”。int 函数将用户的输 (字符串) 转换为 int 类型值后赋值给变量 x。最后一行的内建函数 print 输出字符串表达式“The absolute value of %d is %d” %(x, y) 的值。

程序 3.1 的第 3 到 4 行是只有一个分支的 if 语句，当条件表达式 (x<0) 的值为 True 时运行 if 后面的分支。程序 3.2 的第 2 到 5 行是有两个分支的 if 语句，当表达式 (x<0) 的值为 True 时运行 if 后面的分支，值为 False 时运行 else 后面的分支。程序 3.3 的第 2 行右侧是一个 if-else 表达式，它实现了同样的功能。

Listing 3.1: **单分支 if 语句计算绝对值**

```
1 x = int(input("Please␣enter␣an␣integer:␣"))
2 y = x
3 if x < 0:
4     y = -x
5 print("The␣absolute␣value␣of␣%d␣is␣%d" %(x, y))
6 # The absolute value of -32 is 32
```

Listing 3.2: **多分支 if 语句计算绝对值**

```
1 x = int(input("Please␣enter␣an␣integer:␣"))
2 if x < 0:
3     y = -x
```

```
else:
    y = x
print("The absolute value of %d is %d" %(x, y))
```

Listing 3.3: **if-else 表达式计算绝对值**

```
x = int(input("Please enter an integer: "))
y = -x if x < 0 else x
print("The absolute value of %d is %d" %(x, y))
```

程序 3.4 利用多分支 if 语句将百分制成绩转换为等级分。第 2 行将等级分的默认值设置为'F'。第 3 行判断用户输入的百分制成绩是否在有效范围内。若不在，则第 4 行将等级分设置为一个特殊标记'Z'。第 5 行的条件等价于 90<=x<=100，如果此条件成立，则第 6 行将等级分设置为'A'。如果第 5 行的条件不成立，则继续判断第 7 行的条件，该条件等价于 80<=x<90。如果此条件成立，则第 8 行将等级分设置为'B'。如果第 7 行的条件不成立，则继续判断第 9 行的条件，该条件等价于 70<=x<80。如果此条件成立，则第 10 行将等级分设置为'C'。如果第 9 行的条件不成立，则继续判断第 11 行的条件，该条件等价于 60<=x<70。如果此条件成立，则第 12 行将等级分设置为'D'。如果第 11 行的条件不成立，x 必然满足 0<=x<60，此时无需给 grade 赋值，因为 grade 的值为默认值'F'。

if 语句包含的每个分支可以是一条语句，也可以是多条语句组成的语句块。这些分支相对 if 语句必须有四个空格的缩进。唯一的例外情形是一个分支的 if 语句，例如程序 3.1 的 3 到 4 行可以写成一行: if x < 0: y = -x 。

Listing 3.4: **百分制成绩转换为等级分**

```
1  x = int(input("Please enter a score within [0, 100]: "))
2  grade = 'F'            # 默认值设置为'F'
3  if x > 100 or x < 0:
4      grade = 'Z'        # 特殊标记'Z'表示输入的百分制成绩不在有效范围内
5  elif x >= 90:
6      grade = 'A'        # x满足条件: 90≤x≤100
7  elif x >= 80:
8      grade = 'B'        # x满足条件: 80≤x<90
9  elif x >= 70:
10     grade = 'C'        # x满足条件: 70≤x<80
11 elif x >= 60:
12     grade = 'D'        # x满足条件: 60≤x<70
13 print("The grade of score %d is %c" %(x, grade))
14 # The grade of score 81 is B
```

3.2 for 语句

for 语句包含循环变量、可迭代对象和一个语句块。for 语句是一种循环语句，它的语句块可以反复运行。第二章介绍的所有序列类型 (包括 list、tuple、range 和 str 等) 和 set、dict 等类型的对象称为可迭代对象 (iterable)，即可以通过 for 语句访问其包含的所有元素，在每次迭代时循环变量的值等于一个元素。以下举例说明。

程序 3.5 输出 1 到 10 之间的所有自然数的和，第 2 行的 range(1, n+1) 生成一个 range 类型的序列 (1,2,3...,n)。

Listing 3.5: **for 语句输出 1 到 10 之间的所有自然数的和**

```
1 sum = 0; n = 10
2 for i in range(1, n+1):
3     sum += i
4 print("The␣sum␣of␣1␣to␣%d␣is␣%d" % (n, sum))
5 # The sum of 1 to 10 is 55
```

程序 3.6 输出 100 到 110 之间的所有偶数，第 2 行的 range(lb, ub+1, 2) 生成一个 range 类型的序列 (100,102,...,120), 第 3 行的 print 函数的参数 (end=' ') 的作用是在每输出一个数之后输出一个空格，而不是换行。

Listing 3.6: **for 语句输出 100 到 120 之间的所有偶数**

```
1 lb = 100; ub = 120
2 for i in range(lb, ub+1, 2):
3     print(i, end='␣')
4 # 100 102 104 106 108 110 112 114 116 118 120
```

程序 3.7 输出一个由整数组成的集合中包含的 3 的倍数。

Listing 3.7: **for 语句输出一个由整数组成的集合中所包含的 3 的倍数**

```
1 nums = {25, 18, 91, 365, 12, 78, 59}
2 for i in nums:
3     if i % 3 == 0: print(i, end='␣')
4 #  12 78 18
```

程序 3.8 实现了和 3.7 相同的功能。区别在于第 3 行如果确认 i 不是 3 的倍数，则使用 continue 语句跳过本次循环的剩余语句并开始下一次循环，否则在第 4 行输出 i。

Listing 3.8: **for 语句输出一个由整数组成的集合中所包含的 3 的倍数**

```
1 nums = {25, 18, 91, 365, 12, 78, 59}
2 for i in nums:
```

```
    if i % 3 != 0: continue
    print(i, end='␣')
```

程序 3.9 采用两种方式输出一个通讯录中的每个联系人的姓名和其电话号码。

Listing 3.9: **for 语句输出一个通讯录中的每个联系人的姓名和对应的电话号码**

```
contacts = {"Tom":12345, "Jerry":54321, "Mary":23415}

for name, num in contacts.items():
    print('%s␣->␣%d' % (name, num), end=';␣')
#  Tom -> 12345; Jerry -> 54321; Mary -> 23415;
print()
for name in contacts.keys():
    print('%s␣->␣%d' % (name, contacts[name]), end=';␣')
#  Tom -> 12345; Jerry -> 54321; Mary -> 23415;
```

程序 3.10 使用 for 语句输出一个字符串中的所有字符和其对应的 Unicode 编码值。

Listing 3.10: **for 语句输出一个字符串中的所有字符和其对应的 Unicode 编码值**

```
s = 'Python'
for c in s:
    print('(%s␣:␣%d)' % (c, ord(c)), end='␣')
# (P : 80) (y : 121) (t : 116) (h : 104) (o : 111) (n : 110)
```

怎样使用 for 语句实现一个程序输出某一给定自然数区间内的所有质数？这个问题比我们之前所解决的问题更加复杂。当问题比较复杂时，在编写程序之前应提出一个设计方案，这样便于对解决问题的策略和步骤进行深入而细致的思考，避免错误。此外，还可以在保证正确性的前提下选择最优解决方案，提高程序的运行效率并降低资源占用。

这个问题的设计方案如下：

1. 列举给定区间内的所有自然数。
   (a) 对于每个自然数 i，判断其是否质数。对于每个从 2 到 $i-1$ 的自然数 j：
      i. 检查 i 是否可以被 j 整除。
   (b) 若存在这样的 j，则 i 非质数。否则 i 为质数，输出 i。

根据质数的定义，这个设计方案是正确的，而且每个步骤都易于实现，但是在运行效率上还有改进的余地。在步骤 1 列举自然数时，只需列出奇数，因为偶数肯定不是质数。在步骤 1(a) 查找 i 的因子 j 时，j 的取值范围的上界可以缩小为 $\lceil\sqrt{i}\rceil$。因为若 $i = j \times k$，则 j 和 k 中至少有一个不大于 $\lceil\sqrt{i}\rceil$。

基于以上改进的设计方案，可以使用嵌套 for 语句写出程序 3.11。第 5 行开始的外循环用 range 类型列举所有奇数。第 7 至第 10 行的内循环检查 i 是否有因子，如果有则将 isPrime 的值设为 False，并使用 break 语句跳出内循环。第 11 行根据 isPrime 的值决定是否输出 i。第 7 行至第 10 行的内循环作为一个整体相对于第 5 行的 for 必须有四个空格的缩进。

Listing 3.11: **for 语句和 break 语句输出 100 到 200 之间的所有质数**

```
1  import math
2  lb = 100; ub = 200
3  if lb % 2 == 0: lb += 1
4  if ub % 2 == 0: ub -= 1
5  for i in range(lb, ub + 1, 2):
6      isPrime = True
7      for j in range(2, math.ceil(math.sqrt(i)) + 1):
8          if i % j == 0:
9              isPrime = False
10             break
11     if isPrime: print(i, end=' ')
12 #   101 103 107 109 113 127 ... 199
```

3.3 while 语句

while 语句包含一个条件表达式和一个语句块。while 语句是一种循环语句，它的语句块可以反复运行。while 语句的运行过程如下：

1. 对条件表达式求值。
2. 若值为 False，则 while 语句运行结束。
3. 若值为 True，则运行语句块，然后跳转到 1。

程序 3.12 用 while 语句实现了和 3.5 相同的功能。

Listing 3.12: **while 语句输出 1 到 10 之间的所有自然数的和**

```
sum = 0; n = 10; i = 1
while i <= n:
    sum += i
    i += 1
print("The sum of 1 to %d is %d" % (n, sum))
```

程序 3.13 用 while 语句实现了和 3.6 相同的功能。

Listing 3.13: **while 语句输出 100 到 120 之间的所有偶数**

```
i = lb = 100; ub = 120
while i <= ub:
    print(i, end=' ')
    i += 2
```

for 语句常用于循环次数已知的情形，而 while 语句也适用于循环次数未知的情形。程序 3.14 用 while 语句实现了辗转相减法求两个正整数的最大公约数。

Listing 3.14: **辗转相减法求两个正整数的最大公约数**

```
a = 156; b = 732
str = 'The greatest common divisor of %d and %d is ' % (a, b)
while a != b:
    if a > b:
        a -= b;
    else:
        b -= a;
print(str + ('%d' % a))
# The greatest common divisor of 156 and 732 is 12
```

3.4 推导式

list、dict 和 set 等容器类型都提供了一种称为推导式 (comprehension) 的紧凑语法，可以通过迭代从已有容器创建新的容器。

程序 3.15 演示了推导式的用法。第 2 行创建一个列表 multiplier_of_3，由集合 nums 中 3 的倍数构成。第 4 行创建一个集合 square_of_odds，由 nums 中的奇数的平方构成。第 8 行基于从列表 s 转换到的集合 set(s) 创建一个字典 sr，sr 中的每个元素由集合中的每个数和其除以 3 得到的余数组成。第 10 行从字典 sr 创建另一个字典 tr，由 sr 中 3 的倍数组成。

Listing 3.15: **推导式的用法**

```
1 nums = {25, 18, 91, 365, 12, 78, 59}
2 multiplier_of_3 = [n for n in nums if n % 3 == 0]
3 print(multiplier_of_3)  # [12, 78, 18]
4 square_of_odds = {n*n for n in nums if n % 2 == 1}
5 print(square_of_odds)   # {133225, 3481, 625, 8281}
6
7 s = [25, 18, 91, 365, 12, 78, 59, 18, 91]
8 sr = {n:n%3 for n in set(s)}
9 print(sr)  # {18: 0, 25: 1, 91: 1, 59: 2, 12: 0, 365: 2, 78: 0}
```

```
tr = {n:r for (n,r) in sr.items() if r==0}
print(tr)  # {18: 0, 12: 0, 78: 0}
```

3.5 实验 3：分支和迭代

实验目的

本实验的目的是掌握分支和迭代的语句。

提交方式

在 Blackboard 提交一个文本文件 (txt 后缀)，文件中记录每道题的源程序和运行结果。

实验内容

1. 考拉兹猜想 (Collatz conjecture)
定义一个从给定正整数 n 构建一个整数序列的过程如下。开始时序列只包含 n。如果序列的最后一个数 m 不为 1 则根据 m 的奇偶性向序列追加一个数。如果 m 是偶数，则追加 $m/2$，否则追加 $3 \times m + 1$。考拉兹猜想认为从任意正整数构建的序列都会以 1 终止。编写程序读取用户输入的正整数 n，然后在 while 循环中输出一个以 1 终止的整数序列。输出的序列显示在一行，相邻的数之间用空格分隔。例如用户输入 17 得到的输出序列是“17 52 26 13 40 20 10 5 16 8 4 2 1”。

2. 字符串加密
编写程序实现基于偏移量的字符串加密。加密的过程是对原字符串中的每个字符对应的 Unicode 值加上一个偏移量，然后将得到的 Unicode 值映射到该字符对应的加密字符。用户输入一个不小于-15 的非零整数和一个由大小写字母或数字组成的字符串，程序生成并输出加密得到的字符串。例如用户输入 10 和字符串“Attack at 1600”得到的加密字符串是“K~~kmu*k~*;@::”。需要思考的问题是：怎样对加密得到的字符序列进行解密？怎样改进这个加密方法 (例如对每个字符设置不同的偏移量)？

3. 推导式转换为 for 语句
将程序 3.15 中的所有推导式转换为 for 语句。

第 4 章　函数和模块

4.1　定义和调用函数

函数是一组语句，可以根据输入参数计算输出结果。把需要多次运行的代码写成函数，可以实现代码的重复利用。以函数作为程序的组成单位使程序更易理解和维护。

函数的定义包括函数头和函数体两部分。例如程序 3.14 可以改写成一个函数 gcd，它接受两个自然数作为输入值 (即形参，formal parameters)，计算其最大公约数并返回。在程序 4.1 中，函数 gcd 的定义包括前 9 行语句。第 1 行是函数头，以关键字 def 开始，之后是空格和函数的名称 (gcd)。函数名称后面是用括号括起的一个或多个形参。如果有多个形参，它们之间用逗号分隔。第 2 行至第 9 行构成函数体，相对函数头需要有四个空格的缩进。函数体由一条或多条语句构成，完成函数的功能。函数头后面通常写一个由三个 (单或双) 引号括起的字符串作为函数的注释。注释的内容包括函数的形参、实现的功能、返回值和设计思路等。第 9 行的 return 语句将变量 a 的值作为运行结果返回。

调用函数的语法是在函数的名称后面加上用括号括起一个或多个实参 (argument)。如果有多个实参，它们之间用逗号分隔。这些实参必须和函数的形参在数量上相同，并且在顺序上一一对应。第 10 行用参数 156 和 732 调用函数 gcd，实参 156 赋值给了函数的形参 a，实参 732 赋值给了函数的形参 b，函数的返回值是 12。第 11 行用参数 1280 和 800 调用函数 gcd，函数的返回值是 160。

Listing 4.1: **定义一个辗转相减法求两个正整数的最大公约数的函数**

```
1 def gcd(a, b):
2     """ 辗转相减法求两个正整数a和b的最大公约数 """
3     while a != b:
4         if a > b:
5             a -= b;
6         else:
7             b -= a;
```

```
    return a

print(gcd(156, 732))  # 12
print(gcd(1280, 800))  # 160
```

函数可以返回多个结果，这些结果之间用逗号分隔，构成一个元组。例如程序 4.2 的第 1 行至第 8 行中定义了一个函数 max_min，它接受两个数作为参数，返回它们的最大值和最小值。

Listing 4.2: **定义一个求最大值和最小值的函数**

```
1 def max_min(a, b):
2     """ 计算a和b的最大值和最小值 """
3     if a > b:
4         return a, b
5     else:
6         return b, a
7
8 print(max_min(156, 34))  # (156, 34)
9 print(max_min(12, 800))  # (800, 12)
```

4.2 局部变量和全局变量

函数的形参和在函数体内定义的变量称为局部变量。局部变量只能在函数体内访问，在函数运行结束时即被销毁。在函数体外定义的变量称为全局变量。全局变量在任何函数中都可以被访问，除非某个函数中定义了同名的局部变量。

例如程序 4.3 的前两行定义的变量 b 和 c 是全局变量。第 3 行函数 f 的形参 a 和第 4 行定义的变量 b 是函数 f 的局部变量。在函数 f 中可以访问第 2 行定义的全局变量 c，但无法访问第 1 行定义的全局变量 b。第 6 行的输出结果表明第 5 行中出现的 b 是第 4 行定义的局部变量。第 7 行的输出结果表明第 4 行是给局部变量 b 赋值，而不是给第 1 行定义的全局变量 b 赋值。

Listing 4.3: **局部变量和全局变量**

```
1 b = 10
2 c = 15
3 def f(a):
4     b = 20
5     return a + b + c
6 print(f(5))  # 40
7 print('b␣=␣%d' % b)  # b = 10
```

如果需要在在函数体中修改某个全局变量，需要用 global 声明它。例如程序 4.4 的第 4 行用 global 声明了全局变量 b。第 8 行的输出结果表明第 5 行是给全局变量 b 赋值。

Listing 4.4: **函数中修改全局变量**

```
1 b = 10
2 c = 15
3 def f(a):
4     global b
5     b = 20
6     return a + b + c
7 print(f(5))  # 40
8 print('b␣=␣%d' % b)  # b = 20
```

4.3 默认值形参和关键字实参

函数头可以给一个或多个形参赋予默认值，这些形参称为默认值形参 (default parameters)。这些默认值形参的后面不能出现普通的形参。

在调用函数的语句中，可以在一个或多个实参的前面写上其对应的形参的名称。这些实参称为关键字实参 (keyword arguments)。此时实参的顺序不必和函数头中的形参的顺序保持一致。

程序 4.5 的第 2 行至第 14 行定义了一个函数 get_primes，它接受两个自然数作为形参，并返回以这两个自然数为下界和上界的区间中的所有质数。这里表示上界的形参 ub 设置了默认值 100，所以第 17 行的调用 get_primes(80) 等同于 get_primes(80, 100)。第 18 行的调用 get_primes(ub=150, lb=136) 使用了两个关键字实参，即和实参 150 对应的形参是 ub 并且和实参 136 对应的形参是 lb，这里关键字实参的顺序和函数头中的形参顺序并不一致。

这个函数的返回值有多个而且数量未知，对于类似的情形可以把所有需要返回的结果存储在一个容器 (例如列表) 中，最后返回整个容器。第 4 行定义了一个空列表 primes。第 13 行确认 isPrime 为 True 时将 i 追加到 primes 中。第 14 行返回列表 primes。

Listing 4.5: **定义一个求解给定取值范围内的所有质数的函数**

```
1 import math
2 def get_primes(lb, ub=100):
3     """ 求解给定取值范围[lb, ub]内的所有质数 """
4     primes = []
5     if lb % 2 == 0: lb += 1
6     if ub % 2 == 0: ub -= 1
7     for i in range(lb, ub + 1, 2):
```

```
        isPrime = True
        for j in range(2, math.ceil(math.sqrt(i)) + 1):
            if i % j == 0:
                isPrime = False
                break
        if isPrime: primes.append(i)
    return primes

print(get_primes(40, 50))  # [41, 43, 47]
print(get_primes(120, 140))  # [127, 131, 137, 139]
print(get_primes(80))  # [83, 89, 97]
print(get_primes(ub=150, lb=136))  # [137, 139, 149]
```

4.4 可变数量的实参

函数可以接受未知数量的位置实参和关键字实参。程序 4.6 的第 1 行至第 4 行定义了一个函数 fun。形参 args 是一个元组，可接受未知数量的位置实参。形参 kwargs 是一个字典，可接受未知数量的关键字实参。

Listing 4.6: **未知数量的位置实参和关键字实参**

```
1 def fun(*args, **kwargs):
2     print(type(args), type(kwargs))
3     print('The␣positional␣arguments␣are', args)
4     print('The␣keyword␣arguments␣are', kwargs)
5
6 fun(1, 2.3, 'a', True, u=6, x='Python', f=3.1415)
7 # <class 'tuple'> <class 'dict'>
8 # The positional arguments are (1, 2.3, 'a', True)
9 # The keyword arguments are {'u': 6, 'x': 'Python', 'f': 3.1415}
```

4.5 函数式编程

Python 语言支持函数式编程 (functional programming) 范式的基本方式是函数具有和其他类型 (如 int、float 等) 同样的性质: 被赋值给变量；作为实参传给被调用函数的形参；作为函数的返回值。

4.5.1 函数作为实参

内置函数 sorted 可以对一个可遍历对象 (iterable) 中的元素进行排序，排序的结果存储在一个新创建的列表中。关键字实参 key 指定一个函数，它从每个元素生成用于排序的比较值。关键字实参 reverse 的默认值为 False，若设为 True 则表示从大到小的次序排序。

程序 4.7 演示了用函数作为实参调用 sorted 函数。In[1] 行定义了由字符串构成的列表。In[2] 行对其排序，Out[2] 行显示了输出结果，默认的排序方式是按照两个字符串的字符序列的 Unicode 编码值从小到大排序，即首先比较两个字符串的第一个字符的 Unicode 编码值，若不等则已确定顺序，若相等则再比较第二个字符，以此类推。In[3] 行在调用 sorted 函数时设置了关键字实参 key 为求字符串长度的内置函数 len，Out[3] 行显示了输出结果，即按照字符串的长度从小到大排序。In[4] 行和 In[3] 行的区别在于设置了关键字实参 reverse 为 True，Out[4] 行显示了输出结果，即按照字符串的长度从大到小排序。In[5] 行定义了一个函数 m1，它返回一个字符串中的所有字符的 Unicode 编码值的最小值。In[6] 行在调用 sorted 函数时设置了关键字实参 key 为 m1，Out[6] 行显示了输出结果，即按照字符串的所有字符的 Unicode 编码值的最小值从小到大排序。In[7] 行定义了一个函数 m2，它返回一个元组，由一个字符串中的所有字符的 Unicode 编码值的最小值 (以下简称为最小编码值) 和字符串的长度组成。元组在排序时看成组成元组的元素的序列，即先比较两个元组的第一个元素，若不等则已确定顺序，若相等则再比较第一个元素，以此类推。In[8] 行在调用 sorted 函数时设置了关键字实参 key 为 m2，Out[8] 行显示了输出结果，即先按照最小编码值从小到大排序，若两个字符串具有相同的最小编码值，则按照长度从小到大排序。

Listing 4.7: **用函数作为实参调用 sorted 函数**

```
1  In[1]: animals = ["elephant", "tiger", "rabbit", "goat", "dog",
2                    "penguin"]
3  In[2]: sorted(animals)
4  Out[2]: ['dog', 'elephant', 'goat', 'penguin', 'rabbit', 'tiger']
5  In[3]: sorted(animals, key=len)
6  Out[3]: ['dog', 'goat', 'tiger', 'rabbit', 'penguin', 'elephant']
7  In[4]: sorted(animals, key=len, reverse=True)
8  Out[4]: ['elephant', 'penguin', 'rabbit', 'tiger', 'goat', 'dog']
9  In[5]: def m1(s): return ord(min(s))
10 In[6]: sorted(animals, key=m1)
11 Out[6]: ['elephant', 'rabbit', 'goat', 'dog', 'tiger', 'penguin']
12 In[7]: def m2(s): return ord(min(s)), len(s)
13 In[8]: sorted(animals, key=m2)
14 Out[8]: ['goat', 'rabbit', 'elephant', 'dog', 'tiger', 'penguin']
```

如果一个函数在定义以后只使用一次，并且函数体可以写成一个表达式，则可以使用 Lambda 函数语法将其定义成一个匿名函数：g = lambda 形参列表: 函数体表达式

例如程序 4.8 中定义的函数 map 将函数 f 作用于列表 s 中的每个元素，用函数的返回值替代原来的元素，最后返回 s。第 6 行调用函数 map 时提供的第一个实参是一个 Lambda 函数，它对于形参 x 返回 x+1。第 7 行的 Lambda 函数对于形参 x 返回 x*x-1。

Listing 4.8: **Lambda 函数**

```
1 def map_fs(f, s):
2     for i in range(len(s)): s[i] = f(s[i])
3     return s
4
5 a = [1, 3, 5, 7, 9]
6 print(map_fs(lambda x: x+1, a))  # [2, 4, 6, 8, 10]
7 print(map_fs(lambda x: x*x-1, a))  # [3, 15, 35, 63, 99]
```

程序 4.8 中定义的函数 map_fs 是标准库中的内置函数 map 的简化。内置函数 map 可将它的第一个参数 (一个函数) 作用于其余参数 (一个或多个可遍历对象) 中的每个元素，并返回一个可遍历对象，它可以生成一个列表。内置函数 filter 类似一个过滤器，它的第一个参数 (一个函数) 作用于其余参数 (一个或多个可遍历对象) 中的每个元素时返回一个 bool 类型值。若值为 True，则对应元素被保留在输出结果中，否则被舍弃。程序 4.9 演示了用函数作为实参调用内置函数 map 和 filter。In[2] 行定义了一个反转字符串的函数 reverse。In[3] 行使用 reverse 调用 map 函数，反转了 In[1] 行定义的列表 animals 中的每个字符串，结果显示在 Out[3] 行。In[4] 行定义了程序 4.7 中定义的函数 m2。In[5] 行使用 m2 调用 map 函数，对列表 animals 中的每个字符串输出一个元组，结果显示在 Out[5] 行。In[6] 行定义了一个函数 f，它返回三个形参的和。In[7] 行使用 f 调用 map 函数，对三个列表中对应位置的元素分别求和 (1+10+100=111,2+20+200=222,3+30+300=333)，结果显示在 Out[7] 行。In[9] 行定义了一个函数 r3，它判断形参是不是 3 的倍数。In[10] 行使用 r3 调用 filter 函数，提取 In[8] 行定义的集合 nums 包含的 3 的倍数，结果显示在 Out[10] 行。

Listing 4.9: **用函数作为实参调用内置函数 map 和 filter**

```
1  In[1]: animals = ["elephant", "tiger", "rabbit", "goat", "dog",
2                    "penguin"]
3  In[2]: def reverse(s): return s[::-1]
4  In[3]: list(map(reverse, animals))
5  Out[3]: ['tnahpele', 'regit', 'tibbar', 'taog', 'god', 'niugnep']
6  In[4]: def m2(s): return ord(min(s)), len(s)
7  In[5]: list(map(m2, animals))
8  Out[5]: [(97, 8), (101, 5), (97, 6), (97, 4), (100, 3), (101, 7)]
9  In[6]: def f(a, b, c): return a + b + c
10 In[7]: list(map(f, [1, 2, 3], [10, 20, 30], [100, 200, 300]))
11 Out[7]: [111, 222, 333]
```

```
In[8]: nums = {25, 18, 91, 365, 12, 78, 59}
In[9]: def r3(n): return n % 3 == 0
In[10]: list(filter(r3, nums))
Out[10]: [12, 78, 18]
```

4.5.2 函数作为返回值

程序 4.10 定义了一个函数 key_fun，其中定义了两个用于字符串排序的函数 m1 和 m2。列表 ms 存储了 None、len 和这些函数。key_fun 以实参为索引值返回 ms 中的对应函数，即该函数的返回值是一个函数。第 10 行至第 11 行的循环依次使用这些函数对字符串进行排序，其中 None 表示默认的排序方式。输出结果显示在 4.11。

Listing 4.10: **用函数作为作为返回值进行字符串排序**

```
1  def key_fun(n):
2      def m1(s): return ord(min(s))
3      def m2(s): return ord(min(s)), len(s)
4
5      ms = [None, len, m1, m2]
6      return ms[n]
7
8  animals = ["elephant", "tiger", "rabbit", "goat", "dog", "penguin"]
9  for i in range(4):
10     print(sorted(animals, key=key_fun(i)))
```

Listing 4.11: **程序 4.10 的输出结果**

```
1  ['dog', 'elephant', 'goat', 'penguin', 'rabbit', 'tiger']
2  ['dog', 'goat', 'tiger', 'rabbit', 'penguin', 'elephant']
3  ['elephant', 'rabbit', 'goat', 'dog', 'tiger', 'penguin']
4  ['goat', 'rabbit', 'elephant', 'dog', 'tiger', 'penguin']
```

4.6 递归

递归就是一个函数调用自己。当要求解的问题满足以下三个条件时，递归是有效的解决方法。

1. 原问题可以分解为一个或多个结构类似但规模更小的子问题。
2. 当子问题的规模足够小时可以直接求解，称为递归的终止条件；否则可以继续对子问题递归求解。
3. 原问题的解可由子问题的解合并而成。

用递归方法解决问题的过程是基于对问题的分析提出递归公式。以下举例说明。

4.6.1 阶乘

阶乘的定义本身就是一个递归公式：

$$n! = \begin{cases} 1 & \text{if } n = 1 \\ n * (n-1)! & \text{if } n > 1 \end{cases}$$

程序 4.12 中的 factorial 函数计算阶乘。

Listing 4.12: 计算阶乘的递归函数

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(10))  # 3628800
```

4.6.2 最大公约数

设 a 和 b 表示两个正整数。若 $a > b$，则易证 a 和 b 的公约数集合等于 $a - b$ 和 b 的公约数集合，因此 a 和 b 的最大公约数等于 $a - b$ 和 b 的最大公约数。若 $a = b$，则 a 和 b 的最大公约数等于 a。由此可总结出递归公式如下：

$$\gcd(a, b) = \begin{cases} a & \text{if } a = b \\ \gcd(a-b, b) & \text{if } a > b \\ \gcd(a, b-a) & \text{if } a < b \end{cases}$$

程序 4.13 的 gcd 函数求解两个正整数的最大公约数。递归函数都可以转换成与其等价的迭代形式，例如这个 gcd 函数对应的迭代形式是程序 4.1 中的 gcd 函数。

Listing 4.13: 计算最大公约数的递归函数

```
def gcd(a, b):
    if a == b:
        return a
    elif a > b:
        return gcd(a-b, b)
    else:
        return gcd(a, b-a)
```

```
8
9 print(gcd(156, 732))  # 12
```

4.6.3 字符串反转

字符串反转就是将原字符串中的字符的先后次序反转，例如“ABCDE”反转以后得到“EDCBA”。问题的分析过程如下。原字符串“ABCDE”可看成是两个字符串“ABCD”和“E”的连接。反转以后的字符串“EDCBA”可看成是两个字符串“E”和“DCBA”的连接。“DCBA”是“ABCD”的反转，是原问题的子问题。“E”是“E”的反转，也是原问题的子问题。递归的终止条件是: 由单个字符构成的字符串的反转就是原字符串。由此可总结出递归公式如下：

$$\text{reverse}(s) = \begin{cases} s & \text{if len}(s) = 1 \\ s[-1] + \text{reverse}(s[:-1]) & \text{if len}(s) > 1 \end{cases}$$

程序 4.14 的 reverse 函数反转一个字符串。

Listing 4.14: **计算字符串反转的递归函数**

```
1 def reverse(s):
2     if len(s) == 1:
3         return s
4     else:
5         return s[-1] + reverse(s[:-1])
6
7 print(reverse("ABCDE"))  # EDCBA
```

4.6.4 快速排序

快速排序是一种著名的排序算法，以下用一个实例描述其求解过程。要排序的原始数据集是列表 [3, 6, 2, 9, 7, 3, 1, 8]。以第一个元素 3 为基准对其进行调整，把比 3 小的元素移动到 3 的左边，把比 3 大的元素移动到 3 的右边。调整的结果为 [2, 1, 3, 3, 6, 9, 7, 8]，可看成是三个列表的连接: [2, 1], [3, 3], 和 [6, 9, 7, 8]。这三个列表分别由小于 3 的元素、等于 3 的元素和大于 3 的元素组成。排序完成的结果是 [1, 2, 3, 3, 6, 7, 8, 9]，也可看成是三个列表的连接: [1, 2], [3, 3], 和 [6, 7, 8, 9]。对 [2, 1] 进行排序可得 [1, 2]，这是原问题的一个子问题。对 [6, 9, 7, 8] 进行排序可得 [6, 7, 8, 9]，这也是原问题的一个子问题。由此可总结出递归公式如下：

$$\text{qsort}(s) = \begin{cases} s & \text{if len}(s) \leq 1 \\ \text{qsort}(\{i \in s | i < s[0]\}) + \{i \in s | i = s[0]\} + \text{qsort}(\{i \in s | i > s[0]\}) & \text{if len}(s) > 1 \end{cases}$$

程序 4.15 的 qsort 函数实现了一个易于理解的快速排序算法，并非这个算法的高效实现。第 4 行至第 10 行的循环将列表 s 中比 s[0] 小的元素添加到列表 s_less 中，将 s 中比 s[0] 大的元素添加到列表 s_greater 中，将 s 中等于 s[0] 的元素添加到列表 s_equal 中。第 11 行分别对 s_less 和 s_greater 执行递归调用，并将其结果和 s_equal 连接在一起得到排序的最终结果。

Listing 4.15: 实现快速排序的递归函数

```
def qsort(s):
    if len(s) <= 1: return s
    s_less = []; s_greater = []; s_equal = []
    for k in s:
        if k < s[0]:
            s_less.append(k)
        elif k > s[0]:
            s_greater.append(k)
        else:
            s_equal.append(k)
    return qsort(s_less) + s_equal + qsort(s_greater)

print(qsort([3, 6, 2, 9, 7, 3, 1, 8])) # [1, 2, 3, 3, 6, 7, 8, 9]
```

4.6.5 Fibonacci 数列

Fibonacci 数列由以下递归公式定义：

$$F(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

程序 4.16 的第 1 行至第 4 行定义了一个由递归公式得到的函数 F。该函数在运行时会进行一些重复计算。例如计算 F(6)=F(5)+F(4) 时需要计算 F(4)，而计算 F(5)=F(4)+F(3) 时又需要计算 F(4)。

为避免类似的重复计算，第 6 行至第 13 行定义的函数 F_memoization 用一个列表 v 记录已经计算的函数值，列表中用-1 表示尚未计算的函数值。第 8 行至第 12 行定义的递归函数 F 在计算 F(n) 时先检查该列表，若其中已经存储了 F(n) 的值 v[n] 则直接返回 v[n]，否则再进行递归调用。

从 Fibonacci 数列的递归公式可以推导出以下迭代公式：

$$F[n] = \begin{cases} 1 & \text{if } n \leq 1 \\ F[n-1] + F[n-2] & \text{if } n > 1 \end{cases}$$

公式中用一维表格 F 记录 Fibonacci 数列。第 15 行至第 19 行定义的函数 F_iter1 根据迭代公式用 for 循环计算 F(n)。第 7 行初始化的列表 v 表示公式中的一维表格 F。第 21 行至第 27 行定义的函数 F_iter2 也根据迭代公式用 for 循环计算 F(n)，但不用列表记录中间结果。

Listing 4.16: 计算 Fibonacci 数列

```python
def F(n):
    if n <= 1:
        return 1
    return F(n-1) + F(n-2)

def F_memoization(n):
    v = [-1] * (n+1); v[0] = 1; v[1] = 1
    def F(n):
        if v[n] > -1:
            return v[n]
        v[n] = F(n-1) + F(n-2)
        return v[n]
    return F(n)

def F_iter1(n):
    v = [-1] * (n + 1); v[0] = 1; v[1] = 1
    for i in range(2, n+1):
        v[i] = v[i-1] + v[i-2]
    return v[n]

def F_iter2(n):
    if n <= 1:
        return 1
    a = 1; b = 1
    for i in range(2, n+1):
        c = a + b; a = b; b = c
    return c

n = 19
print(F(n), F_memoization(n), F_iter1(n), F_iter2(n))
# 6765 6765 6765 6765
```

4.6.6 0-1 背包问题

给定 n 件物品，它们的编号是 $0, 1, ..., n-1$。每件物品都有其重量 w_i 和价值 $v_i(0 \le i \le n-1)$。0-1 背包问题从这 n 件物品中选取若干件放入一个背包中，使得在这些物品的总重量不超过背包的容量的前提下最大化它们的总价值。这里的 0-1 指每件物品是不可分割的。

设 A 表示某种选取方式所选取的物品的集合，$\text{weight}(A)$ 表示它们的总重量，$\text{value}(A)$ 表示它们的总价值。当背包的当前容量为 c 并且选取范围是前 k 件物品时，符合不超重条件的 A 构成的集合定义为 $F(c,k)=\{A\subseteq\{0,1,...,k-1\}|\text{weight}(A)\le c\}$，最优解定义为 $\text{items}(c,k)=\underset{A\in F(c,k)}{\arg\max}\ \text{value}(A)$，最优解的总价值定义为 $\text{ks}(k,c)=\text{value}(\text{items}(c,k))$。对于编号为 $0,...,k-1$ 的前 k 件物品做出的选择可分为两部分：首先对编号为 $k-1$ 的物品做出选择，再对前 $k-1$ 件物品做出选择。这两个部分做出的决策是互相独立的，因此可以分两步进行。以下列出了设计方案。

1. 在第一步，第 k 件物品的重量 w_{k-1} 是否大于背包的当前容量 c?

   (a) 是。此时无法选取第 k 件物品。对于任意的 $A\in F(c,k)$，则 $A\in F(c,k-1)$。对于任意的 $B\in F(c,k-1)$，则 $B\in F(c,k)$。因此 $F(c,k)=F(c,k-1)$。$F(c,k)$ 的最优解就是 $F(c,k-1)$ 的最优解。原问题转换为在第二步的子问题：当背包的当前容量为 c 时对前 $k-1$ 件物品进行选取获得最大总价值，即求解子问题 $\text{ks}(k-1,c)$。

   (b) 否。可以选取第 k 件物品也可以不选取第 k 件物品，需要从两种决策中选择总价值最大者。

      i. 若选取了第 k 件物品，则它占据了背包的一部分容量 w_{k-1} 并且为总价值增加了 v_{k-1}。对于任意的 $A\in F(c,k)$，若 $\{k-1\}\subseteq A$ 则 $A-\{k-1\}\in F(c-w_{k-1},k-1)$。对于任意的 $B\in F(c,k-1)$，若 $\text{weight}(B)+w_{k-1}\le c$ 则 $B\cup\{k-1\}\in F(c,k)$。因此，若 $\{k-1\}$ 是 $F(c,k)$ 的最优解的子集，则 $F(c,k)$ 的最优解等于 $F(c-w_{k-1},k-1)$ 的最优解和 $\{k-1\}$ 的并集。原问题转换为在第二步的子问题：当背包的当前容量为 $c-w_{k-1}$ 时对前 $k-1$ 件物品进行选取, 并获得最大总价值，即求解子问题 $\text{ks}(k-1,c-w_{k-1})$。

      ii. 若不选取第 k 件物品，在第二步的问题是当背包的当前容量为 c 时对前 $k-1$ 件物品进行选择并获得最大总价值，即求解子问题 $\text{ks}(k-1,c)$。

递归的终止条件是: 当 $k=0$(无物品可选) 或 $c=0$(背包的当前容量为 0) 时，$\text{ks}(k,c)$ 的值为 0。由此可总结出递归公式如下：

$$\text{ks}(k,c)=\begin{cases}0 & \text{if } k=0 \text{ or } c=0\\ \text{ks}(k-1,c) & \text{if } w_{k-1}>c\\ \max(\text{ks}(k-1,c),\ v_{k-1}+\text{ks}(k-1,c-w_{k-1})) & \text{if } w_{k-1}\le c\end{cases}$$

用二维表格 ks 记录 $\text{ks}(k,c)$，从递归公式可以推导出以下迭代公式：

$$\text{ks}[k,c]=\begin{cases}0 & \text{if } k=0 \text{ or } c=0\\ \text{ks}[k-1,c] & \text{if } w_{k-1}>c\\ \max(\text{ks}[k-1,c],\ v_{k-1}+\text{ks}[k-1,c-w_{k-1}]) & \text{if } w_{k-1}\le c\end{cases}$$

程序 4.17 的 knapsack 函数使用递归方法求解 0-1 背包问题。它的 return 语句先调用定义在其函数体内的递归函数 ks 获得最优解的总价值，然后调用函数 get_items 获取最优解。

knapsack_iter 函数根据迭代公式通过双重循环求解 0-1 背包问题。

Listing 4.17: **求解 0-1 背包问题**

```
def knapsack(weights, values, c0):
    # 使用递归方法求解
    # 形参weights是存储了每件物品的重量的列表
    # 形参values是存储了每件物品的价值的列表
    # 形参c0是背包的容量
    n = len(weights)
    selects = [[False]*(c0+1) for i in range(n+1)]
    # selects记录在子问题ks(k,c)中是否选取了第k件物品
    def ks(k, c): # 递归函数ks实现了上述递归公式
        if k == 0 or c == 0: return 0
        exclude_k = ks(k-1, c)
        if weights[k-1] > c: return exclude_k
        select_k = values[k-1] + ks(k - 1, c - weights[k-1])
        if exclude_k > select_k:
            return exclude_k
        else:
            selects[k][c] = True
            return select_k
    return ks(n, c0), get_items(n, c0, weights, selects)

def get_items(k, c, weights, selects):
    # 根据selects获取最优解
    items = set()
    while k > 0 and c > 0:
        if selects[k][c]:
            items.add(k-1); c -= weights[k-1]
        k -= 1
    return items

def knapsack_iter(weights, values, c0):
    # 使用迭代方法求解
    n = len(weights)
    ks = [[0]*(c0+1) for i in range(n+1)] # ks表示公式中的二维表格ks
    selects = [[False]*(c0+1) for i in range(n+1)]
    # selects记录在子问题ks(k,c)中是否选取了第k件物品
```

```
    for k in range(1, n+1):
        for c in range(1, c0+1):
            exclude_k = ks[k-1][c]
            if weights[k-1] > c:
                ks[k][c] = ks[k-1][c]
                continue
            select_k = values[k-1] + ks[k - 1][c - weights[k-1]]
            if exclude_k > select_k:
                ks[k][c] = exclude_k
            else:
                selects[k][c] = True
                ks[k][c] = select_k
    import pprint; pprint.pprint(ks)
    return ks[n][c0], get_items(n, c0, weights, selects)

weights = [2,1,3,2]; values = [12,10,20,15]; c0=5
# 本例中背包的容量是5, weights和values分别存储了每件物品的重量和价值
print(knapsack(weights, values, c0))  # (37, {0, 1, 3})
# 最优解选取了编号为0、1和3的物品, 它们的总价值是37。
print(knapsack_iter(weights, values, c0)) # 输出结果同上
'''
[[0, 0, 0, 0, 0, 0],
 [0, 0, 12, 12, 12, 12],
 [0, 10, 12, 22, 22, 22],
 [0, 10, 12, 22, 30, 32],
 [0, 10, 15, 25, 30, 37]]
'''
```

求解 Fibonacci 数列和 0-1 背包问题既可以使用递归方法，也可以使用迭代方法。一般而言，递归方法的优点是问题分析和程序设计更加容易，缺点有以下三个方面：递归在运行时需要进行多次函数调用从而导致一些额外的时间和空间开销；递归深度受到内存容量和运行时系统的限制；可能存在重复计算。

4.6.7 矩阵链乘积问题

给定 n 个矩阵 $A_1, A_2, ..., A_n$ 构成的序列，需要计算它们的乘积 $A_1A_2...A_n$。计算乘积的次序可以有多种，通过添加括号可以确定一种具体的次序。例如当 $n = 4$ 时，$((A_1A_2)(A_3A_4))$ 和 $((A_1(A_2A_3))A_4)$ 是两种可行的次序。对于 $1 \le i \le n$，设矩阵 A_i 的行数和列数分别为 r_i 和 c_i，则根据矩阵乘法的规则，对于 $1 \le i \le n-1$ 成立 $c_i = r_{i+1}$，即矩阵 A_i 的列数是 r_{i+1}。两

个大小分别为 m 行 n 列和 n 行 p 列的矩阵相乘的计算代价定义为需要进行的乘法次数 mnp。矩阵链乘积的计算代价定义为所有的矩阵相乘的计算代价的总和。由于矩阵乘法满足结合律，采用不同的次序计算得到的结果一定是相同的，但计算代价不一定相同。例如，设 4 个矩阵的大小分别是 35 行 15 列、15 行 5 列、5 行 10 列和 10 行 20 列。第一种次序的计算代价是：$35 \times 15 \times 5 + 5 \times 10 \times 20 + 35 \times 5 \times 20 = 7125$。第二种次序的计算代价是：$15 \times 5 \times 10 + 35 \times 15 \times 10 + 35 \times 10 \times 20 = 13000$。

矩阵链乘积 (Matrix-chain multiplication) 问题 [CL2009] 求解一个最小化计算代价的次序。当 $i < j$ 时，计算矩阵链乘积 $A_iA_{i+1}...A_j$ 的最后一次矩阵相乘可表示为 $(A_i...A_k)(A_{k+1}..A_j)$，其中整数 k 满足 $i \le k \le j-1$。k 的值有多种选择，每个 k 值导致矩阵链乘积 $A_iA_{i+1}...A_j$ 分解为两个互相独立的子问题 $A_i...A_k$ 和 $A_{k+1}..A_j$。这两个子问题包含的下标集合分别是 $\{i, ...k\}$ 和 $\{k+1, ...j\}$，它们的交集是空集。因此，在求解第一个子问题时做出的决策 (即矩阵相乘的次序) 不会影响第二个子问题，反之亦然。

对于每个 k 值，矩阵链乘积 $A_iA_{i+1}...A_j$ 的计算代价 $\mathrm{Cost}(i,j)$ 是以下三部分之和：第一个子问题的计算代价 $\mathrm{Cost}(i,k)$；第二个子问题的计算代价 $\mathrm{Cost}(k+1,j)$；两个子问题的计算结果进行相乘的计算代价 $r_ir_{k+1}r_{j+1}$。若要求解 $A_iA_{i+1}...A_j$ 的最小计算代价，对于每个选定的 k 值必须分别求解两个子问题的最小计算代价。由于这三部分都依赖于 k 的值，$\mathrm{Cost}(i,j)$ 也依赖于 k 的值。应选取最小化 $\mathrm{Cost}(i,j)$ 的 k 值。用 $\mathrm{mcm}(i,j)$ 表示乘积 $A_iA_{i+1}...A_j$ 的最小计算代价，它满足如下递归公式：

$$\mathrm{mcm}(i,j) = \begin{cases} 0 & \text{if } i = j \\ \arg\min\limits_{i \le k \le j-1} \ \mathrm{mcm}(i,k) + \mathrm{mcm}(k+1,j) + r_ir_{k+1}r_{j+1} & \text{if } i < j \end{cases}$$

用二维表格 mcm_value 记录乘积 $A_iA_{i+1}...A_j$ 的最小计算代价，从递归公式可以推导出以下迭代公式：

$$\mathrm{mcm_value}[i,j] = \begin{cases} 0 & \text{if } i = j \\ \arg\min\limits_{i \le k \le j-1} \ \mathrm{mcm_value}[i,k] + \mathrm{mcm_value}[k+1,j] + r_ir_{k+1}r_{j+1} & \text{if } i < j \end{cases}$$

程序 4.18 定义的函数 matrix_chain 实现了矩阵链乘积问题的递归求解。它的 return 语句先调用定义在其函数体内的递归函数 mcm 获得最优解的计算代价，然后调用函数 get_mcm_str 获取最优解的计算次序。函数 matrix_chain_iter 根据迭代公式通过双重循环求解矩阵链乘积问题。为了计算矩阵链长度为 $j-i+1$ 的 mcm_value$[i,j]$，需要先计算矩阵链长度为 $k-i+1$ 的 mcm_value$[i,k]$ 和矩阵链长度为 $j-k$ 的 mcm_value$[i,k]$。由于 $k-i+1 < j-i+1$ 并且 $j-k < j-i+1$，应按照矩阵链长度递增的次序计算 mcm_value。

Listing 4.18: **求解矩阵链乘积问题**

```
def matrix_chain(r):
    # 使用递归方法求解
    # 形参r是存储了公式中的r_i值(1 ≤ i ≤ n+1)的列表
```

```
    n = len(r) - 2 # 矩阵的下标从1开始，n表示矩阵的个数
    mcm_value = [[0]*(n+1) for i in range(n+1)]
    # mcm_value存储公式中的mcm(i,j)(1 ≤ i ≤ j ≤ n)
    k_value = [[0]*(n+1) for i in range(n+1)]
    # k_value存储公式中最小化Cost(i,j)的k值
    def mcm(i, j):
        if i == j: return 0
        if mcm_value[i][j] > 0:
            # 若条件成立，则已经调用过mcm(i, j)
            return mcm_value[i][j]
        min_cost = mcm(i, j-1) + r[i] * r[j] * r[j+1]
        # min_cost记录Cost(i,j)的最小值，初始化为k = j − 1时的Cost(i,j)
        k_min_cost = j-1
        for k in range(i, j-1): # 在循环中计算min_cost
            cost = mcm(i, k) + mcm(k+1, j) + \
                   r[i] * r[k+1] * r[j+1]
            if cost < min_cost:
                min_cost = cost
                k_min_cost = k
        mcm_value[i][j] = min_cost # 存储min_cost以避免重复计算
        k_value[i][j] = k_min_cost
        return min_cost
    return mcm(1, n), get_mcm_str(n, k_value)

def get_mcm_str(n, k_value):
    # 获取最优解的计算次序
    def mcm_str(i, j):
        if i == j: return 'A%d' % i
        k = k_value[i][j]
        return '(%s%s)' % (mcm_str(i, k), mcm_str(k+1, j))
    return mcm_str(1, n)

def matrix_chain_iter(r):
    # 使用迭代方法求解
    n = len(r) - 2
    mcm_value = [[0]*(n+1) for i in range(n+1)]
    k_value = [[0]*(n+1) for i in range(n+1)]
    for i in range(1, n+1):
        mcm_value[i][i] = 0
```

```
    for l in range(2, n+1): # l表示矩阵链长度
        for i in range(1, n-l+2): # i表示矩阵链的起始下标
            j = i + l - 1 # j表示矩阵链的终止下标
            min_cost = mcm_value[i][j-1] + r[i] * r[j] * r[j+1]
            k_min_cost = j-1
            for k in range(i, j-1):
                cost = mcm_value[i][k] + mcm_value[k+1][j] + \
                       r[i] * r[k+1] * r[j+1]
                if cost < min_cost:
                    min_cost = cost
                    k_min_cost = k
            mcm_value[i][j] = min_cost
            k_value[i][j] = k_min_cost
    return mcm_value[1][n], get_mcm_str(n, k_value)

rs = [0, 30, 35, 15, 5, 10, 20, 25]
# rs存储了由6个矩阵组成的矩阵链的r_i值(1 ≤ i ≤ 7)
print(matrix_chain(rs))  # (15125, '((A1(A2A3))((A4A5)A6))')
# ((A1(A2A3))((A4A5)A6))实现了最小计算代价15125
print(matrix_chain_iter(rs)) # 输出结果同上
```

4.7 创建和使用模块

模块是一个包含了若干函数和语句的文件，文件名是模块的名称加上“.py”后缀。一个模块实现了某类功能，是规模较大程序的组成单位，易于重复利用代码。每个模块都有一个全局变量__name__。模块的使用方式有两种。

1. 模块作为一个独立的程序运行，此时变量__name__的值为'__main__'。
2. 被其他程序导入以后调用其中的函数，此时变量__name__的值为模块的名称。

以 1.3 节的日历问题和 (输出给定年份和月份的日历) 为例说明创建和使用模块的方法。程序 4.19 列出了根据 1.3 节提供的设计方案所编写的模块 month_calendar.py。开发完成的软件难免会有错误。在软件交付使用前应通过充分测试尽可能查找和改正错误。以“test_____”为名称前缀的测试函数使用已知正确答案的数据测试程序中的一些关键函数。

Listing 4.19: **输出给定年份和月份的日历的模块**

```
"""
Module for printing the monthly calendar for the year and
the month specified by the user.
```

```
4
5  For example, given year 2022 and month 9, the module prints
6  the monthly calendar of September 2022.
7
8  >>> run month_calendar.py --year 2022 --month 9
9  2022  9
10 ----------------------------
11 Sun Mon Tue Wed Thu Fri Sat
12                 1   2   3
13 4   5   6   7   8   9   10
14 11  12  13  14  15  16  17
15 18  19  20  21  22  23  24
16 25  26  27  28  29  30
17 """
18 # 第1行至第17行是模块的文档，解释了模块的功能和用法
19
20 import sys, math
21
22 def is_leap(year):
23     # 若year是闰年返回True，否则返回False
24     return (year % 4 == 0 and year % 100 != 0) or year % 400 == 0
25
26 def test____is_leap():
27     # 用已知正确答案的数据测试is_leap函数，若计算结果不正确则报错
28     d = {1900:False, 2000:True, 2020:True, 2022:False}
29     # 字典d的每个元素的键是一个年份，对应的值表示该年份是不是闰年。
30     for y in d.keys():  # 遍历d中每个元素的键，即一个年份
31         if is_leap(y) != d[y]:  # 判断is_leap函数的计算结果是否正确
32             print("test␣failed:␣is_leap(%d)␣!=␣%s" % (y, d[y]))
33
34 def get_0101_in_week(year):
35     # 计算year年的元旦是星期几。返回0表示星期日，
36     # 返回1至6分别表示星期一至星期六。
37     return (year + math.floor((year - 1) / 4) -
38             math.floor((year - 1) / 100) +
39             math.floor((year - 1) / 400)) % 7
40
41 def test____get_0101_in_week():
42     # 用已知正确答案的数据测试get_0101_in_week函数，
```

```
    # 若计算结果不正确则报错。
    d = {2008:2, 2014:3, 2021:5, 2022:6}
    # 字典d的每个元素的键是一个年份，对应的值表示该年的元旦是星期几。
    for y in d.keys():
        if d[y] != get_0101_in_week(y):
            print("test␣failed:␣get_0101_in_week(%d)␣!=␣%s"
                  % (y, d[y]))

month_days = {1:31, 2:28, 3:31, 4:30, 5:31, 6:30,
              7:31, 8:31, 9:30, 10:31, 11:30, 12:31}
def get_num_days_in_month(year, month):
    # 返回year年month月所包含的天数，需要对闰年的二月单独处理。
    n = month_days[month]
    if month == 2 and is_leap(year):
        return n + 1
    return n

def get_num_days_from_0101_to_m01(year, month):
    # 计算从year年的元旦到该年的month月的第一天之间共经历了多少天。
    n = 0
    for i in range(1, month):
        n += get_num_days_in_month(year, i)
    return n

def get_m01_in_week(year, month):
    # 计算year年的month月的第一天是星期几。
    n1 = get_0101_in_week(year)
    n2 = get_num_days_from_0101_to_m01(year, month)
    n = (n1 + n2) % 7
    return n

def test____get_m01_in_week():
    # 用已知正确答案的数据测试get_m01_in_week函数，
    # 若计算结果不正确则报错。
    d = {(2022, 6):3, (2019, 10):2, (2016, 5):0, (2011, 7):5}
    # 字典d的每个元素的键是由一个年份和一个月份组成的元组，
    # 对应的值表示该年该月的第一天是星期几。
    for y in d.keys():
        if d[y] != get_m01_in_week(y[0], y[1]):
```

```
            print("test␣failed:␣get_m01_in_week(%s)␣!=␣%s"
                  % (y, d[y]))

def print_header(year, month):
    # 输出year年month月的日历的标题。
    print("%d␣␣%d␣" % (year, month))
    print("---------------------------")
    print("Sun␣Mon␣Tue␣Wed␣Thu␣Fri␣Sat")

def print_body(year, month):
    # 输出year年month月的日历的主体。
    n = get_m01_in_week(year, month)
    print(n * 4 * '␣', end='')
    for i in range(1, get_num_days_in_month(year, month) + 1):
        print('%-04d' % i, end='')
        if (i + n) % 7 == 0: print()

def print_monthly_calendar(year, month):
    # 输出year年month月的日历。
    print_header(year, month)
    print_body(year, month)

def test_all_functions():
    # 调用所有以“test____”为名称前缀的测试函数。
    test____is_leap()
    test____get_0101_in_week()
    test____get_m01_in_week()

if __name__ == '__main__':  # 判断模块是否作为一个独立的程序运行
    # sys.argv是一个记录了用户在命令行输入的所有参数的列表。
    # 列表的第一个元素是模块名称，列表的其余元素(若存在)是用户
    # 依次输入的所有参数。
    if len(sys.argv) == 1:
        print(__doc__)  # 用户未输入参数，输出模块的文档
    elif len(sys.argv) == 2 and sys.argv[1] == '-h':
        print(__doc__)  # 用户输入了'-h'，输出模块的文档
    elif len(sys.argv) == 2 and sys.argv[1] == 'test':
        test_all_functions()  # 用户输入了'test'，测试模块中的函数
    else:
```

```
121         import argparse  # 此模块获取用户在命令行输入的年份和月份。
122         parser = argparse.ArgumentParser()
123         # 在每个参数(年份或月份)的输入值前面都要用"--参数名称"的
124         # 格式指定对应的参数名称，参数的顺序无关紧要。
125         parser.add_argument('--year', type=int, default=2022)
126         parser.add_argument('--month', type=int, default=1)
127         args = parser.parse_args()
128         year = args.year; month = args.month
129         print_monthly_calendar(year, month) # 输出日历。
```

4.20 列出了运行模块的示例。在 IPython 中运行程序时，首先进入文件 calendar.py 所在目录，然后在输入 In[2] 行的命令。也可以在操作系统的命令行窗口运行模块，首先进入文件 calendar.py 所在目录，然后输入命令“python month_calendar.py –year 2022 –month 10”。

Listing 4.20: **运行模块 calendar.py 的示例**

```
1  In[1]: cd D:\Python\src
2  Out[1]: D:\Python\src
3  In[2]: run month_calendar.py --year 2022 --month 10
4  2022  10
5  ---------------------------
6  Sun  Mon  Tue  Wed  Thu  Fri  Sat
7                              1
8  2    3    4    5    6    7    8
9  9    10   11   12   13   14   15
10 16   17   18   19   20   21   22
11 23   24   25   26   27   28   29
12 30   31
```

模块除了可以作为一个独立的程序运行，也可以被其他程序导入以后调用其中的函数。如果使用模块的程序和模块文件在同一个目录下时，使用 import 语句导入模块即可使用。例如程序 4.21 调用 month_calendar 模块的 get_m01_in_week 函数以计算给定的某年某月某日是星期几。第 1 行也可以写成“from month_calendar import get_m01_in_week”，表示仅导入 month_calendar 模块的 get_m01_in_week 函数，此时第 3 行的函数调用需写成“get_m01_in_week(y, m)”。

Listing 4.21: **程序 ymd.py 调用 month_calendar 模块的 get_m01_in_week 函数**

```
1 import month_calendar
2 y, m, d = 2022, 9, 18
3 n = (month_calendar.get_m01_in_week(y, m) + d - 1) % 7
4 dw = "Sun␣Mon␣Tue␣Wed␣Thu␣Fri␣Sat"
```

```
print(dw[4*n:4*n+4])  # Sun
```

如果使用模块的程序和模块文件不在同一个目录下时，使用 import 语句导入模块会报错。此时需要将模块所在目录插入到列表 sys.path 中，然后可以导入模块。

Listing 4.22: **将模块所在目录加入到列表 sys.path 中**

```
In[1]: run ymd.py
Out[1]: ... ModuleNotFoundError: No module named 'month_calendar'
In[2]: import sys; sys.path.insert(0, 'D:\Python\src')
In[3]: run ymd.py
Sun
```

程序 4.19 虽然可以正确输出给定年份和月份的日历，但在运行效率上还有改进的余地。给定 y 年 m 月，每次输出日历时都需要计算从 y 年 1 月 1 日到 y 年 m 月 1 日所经历的总天数, 再使用公式 $w(y,m,1) = (w(y,1,1) + v(y,m,1,y,1,1))\ \%\ 7$ 计算 y 年 m 月 1 日是星期几。由于每年的月数和每月的天数 (除了闰年的二月增加一天) 是固定的，$v(y,m,1,y,1,1)$ 对于给定 y 年 m 月的取值只有两种。为了避免多次运行模块时的重复计算，可以事先计算好 $v(y,m,1,y,1,1)\%7$ 并保存在一个表格中，再使用公式

$$w(y,m,1) = (w(y,1,1) + v(y,m,1,y,1,1))\ \%\ 7 = (w(y,1,1) + v(y,m,1,y,1,1)\%7)\ \%\ 7$$

计算 y 年 m 月 1 日是星期几。这一改进的具体实现方式是在程序 4.19 中添加以下两个函数，然后将函数 print_body 的第一条语句 (第 93 行) 改为
n = get_m01_in_week_precomputed(year, month)。

Listing 4.23: **避免多次运行模块 4.19 时的重复计算**

```
def get_num_days_from_0101_to_m01_remainder():
    n = 0; r = [0]
    for i in range(1, 12):
        n += get_num_days_in_month(2022, i)
        r.append(n % 7)
    print(r) # [0, 3, 3, 6, 1, 4, 6, 2, 5, 0, 3, 5]

def get_m01_in_week_precomputed(year, month):
    r = [0, 3, 3, 6, 1, 4, 6, 2, 5, 0, 3, 5]
    n1 = get_0101_in_week(year)
    n2 = r[month-1];
    if (is_leap(year)) and month > 2:
        n2 += 1
    n = (n1 + n2) % 7
    return n
```

本节以输出日历为例介绍创建和使用模块的方法。Python 标准库的 calendar 模块的 TextCalendar 类已提供了输出日历的功能。程序 4.26 的第 3 行的输出结果是 2022 年 10 月的日历。第 4 行的输出结果是 2022 年的日历，关键字实参 m 指定列数。

Listing 4.24: **Python 标准库的 calendar 模块的 TextCalendar 类**

```
1 from calendar import TextCalendar
2 tc = TextCalendar()
3 print(tc.formatmonth(2022, 10))
4 print(tc.formatyear(2022, m=4))
```

4.8 实验 4：函数和模块

实验目的

本实验的目的是掌握以下内容：定义和调用函数，创建和使用模块。

提交方式

在 Blackboard 提交一个文本文件 (txt 后缀)，文件中记录每道题的源程序和运行结果。

实验内容

1. 二分查找

编写一个程序使用二分法查找给定的包含若干整数的列表 s 中是否存在给定的整数 k。使用二分查找的前提是列表已按照从小到大的顺序排序。为此，程序需要先判断 s 是否已经排好序。若未排好序，则需调用 qsort 函数进行排序并输出排序结果。4.25 已列出了部分代码，需要实现函数 is_sorted 和递归函数 binary_search。binary_search 在列表 s 的索引值属于闭区间 [low,high] 的元素中查找 k，若找到则返回 k 的索引值，否则返回-1。binary_search 的设计方案是：首先判断 low 是否大于 high；若是则返回-1；否则计算 low 和 high 的平均值 mid；若 k 等于 s[mid]，则返回 mid；若 k 大于 s[mid] 或小于 s[mid]，则分别确定合适的 low 和 high 值作为实参递归调用 binary_search 并返回结果。

Listing 4.25: **二分查找**

```
1 def is_sorted(s):  # 判断列表s是否已经排好序: to be implemented
2
3 def qsort(s):
4     # 对列表s排序
5     if len(s) <= 1: return s
6     s_less = []; s_greater = []; s_equal = []
7     for k in s:
8         if k < s[0]:
```

```
            s_less.append(k)
        elif k > s[0]:
            s_greater.append(k)
        else:
            s_equal.append(k)
    return qsort(s_less) + s_equal + qsort(s_greater)

def binary_search(s, low, high, k):  # 二分查找: to be implemented

s = [5, 6, 21, 32, 51, 60, 67, 73, 77, 99]
if not is_sorted(s):
    s = qsort(s)
    print(s)

print(binary_search(s, 0, len(s) - 1, 5))  # 0
print(binary_search(s, 0, len(s) - 1, 31)) # -1
print(binary_search(s, 0, len(s) - 1, 99)) # 9
print(binary_search(s, 0, len(s) - 1, 64)) # -1
print(binary_search(s, 0, len(s) - 1, 51)) # 4
```

2. 在程序 4.26 中添加一些语句，使之输出以下字典：{10: 1, 9: 1, 8: 2, 7: 3, 6: 5, 5: 8, 4: 13, 3: 21, 2: 34, 1: 55, 0: 34}。字典中每个元素的键是一个整数 n，其映射到的值是 F(n) 被调用的次数。

Listing 4.26: **计算 Fibonacci 数列的递归方法的重复计算**

```
def F(n):
    if n <= 1:
        return 1
    return F(n-1) + F(n-2)

F(10)
```

3. 有理数的四则运算

有理数的一般形式是 a/b，其中 a 是整数，b 是正整数，并且当 a 非 0 时 $|a|$ 和 b 的最大公约数是 1。编写一个模块 rational.py 实现有理数的四则运算。4.27 已列出了部分代码，需要实现标注了“to be implemented”的函数。程序中用一个列表 [n, d] 表示有理数，其中 n 表示分子，d 表示分母。reduce 函数调用 gcd 函数进行约分。函数 add、sub、mul 和 div 分别进行加减乘除运算，运算的结果都需要约分，并且分母不出现负号。函数 test_all_functions 使用已知答案的数据对这些运算进行测试。函数 output 按照示例的格式输出有理数，例如 [-13,12] 表示的有理数的输出结果是字符串“-13/12”。用户在命令行输入三个命名参数。“–op”表示

运算符，可以是“add”(加法)、“sub”(减法)、“mul”(乘法) 或“div”(除法)。“–x”和“–y”表示进行计算的两个有理数。有理数以字符串的形式输入，必须用圆括号括起，分子和分母之间用“/”分隔。例如有理数“-20/-3”对应的输入形式是 (-20/-3)，用户输入的有理数可以在分母出现负号。函数 get_rational 从表示有理数的字符串中得到列表 [n, d]，例如从字符串“(-20/-3)”得到 [-20,-3]。

Listing 4.27: 有理数的四则运算

```
"""
Module for performing arithmetic operations for rational numbers.

To run the module, user needs to supply three named parameters:
1. op stands for the operation:
    add for addition
    sub for subtraction
    mul for multiplication
    div for division
2. x stands for the first operand
3. y stands for the second operand

x and y must be enclosed in paired parentheses.

For example:

>>> run rational.py --op add --x (2/3) --y (-70/40)
-13/12
>>> run rational.py --op sub --x (-20/3) --y (120/470)
-976/141
>>> run rational.py --op mul --x (-6/19) --y (-114/18)
2/1
>>> run rational.py --op div --x (-6/19) --y (-114/-28)
-28/361
"""

import sys, math

def test_all_functions():  # 测试: to be implemented

def gcd(a, b):
    # 计算正整数a和b的最大公约数
```

```
    while a != b:
        if a > b:
            a -= b
        else:
            b -= a
    return a

def reduce(n, d):  # 约分: to be implemented

def add(x, y):  # 加法: to be implemented

def sub(x, y):  # 减法: to be implemented

def mul(x, y):  # 乘法: to be implemented

def div(x, y):  # 除法: to be implemented

def output(x):  # 按照示例的格式输出有理数: to be implemented

def get_rational(s):  # 从字符串s中得到列表[n, d]: to be implemented

if __name__ == '__main__':
    if len(sys.argv) == 1:
        print(__doc__)
    elif len(sys.argv) == 2 and sys.argv[1] == '-h':
        print(__doc__)
    elif len(sys.argv) == 2 and sys.argv[1] == 'test':
        test_all_functions()
    else:
        import argparse
        parser = argparse.ArgumentParser()
        parser.add_argument('--op', type=str)
        parser.add_argument('--x', type=str)
        parser.add_argument('--y', type=str)
        args = parser.parse_args()
        op = args.op
        x = get_rational(args.x); y = get_rational(args.y)
        f = {'add':add, 'sub':sub, 'mul':mul, 'div':div}
        output(f[op](x, y))
```

第 5 章　类和继承

面向对象的软件设计和开发技术是当前软件开发的主流技术，使得软件更易维护和复用。规模较大的软件大多是基于面向对象技术开发，以类作为基本组成单位。

5.1　定义和使用类

在使用面向对象技术开发软件时，首先通过对软件需求的分析找到问题域中同一类的客观事物，称为对象。把对象共同的属性和运算封装在一起得到的程序单元就是类。类可作为一个独立单位进行开发和测试。以下举例说明。

5.1.1　二维平面上的点

将二维平面上的点视为对象，抽象出其共同的属性和运算，定义一个类表示点。点的属性包括 x 坐标、y 坐标和名称 (默认值为空串)。点的运算包括：给定坐标创建一个点、沿 x 轴平移、沿 y 轴平移、以另一个点为中心旋转、计算与另一个点之间的距离等。这些运算通过函数实现。定义在类内部的函数称为方法 (method)。

程序 5.1 的第 2 行至第 34 行定义了一个 Point2D 类。类的定义由“class 类名 (父类)”开始。第 3 行至第 7 行的方法__init__称为构造方法，用来初始化新创建的对象的所有属性。从一个类创建一个对象的语法是“类名 (实参列表)”，其中的实参列表需要和构造方法的形参列表一一对应。在一个类中的所有方法中出现的属性都需要使用“对象名.”进行限定。self 是一个特殊的对象名，表示当前对象。一个类中的所有方法的第一个形参都是 self。对于一个对象调用其所属类的的方法的语法是“对象名. 方法名 (实参列表)”。Python 规定了类的一些特殊的方法名称，这些方法的名称都以“__”开始和结束。如果一个类定义了这些方法，则调用这些方法时可以使用简化语法。例如对于 Point2D 类的对象 a 而言，简化语法'%s' % a 被转换为与其等价的方法调用 a.__str__()。

Listing 5.1: **Point2D 类**

```
1 import math
2 class Point2D:
```

```
    def __init__(self, x, y, name=''):
        # __init__称为构造方法，用来初始化新创建的对象的所有属性。
        self.x = x  # 用形参x初始化表示点的x坐标的self.x属性。
        self.y = y  # 用形参x初始化表示点的y坐标的self.y属性。
        self.name = name  # 用形参name初始化表示点的名称的name属性。

    def move_x(self, delta_x):
        # 计算self沿x轴平移一段距离delta_x以后的x坐标。
        self.x += delta_x

    def move_y(self, delta_y):
        # 计算self沿y轴平移一段距离delta_y以后的x坐标。
        self.y += delta_y

    def rotate(self, p, t):
        # 计算self以点p为中心旋转角度t以后的坐标。
        xr = self.x - p.x; yr = self.y - p.y
        x1 = p.x + xr * math.cos(t) - yr * math.sin(t)
        y1 = p.y + xr * math.sin(t) + yr * math.cos(t)
        self.x = x1; self.y = y1;

    def distance(self, p):
        # 计算self与另一个点p之间的距离。
        xr = self.x - p.x; yr = self.y - p.y
        return math.sqrt(xr * xr + yr * yr)

    def __str__(self):
        # 返回self的字符串表示。
        if len(self.name) < 1:
            return '(%g, %g)' % (self.x, self.y)
        else:
            return '%s: (%g, %g)' % (self.name, self.x, self.y)

# 创建了一个点a，然后输出其字符串表示。
a = Point2D(-5, 2, 'a'); print(a)  # a: (-5, 2)
# 调用点a的move_x方法将点a沿x轴平移，然后输出其字符串表示。
a.move_x(-1); print(a)             # a: (-6, 2)
# 调用点a的move_y方法将点a沿y轴平移，然后输出其字符串表示。
a.move_y(2); print(a)              # a: (-6, 4)
```

```
42 # 创建了一个点b，然后输出其字符串表示。
43 b = Point2D(3, 4, 'b');  print(b)  # b: (3, 4)
44 # 调用点a的distance方法计算a和b这两个点之间的距离，然后输出结果。
45 print('The distance between a and b is %f' % a.distance(b))
46 # The distance between a and b is 9.000000
47 # 调用点a的rotate方法将点b以点a为中心旋转90度。
48 b.rotate(a, math.pi/2)
49 print(a); print(b)      # a: (-6, 4)   b: (-6, 13)
50 # 调用点a的rotate方法将点a以点b为中心旋转180度。
51 a.rotate(b, math.pi)
52 print(a); print(b)      # a: (-6, 22)  b: (-6, 13)
```

5.1.2 复数

程序 5.2 定义了一个 Complex 类表示复数，并实现了复数的基本运算 (部分代码来源于 [1])。Complex 的属性 re 和 im 分别表示复数的实部和虚部。对于一个对象，__dict__是一个存储了其所有属性名和对应的属性值的字典。test 函数对类中的实现基本运算的方法进行测试。第 11 行将形参 re 和 im 的值分别赋值给 self 对象的属性 re 和 im。复数是一种不可变类型。一个或多个复数对象的计算结果是一个新的复数对象，而不应修改参与运算的复数对象。为了确保不可变性，第 15 行至第 16 行的__setattr__方法拦截对 self 的属性值进行修改的试图并抛出异常。如果按照程序 5.1 的第 5 行和第 6 行那样在第 11 行写“self.re = re; self.im = im”，则也会导致__setattr__方法抛出异常。

Listing 5.2: **Complex 类**

```
1  import math
2
3  class Complex:
4      def __init__(self, re=0, im=0):
5          # 判断形参re和im的类型是不是float或int，
6          # 若不是则以抛出异常的方式报错(第七章介绍异常)。
7          if not isinstance(re, (float,int)) or \
8             not isinstance(im, (float,int)):
9              raise TypeError('Error: float or int expected')
10         self.__dict__['re'] = re; self.__dict__['im'] = im
11         # 将属性名're'和'im'分别映射到对应的值re和im。
12
13     def __setattr__(self, name, value):
14         # 当self的任何属性的值被试图修改时，该方法会被自动调用。
```

[1] https://github.com/xbmc/python/blob/master/Demo/classes/Complex.py

```
        # 方法以抛出异常的方式禁止任何属性的值被修改。
        raise TypeError('Error: Complex objects are immutable')

    def __str__(self):
        # 返回self的字符串表示。
        return '(%g, %g)' % (self.re, self.im)

    def __repr__(self):
        # 返回一个完整表示self的字符串，对该字符串表示的表达式
        # 进行求值可生成self。
        return 'Complex' + str(self)

    def __abs__(self):
        # 返回self的模。对于一个Complex类的对象c，计算模的简化语法
        # abs(c)被转换为与其等价的方法调用c.__abs__()。
        return math.hypot(self.re, self.im)

    abs = __abs__
    # 方法abs是方法__abs__的别名。例如方法调用self.abs()
    # 被转换为与其等价的方法调用self.__abs__()。

    def angle(self):
        # 返回self的幅角。
        return math.atan2(self.im, self.re)

    def __add__(self, other):
        # 对于一个Complex类的对象c，计算c与other的和的简化语法
        # c+other被转换为与其等价的方法调用c.__add__(other)。
        other = to_Complex(other)
        return Complex(self.re + other.re, self.im + other.im)

    __radd__ = __add__
    # 对于一个Complex类的对象c，计算other与c的和的简化语法
    # other+c被转换为与其等价的方法调用c.__radd__(other)。
    # 加法运算是可交换的，c.__radd__(other)被转换为c.__add__(other)。

    def __sub__(self, other):
        # 对于一个Complex类的对象c，计算c与other的差的简化语法
        # c-other被转换为与其等价的方法调用c.__sub__(other)。
```

```
        other = to_Complex(other)
        return Complex(self.re - other.re, self.im - other.im)

    def __rsub__(self, other):
        # 对于一个Complex类的对象c，计算other与c的差的简化语法
        # other-c被转换为与其等价的方法调用c.__rsub__(other)。
        other = to_Complex(other) # 将对象other转换成Complex类的对象。
        return other - self
        # other - self 被转换为方法调用other.__sub__(self)。

    def __mul__(self, other):
        # 对于一个Complex类的对象c，计算c与other的乘积的简化语法
        # c*other被转换为与其等价的方法调用c.__mul__(other)。
        other = to_Complex(other)
        return Complex(self.re*other.re - self.im*other.im,
                       self.re*other.im + self.im*other.re)

    __rmul__ = __mul__
    # 对于一个Complex类的对象c，计算other与c的乘积的简化语法
    # other*c被转换为与其等价的方法调用c.__rmul__(other)。
    # 乘法运算是可交换的，c.__rmul__(other)被转换为c.__mul__(other)。

    def __truediv__(self, other):
        # 对于一个Complex类的对象c，计算c与other的商的简化语法
        # c/other被转换为与其等价的方法调用c.__truediv__(other)。
        other = to_Complex(other)
        d = float(other.re*other.re + other.im*other.im)
        if is_zero(d): # 若除数的模为0，则以抛出异常的方式报错。
            raise ZeroDivisionError('Error: division by 0')
        return Complex((self.re*other.re + self.im*other.im) / d,
                       (self.im*other.re - self.re*other.im) / d)

    def __rtruediv__(self, other):
        # 对于一个Complex类的对象c，计算other与c的商的简化语法
        # other/c被转换为与其等价的方法调用c.__rtruediv__(other)。
        other = to_Complex(other)
        return other / self
        # other / self被转换为方法调用other.__truediv__(self)。

```

```
    def __pow__(self, n):
        # 计算以self为底数和n为指数的乘方运算。
        if is_Complex(n):  # 判断n的类型是不是Complex
            if not is_zero(n.im):  # 判断n的虚部是不是0
                if is_zero(self.im):  # 判断self的虚部是不是0
                    # 对于实数t和复数n，令n ln t = a + bi，
                    # 则t^n = e^{n ln t} = e^{a+ib} = e^a(cos b + i sin b)。
                    z = n * math.log(self.re)
                    r = math.exp(z.re)
                    return Complex(r * math.cos(z.im),
                                   r * math.sin(z.im))
                else:
                    raise NotImplementedError('(a+bi)^(c+di)')
                    # 抛出异常：尚未实现底数和指数都是复数的乘方运算。
            else:
                n = n.re # 已确定n的虚部是0，提取其实部
        # 对于复数t和实数n，令t = re^{iθ}，则t^n = r^n cos nθ + ir^n sin nθ。
        r = self.abs() ** n
        phi = n * self.angle()
        return Complex(r * math.cos(phi), r * math.sin(phi))

    def __rpow__(self, base):
        # 计算以base为底数和self为指数的乘方运算。
        base = to_Complex(base)
        return pow(base, self)

def is_zero(x, tol = 1e-15):
    # 判断一个实数是否等于0的条件定义为其绝对值不超过阈值tol，
    # 其默认值为$10^{-15}$。
    return abs(x) < tol

def is_Complex(obj):
    # 判断一个对象是不是Complex类的对象，依据是它是否同时有re和im
    # 这两个属性。
    return hasattr(obj, 're') and hasattr(obj, 'im')

def to_Complex(obj):
    # 将一个对象obj转换成Complex类的对象。
    if is_Complex(obj): # 对象obj是否已经是Complex类的对象？
```

```
132         return obj       # 若是，直接返回即可。
133     elif isinstance(obj, tuple): # 判断obj的类型是不是元组。
134         if len(obj) <= 2:  # obj包含的元素的个数是否超过2?
135             return Complex(*obj)  # 语法*obj提取obj包含的所有元素。
136         else:
137             raise TypeError('Error:␣<=␣2␣numbers␣expected')
138     else:
139         return Complex(obj)
140
141 def polar_to_Complex(r = 0, phi = 0):
142     # 将极坐标表示的复数re^{iφ}转换为直角坐标表示r cos φ + ir sin φ。
143     return Complex(r * math.cos(phi), r * math.sin(phi))
144
145 def Re(obj):
146     return obj.re if is_Complex(obj) else obj
147     # 在确认形参obj是Complex类的对象后返回其re属性，否则返回obj。
148
149 def Im(obj):
150     return obj.im if is_Complex(obj) else 0
151     # 在确认形参obj是Complex类的对象后返回其im属性，否则返回0。
152
153 def check(expr, a, b, value, verbose = False, rel_tol = 1e-6):
154     # 将a和b的值代入expr中求值，若结果与正确答案value不一致则报错。
155     if verbose: print('␣␣␣␣␣␣␣', a, 'and', b, end = '')
156     try:
157         result = eval(expr)
158         # 内置函数eval对表达式求值，求值环境包括当前可访问的
159         # 所有全局和局部变量。例如第一次运行时expr的值是'a+b',
160         # 形参a和b的值分别是Complex(0, 3)和2, eval将a和b的值
161         # 代入expr中求值。若求值过程中出错，则运行except语句块。
162     except: # 处理求值过程中抛出的异常。
163         print('Error␣in␣evaluating␣' + expr)
164         return
165
166     if verbose: print('␣->␣', result)
167     rel_err = abs(result - value) / abs(value)
168     if rel_err > rel_tol:
169         # 判断相对误差是否超过一个预先确定的阈值rel_tol。
170         print('%s␣for␣a=%s␣and␣b=%s␣=␣%s' % (expr, a, b, result))
```

```
        print('␣␣␣␣Correct␣value␣=␣%s␣␣Relative␣error␣=␣%f' %
              (value, rel_err))


def test(verbose):
    # 测试Complex类实现的复数运算。
    # 字典testsuite包含了所有的测试数据。它的键是要测试的表达式，
    # 每个表达式对应的值是一个由多个元组组成的列表。每个元组包含
    # 3个元素，即表达式中的变量a和b的值以及对表达式求值的正确结果。
    testsuite = {
            'a+b': [ (Complex(0, 3), 2, Complex(2, 3)),
                     (2, Complex(0, 3), Complex(2, 3)),
                     (Complex(2,-3), Complex(-4,5),
                      Complex(-2,2)) ],
            'a-b': [ (Complex(0, 3), 2, Complex(-2, 3)),
                     (2, Complex(0, 3), Complex(2, -3)),
                     (Complex(2,-3), Complex(-4,5),
                      Complex(6,-8)) ],
            'a*b': [ (Complex(0, 3), 2, Complex(0, 6)),
                     (2, Complex(0, 3), Complex(0, 6)),
                     (Complex(2,-3), Complex(-4,5),
                      Complex(7,22)) ],
            'a/b': [ (Complex(0, 3), 2, Complex(0, 1.5)),
                     (2, Complex(0, 3), Complex(0, -0.6666667)),
                     (Complex(2, -3), Complex(-4, 5),
                      Complex(-0.5609756, 0.04878049)) ],
            'pow(a,b)': [ (Complex(2, -3), 2.3,
                           Complex(-12.15244, -14.73536)),
                           (2.3, Complex(2, -3),
                            Complex(-4.234017, -3.171309)) ],
            'polar_to_Complex(a.abs(),␣a.angle())': [
                    (Complex(0, -3), 0, Complex(0, -3)),
                    (Complex(2, -3), 0, Complex(2, -3)),
                    (Complex(-1, -3), 0, Complex(-1, -3)) ]
    }
    for expr in testsuite:
        if verbose: print(expr + ':')
        t = (expr,)
        for item in testsuite[expr]:
```

```
            check(*(t + item), verbose)

if __name__ == '__main__':
    # verbose变量是调用test方法的实参，用于控制程序的输出篇幅。
    # 若verbose的值设为True，则会显示每条测试数据并在发生错误时报错。
    # 若verbose的值设为False，则仅在发生错误时报错。
    verbose = False
    test(verbose)
```

5.1.3 一元多项式

程序 5.3 定义了一个类 Polynomial 表示一元多项式，并实现了一些基本运算 (部分代码来源于 [HL2020]) 多项式中的每一项的指数和其对应的系数存储在一个字典 poly 中。如果在运算结果中某一项的系数的绝对值小于一个预先定义的阈值 tol(默认值为 10^{-15})，则认为系数等于零，该项消失。

Listing 5.3: **Polynomial 类**

```
tol = 1E-15
class Polynomial:
    def __init__(self, poly):
        self.poly = {}
        for power in poly:
            if abs(poly[power]) > tol:
                self.poly[power] = poly[power]

    def __call__(self, x):
        # 对多项式p(x)在x=t时求值的语法简化p(t)被转换为与其
        # 等价的方法调用p.__call__(t)。
        value = 0.0
        for power in self.poly:
            value += self.poly[power]*x**power
        return value

    def __add__(self, other):
            # 计算self+other
        sum = self.poly.copy()    # 从self.poly复制一个副本sum
        # 对于other中每一项的指数查找在sum中是否存在相同指数的项，
        # 若存在则执行这两项的系数的加法，否则创建一个新的项并设置
        # 其系数为other中这一项的系数。
```

```
        for power in other.poly:
            if power in sum:
                sum[power] += other.poly[power]
            else:
                sum[power] = other.poly[power]
        return Polynomial(sum) # 调用构造方法创建一个新的多项式。

    def __mul__(self, other):
        # 根据指数相加和系数相乘的规则计算self*other。
        sum = {}
        for self_power in self.poly:
            for other_power in other.poly:
                power = self_power + other_power
                m = self.poly[self_power] * \
                    other.poly[other_power]
                if power in sum:
                    sum[power] += m
                else:
                    sum[power] = m
        return Polynomial(sum)  # 调用构造方法创建一个新的多项式。

    def __str__(self):
        # 返回多项式的字符串表示，处理了多种特殊情形使得
        # 输出结果符合数学表达习惯。
        s = ''
        for power in sorted(self.poly):
            s += '␣+␣%g*x^%d' % (self.poly[power], power)
        s = s.replace('+␣-', '-␣')
        s = s.replace('x^0', '1')
        s = s.replace('␣1*', '␣')
        s = s.replace('x^1␣', 'x␣')
        # s = s.replace('x^1', 'x') replaces x^100 by x^00
        if s[0:3] == '␣+␣':  # remove initial +
            s = s[3:]
        if s[0:3] == '␣-␣':  # fix spaces for initial -
            s = '-' + s[3:]
        return s

# 创建了一个多项式对象p1并输出其字符串表示。
```

```
p1 = Polynomial({0: -1, 2: 1, 7: 3}); print(p1)
#  -1 + x^2 + 3*x^7
# 创建了一个多项式对象p2并输出其字符串表示。
p2 = Polynomial({0: 1, 2: -1, 5: -2, 3: 4}); print(p2)
#  1 - x^2 + 4*x^3 - 2*x^5
p3 = p1 + p2; print(p3)
#  4*x^3 - 2*x^5 + 3*x^7
p4 = p1 * p2; print(p4)
#  -1 + 2*x^2 - 4*x^3 - x^4 + 6*x^5 + x^7 - 3*x^9 + 12*x^10
#   - 6*x^12
print(p4(5))   #  对多项式p4在x=5时的求值结果是-1353419826.0
```

5.2 继承

利用有限差分可以近似计算函数 $f(x)$ 的一阶导数。根据泰勒公式可将 $f(x)$ 在 x 的邻域展开如下:

$$
\begin{aligned}
f(x-2h) &= f(x) - 2hf'(x) + \frac{4h^2 f''(x)}{2} - \frac{8h^3 f'''(x)}{6} + \frac{16h^4 f^{(4)}(x)}{24} - \frac{32h^5 f^{(5)}(x)}{120} + \cdots \\
f(x-h) &= f(x) - hf'(x) + \frac{h^2 f''(x)}{2} - \frac{h^3 f'''(x)}{6} + \frac{h^4 f^{(4)}(x)}{24} - \frac{h^5 f^{(5)}(x)}{120} + \cdots \\
f(x+h) &= f(x) + hf'(x) + \frac{h^2 f''(x)}{2} + \frac{h^3 f'''(x)}{6} + \frac{h^4 f^{(4)}(x)}{24} + \frac{h^5 f^{(5)}(x)}{120} + \cdots \\
f(x+2h) &= f(x) + 2hf'(x) + \frac{4h^2 f''(x)}{2} + \frac{8h^3 f'''(x)}{6} + \frac{16h^4 f^{(4)}(x)}{24} + \frac{32h^5 f^{(5)}(x)}{120} + \cdots
\end{aligned}
\tag{5.1}
$$

由此可以推导出以下这些按照精确度从低到高次序列出的计算数值一阶导数的有限差分公式。这些公式依次称为一阶向前差分、一阶向后差分、二阶中心差分和四阶中心差分。

$$
\begin{aligned}
f'(x) &= \frac{f(x+h) - f(x)}{h} + O(h) \\
f'(x) &= \frac{f(x) - f(x-h)}{h} + O(h) \\
f'(x) &= \frac{f(x+h) - f(x-h)}{2h} + O(h^2) \\
f'(x) &= \frac{4}{3}\frac{f(x+h) - f(x-h)}{2h} - \frac{1}{3}\frac{f(x+2h) - f(x-2h)}{4h} + O(h^4)
\end{aligned}
\tag{5.2}
$$

如果每个公式都用一个类实现，则这些类都有属性 f 和 h，并且它们的构造方法是相同的。这就导致了大量重复代码。面向对象程序编程范式提供了继承机制，新类可从已有类获得属性和方法并进行扩展，实现了代码复用。新类称为子类或派生类。已有类称为父类或基类。对于每个公式，都需要比较其计算结果和精确结果的差别。可为这些公式类定义一个共同的父类

Differentiation，其中包含了属性 f 和 h，以及比较计算结果的方法。通过继承，这些公式类可以获得这些属性和方法 [HL2020]。

程序 5.4 的前第 11 行定义了 Differentiation 类。第 5 行的 dfdx_exact 表示函数 $f(x)$ 的解析形式的一阶导函数 $f'(x)$，其默认值为 None。如果调用构造方法时提供了 dfdx_exact，则可计算微分的精确结果。第 13 行至第 32 行定义了对应这四个公式的四个类。这些类的定义的第一条语句中的“(Differentiation)”表示其父类是 Differentiation。由于从父类继承了所有属性和构造方法，这些类只需实现__call__方法进行公式计算。第 66 行至第 67 行调用 table 函数的输出结果 5.5 表明这些有限差分公式的精确度符合理论预期的从低到高次序。

Listing 5.4: **数值微分类**

```
1  class Differentiation:
2      def __init__(self, f, h=1E-5, dfdx_exact=None):
3          self.f = f
4          self.h = float(h)
5          self.exact = dfdx_exact
6
7      def get_error(self, x):
8          # 计算数值结果和精确结果之间的相对误差
9          if self.exact is not None:
10             df_numerical = self(x)
11             df_exact = self.exact(x)
12             return abs( (df_exact - df_numerical) / df_exact )
13
14 class Forward1(Differentiation):
15     # 一阶向前差分
16     def __call__(self, x):
17         f, h = self.f, self.h
18         return (f(x+h) - f(x))/h
19
20 class Backward1(Differentiation):
21     # 一阶向后差分
22     def __call__(self, x):
23         f, h = self.f, self.h
24         return (f(x) - f(x-h))/h
25
26 class Central2(Differentiation):
27     # 二阶中心差分
28     def __call__(self, x):
```

```
        f, h = self.f, self.h
        return (f(x+h) - f(x-h))/(2*h)

class Central4(Differentiation):
    # 四阶中心差分
    def __call__(self, x):
        f, h = self.f, self.h
        return (4./3)*(f(x+h) - f(x-h))  /(2*h) - \
               (1./3)*(f(x+2*h) - f(x-2*h))/(4*h)

def table(f, x, h_values, methods, dfdx=None):
    # 基于输入函数f、自变量x、步长值列表h_values、公式类列表methods
    # 和函数f(x)的解析形式的一阶导函数dfdx生成一个表格。
    # 表格中的每一项数据表示对应于一个特定公式和特定步长值的数值结果
    # 和精确结果之间的相对误差。
    print('%-10s' % 'h', end='␣')
    for h in h_values: print('%-8.2e' % h, end='␣')
    print()
    for method in methods:
        print('%-10s' % method.__name__, end='␣')
        for h in h_values:
            if dfdx is not None:
                d = method(f, h, dfdx)
                output = d.get_error(x)
            else:
                d = method(f, h)
                output = d(x)
            print('%-8.6f' % output, end='␣')
        print()

import math
def g(x): return math.exp(x*math.sin(x))

# 使用SymPy库(第十章)计算函数g(x) = e^{x sin x}的解析形式的一阶
# 导函数g'(x) = (x cos x + sin x)e^{x sin x}并保存在dgdx中。
import sympy as sym
sym_x = sym.Symbol('x')  # sym_x是一个sympy变量
sym_gx = sym.exp(sym_x*sym.sin(sym_x)) # sym_gx是一个sympy函数
sym_dgdx = sym.diff(sym_gx, sym_x) # sym_dgdx是sym_gx的一阶导函数
```

```
dgdx = sym.lambdify([sym_x], sym_dgdx)
# lambdify把sym_dgdx转换成一个可以被调用的Python函数

table(f=g, x=-0.65, h_values=[10**(-k) for k in range(1, 7)],
      methods=[Forward1, Central2, Central4], dfdx=dgdx)
```

Listing 5.5: **程序 5.4 的输出结果**

```
h          1.00e-01 1.00e-02 1.00e-03 1.00e-04 1.00e-05 1.00e-06
Forward1   0.104974 0.010906 0.001095 0.000110 0.000011 0.000001
Central2   0.004611 0.000046 0.000000 0.000000 0.000000 0.000000
Central4   0.000080 0.000000 0.000000 0.000000 0.000000 0.000000
```

子类可以对父类进行功能上的扩展，例如在子类中定义父类中没有的属性和方法。子类还可以重新定义从父类继承的方法，称为覆盖 (overriding)。覆盖的规则是子类中定义的某个方法和父类中的某个方法在名称、形参列表和返回类型上都相同，但方法体不同。覆盖体现了子类和父类在功能上的差异。

程序 5.6 中定义了一个父类 Parent 和它的三个子类：Child1、Child2 和 Child3。Parent 定义了一个属性 c 和一个方法 m。Child1 定义了一个属性 d 并覆盖了父类的方法 m。Child1 的构造方法的第一条语句 super().__init__() 通过 super 函数调用其父类的构造方法，以便初始化从父类继承的属性 c。Child2 未定义构造方法。Child2 中的方法 m 覆盖了父类的方法 m，其方法体的第一条语句 super().m() 调用其父类的 m 方法，方法体的其余语句的作用可理解为对父类的 m 方法已实现的功能进行补充。Child3 定义了一个属性 f 和一个方法 m2，并从父类继承了方法 m。第 29 行至第 31 行的输出结果表明：如果子类覆盖了父类的方法，则子类对象调用的方法 m 是子类中重新定义的方法。如果子类未覆盖父类的方法，则子类对象调用的方法是从父类中继承的方法 m。第 33 行至第 35 行的输出结果表明：若一个子类定义了构造方法，则子类定义的构造方法覆盖了其父类的构造方法，此时若子类如果仍然需要从其父类继承属性，子类的构造方法必须包含语句 super().__init__()；若一个子类未定义构造方法，则继承了其父类的构造方法，因此自动从其父类继承属性。

Listing 5.6: **覆盖**

```
class Parent:
    def __init__(self):
        self.c = 1
    def m(self):
        print('Calling m in class Parent')

class Child1(Parent):
    def __init__(self):
```

```
9          # 覆盖了父类的构造方法
10         super().__init__()  # 通过super函数调用其父类的构造方法
11         self.d = 2
12     def m(self):  # 覆盖了父类的m方法
13         print('Calling␣m␣in␣class␣Child1')
14
15 class Child2(Parent):
16     def m(self):
17         # 覆盖了父类的m方法
18         super().m()  # 通过super函数调用其父类的m方法
19         print('Calling␣m␣in␣class␣Child2')
20
21 class Child3(Parent):
22     def __init__(self):
23         # 覆盖了父类的构造方法
24         self.f = 3
25     def m2(self):
26         print('Calling␣m2␣in␣class␣Child3')
27
28 c1 = Child1()            # c1是Child1类的对象
29 c2 = Child2()            # c2是Child2类的对象
30 c3 = Child3()            # c3是Child3类的对象
31 p = Parent()             # p是Parent类的对象
32 c1.m()                   # 调用Child1类的m方法
33 c2.m()                   # 调用Child2类的m方法
34 c3.m()                   # 调用Parent类的m方法
35 p.m()                    # 调用Parent类的m方法
36 print(c1.__dict__)       #  {'c': 1, 'd': 2}
37 print(c2.__dict__)       #  {'c': 1}
38 print(c3.__dict__)       #  {'f': 3}
```

5.3 迭代器和生成器

第二章介绍的所有序列类型 (包括 list、tuple 和 str 等) 和 set、map 等类型的对象称为可迭代对象 (iterable)，即可以在 for 循环中遍历其包含的所有元素。可迭代对象是实现了迭代器 (Iterator) 协议的对象。迭代器协议的要求是定义一个___iter___方法，该方法应返回一个实现了___next___方法的对象，___next___方法的作用是返回下一个迭代值。

程序 5.7 的前 8 行定义了 Fibonacci 类，它的__next__方法生成下一个 Fibonacci 数。如果下一个 Fibonacci 数超过了属性 ub 表示的指定上界，则停止。第 11 行至第 16 行定义了 FibonacciIterator 类，它的__iter__方法返回一个 Fibonacci 类的对象。第 19 行至第 20 行的 for 循环输出了第 18 行创建的 FibonacciIterator 类的对象所生成的 Fibonacci 数列中不超过 64 的前几个数。第 22 行生成并输出了一个包含这几个数的列表。

Listing 5.7: **生成 Fibonacci 数列的可迭代对象**

```
1  class Fibonacci:
2      def __init__(self, ub):
3          self.a = 0; self.b = 1; self.ub = ub
4
5      def __next__(self):
6          # 生成下一个Fibonacci数
7          self.a, self.b = self.b, self.a + self.b
8          if self.a > self.ub:  # 如果下一个数超过了指定上界，则停止。
9              raise StopIteration
10         return self.a
11
12 class FibonacciIterator:
13     def __init__(self, ub):
14         self.ub = ub
15
16     def __iter__(self):
17         return Fibonacci(self.ub)
18
19 fibonacciIterator = FibonacciIterator(64)
20 for f in fibonacciIterator:
21     print (f, end = '␣')        # 1 1 2 3 5 8 13 21 34 55
22 print()
23 print(list(fibonacciIterator))
24 # [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

生成器 (generator) 是一种特殊的函数，特殊性体现在它包含的 yield 语句。生成器使用 yield 语句每次产生一个值，然后停止运行直至被唤醒，被唤醒后从停止的位置开始继续运行。生成器返回的是一个可迭代对象，对其迭代可得到 yield 语句产生的每个值 [MH2017]。

程序 5.8 的前 7 行定义了一个递归函数 flatten，它可以从一个包含任意重数嵌套的列表 nested_list 中提取所有数值。第 2 行判断递归的终止条件是否成立，即 nested_list 已经是数值而不是列表，若是则产生该数值。否则，nested_list 是一个列表。第 5 行的循环遍历组成 nested_list 的各元素 sub_list，sub_list 可能是数值或者列表。第 6 行对于每个 sub_list 递归

调用 flatten。由于 flatten 是一个生成器，它返回的是一个可迭代对象，第 6 行至第 7 行的 for 循环遍历了此可迭代对象的所有元素。第 10 行对于第 9 行定义的列表 l 调用 flatten，并通过 for 循环在第 11 行输出其返回的可迭代对象的所有元素。第 13 行生成并输出了一个包含这些元素的列表。

Listing 5.8: 生成器

```
1  def flatten(nested_list):
2      if not isinstance(nested_list, list):
3          yield nested_list
4      else:
5          for sub_list in nested_list:
6              for element in flatten(sub_list):
7                  yield element
8
9  l = [1, [2, [3, 4, 5, [6, 7]], 8], 9]
10 for e in flatten(l):
11     print (e, end = '␣')  # 1 2 3 4 5 6 7 8 9
12 print()
13 print(list(flatten(l)))  # [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

5.4 实验 5：类和继承

实验目的

本实验的目的是掌握以下内容：定义和使用类，通过继承和覆盖实现代码复用。

提交方式

在 Blackboard 提交一个文本文件 (txt 后缀)，文件中记录每道题的源程序和运行结果。

实验内容

1. 表示有理数的类

有理数的一般形式是 a/b，其中 a 是整数，b 是正整数，并且当 a 非 0 时 $|a|$ 和 b 的最大公约数是 1。实现 Rational 类表示有理数和其运算。5.9 已列出了部分代码，需要实现标注了“to be implemented”的函数。Rational 类的属性 nu 和 de 分别表示分子和分母。函数__add__、__sub__、__mul__和__truediv__分别进行加减乘除运算，然后返回一个新创建的 Rational 对象作为运算结果。函数__eq__、__ne__、__gt__、__lt__、__ge__和__le__比较两个有理数，返回一个 bool 类型的值。这些函数对应的比较运算符分别是：==、!=、>、<、>=、<=。例如表达式 Rational(6, -19) > Rational(14, -41) 在求值时被转

换成方法调用 Rational(6, -19).__gt__(Rational(14, -41))。函数 test 测试这些函数。gcd 函数要求形参 a 和 b 都是正整数，如果其中出现 0 或负数，递归不会终止。

Listing 5.9: 表示有理数的类

```
def gcd(a, b):
    # 计算正整数a和b的最大公约数
    while a != b:
        if a > b:
            a -= b
        else:
            b -= a
    return a

class Rational:
    def __init__(self, n=0, d=1):  # to be implemented
        # 将[n,d]表示的有理数转换为标准形式，例如120/-64转换为-15/8
        _nu = n; _de = d
        self.__dict__['nu'] = _nu; self.__dict__['de'] = _de

    def __setattr__(self, name, value):
        raise TypeError('Error: Rational objects are immutable')

    def __str__(self): return '%d/%d' % (self.nu, self.de)

    def __add__(self, other):  # 加法: to be implemented

    def __sub__(self, other):  # 减法: to be implemented

    def __mul__(self, other):  # 乘法: to be implemented

    def __truediv__(self, other):  # 除法: to be implemented

    def __eq__(self, other):  # ==: to be implemented

    def __ne__(self, other):  # !=: to be implemented

    def __gt__(self, other):  # $>$: to be implemented

    def __lt__(self, other):  # $<$: to be implemented
```

```

    def __ge__(self, other):  # $>=$: to be implemented

    def __le__(self, other):  # $<=$: to be implemented

def test():
    testsuite = [
        ('Rational(2, 3) + Rational(-70, 40)',
          Rational(-13, 12)),
        ('Rational(-20, 3) - Rational(120, 470)',
          Rational(-976,141)),
        ('Rational(-6, 19) * Rational(-114, 18)',
          Rational(2, 1)),
        ('Rational(-6, 19) / Rational(-114, -28)',
          Rational(-28,361)),

        ('Rational(-6, 19) == Rational(-14, 41)', False),
        ('Rational(-6, 19) != Rational(-14, 41)', True),
        ('Rational(6, -19) > Rational(14, -41)', True),
        ('Rational(-6, 19) < Rational(-14, 41)', False),
        ('Rational(-6, 19) >= Rational(-14, 41)', True),
        ('Rational(6, -19) <= Rational(14, -41)', False),
        ('Rational(-15, 8) == Rational(120, -64)', True),
    ]
    for t in testsuite:
        try:
            result = eval(t[0])
        except:
            print('Error in evaluating ' + t[0]); continue

        if result != t[1]:
            print('Error:  %s != %s' % (t[0], t[1]))

if __name__ == '__main__':
    test()
```

2. 定积分的数值计算

函数 $f(x)$ 在区间 $[a,b]$ 上的定积分可用区间内选取的 $n+1$ 个点 x_i $(i=0,1,...,n)$(称为积分节

点) 上的函数值的加权和近似计算：

$$\int_a^b f(x)\,dx \approx \sum_{i=0}^{n} w_i f(x_i)$$

其中 w_i 是函数值 $f(x_i)$ 的权重，称为积分系数。不同的数值计算公式的区别体现在积分节点和积分系数上。

公式名称	积分节点的坐标和积分系数
复合梯形 (Trapezoidal) 公式	$x_i = a + ih \quad \text{for} \quad i = 0, ..., n, \quad h = \frac{b-a}{n}$, $w_0 = w_n = \frac{h}{2}, \quad w_i = h \quad \text{for} \quad i = 1, ..., n-1$
复合辛普森 (Simpson) 公式 n 必须是偶数 若输入的 n 是奇数，则执行 $n = n + 1$	$x_i = a + ih \quad \text{for} \quad i = 0, ..., n, \quad h = \frac{b-a}{n}$, $w_0 = w_n = \frac{h}{3}, \quad w_i = \frac{2h}{3} \quad \text{for} \quad i = 2, 4, ..., n-2$, $w_i = \frac{4h}{3} \quad \text{for} \quad i = 1, 3, ..., n-1$
复合高斯-勒让德 (Gauss-Legendre) 公式 n 必须是奇数 若输入的 n 是偶数，则执行 $n = n + 1$	$x_i = a + \frac{i+1}{2}h - \frac{\sqrt{3}}{6}h \quad \text{for} \quad i = 0, 2, ..., n-1$, $x_i = a + \frac{i}{2}h + \frac{\sqrt{3}}{6}h \quad \text{for} \quad i = 1, 3, ..., n$, $h = \frac{2(b-a)}{n+1}, \quad w_i = \frac{h}{2}, \quad \text{for} \quad i = 0, 1, ..., n$

表 5.1: 定积分的几种数值计算公式

在程序 5.10 中实现 Integrator 类的 integrate 方法和它的三个子类，分别对应表 5.1 中的三种公式。在每个子类中只需覆盖父类的 compute_points 方法计算并返回两个列表，它们分别存储了所有积分节点的坐标和积分系数。test() 函数用函数 $f(x) = (x\cos x + \sin x)e^{x\sin x}$ 和它的解析形式的积分函数 $F(x) = e^{x\sin x}$ 测试这三个公式的精确度。

Listing 5.10: **定积分的数值计算**

```
import math

class Integrator:
    def __init__(self, a, b, n):
        self.a, self.b, self.n = a, b, n
        self.points, self.weights = self.compute_points()

    def compute_points(self):
        raise NotImplementedError(self.__class__.__name__)

    def integrate(self, f):  # to be implemented
    # 将self.points和self.weights代入计算公式求和

class Trapezoidal(Integrator):
    def compute_points(self):  # to be implemented
```

```python

class Simpson(Integrator):
    def compute_points(self):  # to be implemented

class GaussLegendre(Integrator):
    def compute_points(self):  # to be implemented

def test():
    def f(x): return (x * math.cos(x) + math.sin(x)) * \
                      math.exp(x * math.sin(x))
    def F(x): return math.exp(x * math.sin(x))

    a = 2; b = 3; n = 200
    I_exact = F(b) - F(a)
    tol = 1E-3

    methods = [Trapezoidal, Simpson, GaussLegendre]
    for method in methods:
        integrator = method(a, b, n)
        I = integrator.integrate(f)
        rel_err = abs((I_exact - I) / I_exact)
        print('%s: %g' % (method.__name__, rel_err))
        if rel_err > tol:
            print('Error in %s' % method.__name__)

if __name__ == '__main__':
    test()
```

第 6 章　NumPy 数组和矩阵计算

NumPy 扩展库 [NumPyDoc] 定义了由同类型的元素组成的多维数组 ndarray 及其常用运算，ndarray 是科学计算中最常用的数据类型。NumPy 数组相对列表的优势是运算速度更快和占用内存更少。ndarray 是一个类，它的别名是 array。它的主要属性包括 ndim(维数)、shape(形状，即由每个维度的长度构成的元组) 、size(元素数量) 和 dtype(元素类型：可以是 Python 的内置类型，也可以是 NumPy 定义的类型，例如 numpy.int32、numpy.int16 和 numpy.float64 等)。

矩阵使用 numpy.array 类表示。SciPy 扩展库的 scipy.linalg 模块提供了常用的矩阵计算函数 [SciPyDoc]。scipy.sparse 模块提供了多个表示稀疏矩阵的类。scipy.sparse.linalg 模块提供了进行稀疏矩阵计算的函数。

6.1　创建数组

6.1.1　已有元素存储在其他类型的容器中

可以从存储了元素的列表、元组或它们的嵌套创建数组，其类型取决于元素的类型，也可以使用 dtype 关键字实参指定类型。如果元素无法使用指定类型表示，可能会发生溢出或精度损失。

Listing 6.1: 创建数组

```
In[1]: import numpy as np
In[2]: a = np.array([2, 8, 64]); a
Out[2]: array([2, 8, 64])
In[3]: a.dtype, a.ndim, a.shape, a.size
Out[3]: (dtype('int32'), 1, (3,), 3)
In[4]: b = np.array([3.14, 2.71, 6.83, -8.34])
In[5]: b.dtype, b.ndim, b.shape, b.size
Out[5]: (dtype('float64'), 1, (4,), 4)
In[6]: c = np.array([(1, 2.4), (6, -3), (8, -5)])
In[7]: c.ndim, c.shape, c.size
```

```
Out[7]:  (2, (3, 2), 6)
In[8]: d = np.array([95536, 2.71, 6, -8.34], dtype=np.int16); d
Out[8]: array([30000,     2,     6,    -8], dtype=int16))
```

6.1.2 没有元素但已知形状

np.zeros 函数和 np.ones 函数创建指定形状的数组，并分布用 0 和 1 填充所有元素。zeros_like 函数和 ones_like 函数创建和已有数组具有相同形状的数组，并分布用 0 和 1 填充所有元素。np.arange 函数根据下界、上界和步长生成一个由等差数列组成的数组，包括下界但不包括上界。np.linspace 函数根据下界、上界和数量生成一个由包含指定数量的元素的等差数列组成的数组，包括下界和上界。numpy 库的 random 模块的 seed 函数设置随机数种子为 10，rand 函数可生成一些服从区间 [0,1) 内的均匀分布的随机数用以初始化一个指定形状的随机数组。如果设置随机数种子为一个常数，则每次运行 rand 函数得到相同的随机数组。如果在调用 seed 函数时不提供实参，则每次运行的结果不同。np.fromfunction 函数创建指定形状的数组，每个元素的值是这个元素的索引值的函数，该函数是传给 np.fromfunction 的第一个实参。

Listing 6.2: **创建数组**

```
In[1]: a = np.zeros((2, 3)); a
Out[1]:
array([[0., 0., 0.],
       [0., 0., 0.]])
In[2]: np.ones((3, 2))
Out[2]:
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
In[3]: c = np.ones_like(a); c
Out[3]:
array([[1., 1., 1.],
       [1., 1., 1.]])
In[4]: np.arange(2, 30, 7)
Out[4]: array([ 2,  9, 16, 23])
In[5]: np.arange(0.2, 3.01, 0.7)
Out[5]: array([0.2, 0.9, 1.6, 2.3, 3. ])
In[6]: np.arange(6)
Out[6]: array([0, 1, 2, 3, 4, 5])
In[7]: np.linspace(0, 3, 7)
Out[7]: array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. ])
In[8]: np.random.seed(10); np.random.rand(3, 2)
```

```
Out[8]:
array([[0.77132064, 0.02075195],
       [0.63364823, 0.74880388],
       [0.49850701, 0.22479665]])
In[9]: def f(x, y): return (x + 2) ** 2 + y ** 3
In[10]: np.fromfunction(f, (2, 3), dtype=int)
Out[10]: array([[ 4,  5, 12],
                [ 9, 10, 17]])
```

6.1.3 改变数组的形状

改变形状是指改变各维度的长度，但不改变组成数组的元素。flatten 方法从一个多维数组生成一维数组。reshape 方法从原数组生成一个指定形状的新数组。T 方法从一个数组 h 生成它的转置。以上方法不会改变原数组的形状。resize 方法则将原数组改变为指定的形状。

Listing 6.3: 改变形状

```
In[1]: h=np.arange(1,13).reshape(3,4); h
Out[1]:
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
In[2]: h.flatten()
Out[2]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
In[3]: h.reshape(2, 6)
Out[3]:
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12]])
In[4]: h.T
Out[4]:
array([[ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11],
       [ 4,  8, 12]])
In[5]: h
Out[5]:
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
In[6]: h.resize(2, 6); h
```

```
Out[6]:
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12]])
```

6.1.4 数组的堆叠 (Stacking)

np.hstack 函数沿第二个维度将两个数组堆叠在一起形成新的数组，np.vstack 函数沿第一个维度将两个数组堆叠在一起形成新的数组。np.r_函数将多个数组 (数值) 堆叠在一起形成新的数组，其中语法 start:stop:step 等同于 np.arange(start, stop, step)，语法 start:stop:stepj 等同于 np.linspace(start, stop, step, endpoint=1)。

Listing 6.4: **堆叠**

```
In[1]: a = np.arange(1, 7).reshape(2,3); a
Out[1]:
array([[1, 2, 3],
       [4, 5, 6]])
In[2]: b = np.arange(7, 13).reshape(2,3); b
Out[2]:
array([[ 7,  8,  9],
       [10, 11, 12]])
In[3]: np.hstack((a, b))
Out[3]:
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
In[4]: np.vstack((a, b))
Out[4]:
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
In[5]: np.r_[np.array([1,3,7]), 0, 8:2:-2, 0]
Out[5]: array([1, 3, 7, 0, 8, 6, 4, 0])
In[6]: np.r_[-1:2:6j, [1]*2, 5]
Out[6]: array([-1. , -0.4,  0.2,  0.8,  1.4,  2. ,  1. ,  1. ,
                5. ])
```

6.1.5 数组的分割

np.hsplit 函数沿第二个维度将一个数组分割成为多个数组，可以指定一个正整数表示均匀分割得到的数组的数量或指定一个元组表示各分割点的索引值。np.vsplit 函数沿第一个维度将一个数组分割成为多个数组，参数和 hsplit 类似。

Listing 6.5: 分割

```
In[1]: c = np.arange(1, 25).reshape(2,12); c
Out[1]:
array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12],
       [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]])
In[2]: np.hsplit(c, 4)
Out[2]:
[array([[ 1,  2,  3],
        [13, 14, 15]]),
 array([[ 4,  5,  6],
        [16, 17, 18]]),
 array([[ 7,  8,  9],
        [19, 20, 21]]),
 array([[10, 11, 12],
        [22, 23, 24]])]
In[3]: np.hsplit(c, (4, 7, 9))
Out[3]:
[array([[ 1,  2,  3,  4],
        [13, 14, 15, 16]]),
 array([[ 5,  6,  7],
        [17, 18, 19]]),
 array([[ 8,  9],
        [20, 21]]),
 array([[10, 11, 12],
        [22, 23, 24]])]
In[4]: d = np.arange(1, 25).reshape(6,4); d
Out[4]:
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16],
       [17, 18, 19, 20],
       [21, 22, 23, 24]])
In[5]: np.vsplit(d, (2, 3, 5))
```

```
Out[5]:
[array([[1, 2, 3, 4],
        [5, 6, 7, 8]]),
 array([[ 9, 10, 11, 12]]),
 array([[13, 14, 15, 16],
        [17, 18, 19, 20]]),
 array([[21, 22, 23, 24]])]
```

6.2 数组的运算

6.2.1 基本运算

数组的一元运算 (取反、乘方) 和二元运算 (加减乘除) 对于每个元素分别进行。两个二维数组之间的矩阵乘法可通过@运算符或 dot 方法完成。二元运算要求两个数组具有相同的形状。数组和单个数值之间的运算等同于数组的每个元素分别和数值之间进行运算。“+=”、“-=”、“*=”、“/=” 和 “**=” 运算符不返回新的数组，而是用运算结果替代原数组。如果参与运算的多个数组的元素的类型不同，则结果的类型设为取值范围最大的类型。

Listing 6.6: **基本运算**

```
In[1]: a = np.arange(6).reshape(2, 3); a
Out[1]:
array([[0, 1, 2],
       [3, 4, 5]])
In[2]: b = np.arange(2,18,3).reshape(2, 3); b
Out[2]:
array([[ 2,  5,  8],
       [11, 14, 17]])
In[3]: a+b, a-b, a*b, a/b, -a, -b+(a**np.e-0.818*b+6)**(-np.pi)
Out[3]:
(array([[ 2,  6, 10],
        [14, 18, 22]]),
 array([[ -2,  -4,  -6],
        [ -8, -10, -12]]),
 array([[ 0,  5, 16],
        [33, 56, 85]]),
 array([[0.        , 0.2       , 0.25      ],
        [0.27272727, 0.28571429, 0.29411765]]),
 array([[ 0, -1, -2],
        [-3, -4, -5]]),
```

```
 array([[ -1.99023341,  -4.96511512,  -7.99647626],
        [-10.99985895, -13.99998898, -16.99999851]]))
In[4]: c = b.reshape(3, 2); c
Out[4]:
array([[ 2,  5],
       [ 8, 11],
       [14, 17]])
In[5]: a@c, a.dot(c)
Out[5]:
(array([[ 36,  45],
        [108, 144]]),
 array([[ 36,  45],
        [108, 144]]))
In[6]: d=a*3+b; b -= a; d, b
Out[6]:
(array([[ 2,  8, 14],
        [20, 26, 32]]),
 array([[ 2,  4,  6],
        [ 8, 10, 12]]))
In[7]: np.random.seed(10); e = np.random.rand(2, 3); e
Out[7]:
array([[0.77132064, 0.02075195, 0.63364823],
       [0.74880388, 0.49850701, 0.22479665]])
In[8]: f = e + a - 2*b; f, f.dtype
Out[8]:
(array([[ -3.22867936,  -8.97924805, -13.36635177],
        [-18.25119612, -23.50149299, -28.77520335]]),
 dtype('float64'))
```

6.2.2 函数运算

sum、max 和 min 方法分别返回一个数组包含的所有元素的总和、最大值和最小值。np.sort 函数可对数组进行排序。对于二维数组，可通过 axis 关键字实参指定对每行或每列分别计算。NumPy 提供了很多数学函数 (sin,cos,exp 等)，这些函数可分别作用于数组的每个元素。输入函数名和问号可以获取该函数的详细说明。

Listing 6.7: 函数运算

```
In[1]: g = np.array([[2,6,5],[4,1,3]]); g
Out[1]: array([[2, 6, 5],
```

```
                [4, 1, 3]])
In[2]: g.sum(), g.max(), g.min()
Out[2]:  (21, 6, 1)
In[3]: g.max(axis=0), g.max(axis=1)
Out[3]:  (array([4, 6, 5]), array([6, 4]))
In[4]: g.min(axis=0), g.min(axis=1)
Out[4]: (array([2, 1, 3]), array([2, 1]))
In[5]: np.sort(g)                # sort along the last axis
Out[5]: array([[2, 5, 6],
                [1, 3, 4]])
In[6]: np.sort(g, axis=None)     # sort the flattened array
Out[6]: array([1, 2, 3, 4, 5, 6])
In[7]: np.sort(g, axis=0)        # sort along the first axis
Out[7]: array([[2, 1, 3],
                [4, 6, 5]])
In[8]: np.sort?
Out[8]: Signature: np.sort(a, axis=-1, kind=None, order=None)
Docstring:
Return a sorted copy of an array.

Parameters
----------
a : array_like
    Array to be sorted.
axis : int or None, optional
......
In[9]: np.sqrt(b) + np.exp(a - 5) * np.cos(e**1.3 - f)
Out[9]:
array([[1.40958124, 2.22018241, 2.83965649],
       [3.45077796, 3.87699024, 3.31766545]])
```

6.3 索引、切片和迭代

一维数组可以像列表一样进行索引、切片和迭代。

Listing 6.8: 一维数组的索引、切片和迭代

```
In[1]: a = np.arange(1, 16, 2)**2; a
Out[1]: array([  1,   9,  25,  49,  81, 121, 169, 225],
              dtype=int32)
```

```
In[2]: a[3], a[1:7:2]
Out[2]: (49, array([  9,  49, 121], dtype=int32))
In[3]: a[:6:3] = 361; a
Out[3]: array([361,   9,  25, 361,  81, 121, 169, 225],
              dtype=int32)
In[4]: a[::-1]
Out[4]: array([225, 169, 121,  81, 361,  25,   9, 361],
              dtype=int32)
In[5]: for i in a: print(np.sqrt(i), end='␣')
Out[6]: 19.0 3.0 5.0 19.0 9.0 11.0 13.0 15.0
```

多维数组的每个维度都有一个索引，这些索引用逗号分隔共同构成一个完整的索引元组。如果提供的索引的数量小于维度，则等同于将缺失的维度全部选择 (即冒号“:”)。省略号 (“...”) 代表多个可省略的冒号。将二维数组看成是一个矩阵，对于二维数组的索引和迭代以矩阵的行为单位。对于一个二维数组 a，a.flat 是一个迭代器，可用在 for 循环中以每个元素为单位进行迭代。

数组不仅可以使用整数和切片作为索引，也可以使用整数数组 (或列表) 或布尔值数组 (或列表) 作为索引。使用一个整数数组 a 作为一个数组 b 的索引得到的结果是一个数组 c，c 中的每个元素是以 a 中的每个元素作为索引值所获取的数组 b 中的对应元素。使用一个布尔值数组 a 作为一个数组 b 的索引得到的结果是一个数组 c，c 中仅包含以数组 a 中的值为 True 的元素的索引值所获取的数组 b 中的对应元素。对于多维数组，可分别对每个维度索引。

对于一个二维数组，np.ix_函数使用两个整数数组作为行和列的索引，结果包含了行索引值属于第一个数组并且列索引值属于第二个数组的所有元素 (Out[9])。对于一个二维数组，直接使用两个长度均为 n 的整数数组 a 和 b 作为行和列的索引，结果包含 n 个元素，其中第 i 个元素 ($0 \le i \le n-1$) 的行索引值和列索引值分别为 a[i] 和 b[i](Out[10])。

Listing 6.9: **多维数组的索引、切片和迭代**

```
In[1]: def f(x, y): return x * 4 + y + 1
In[2]: h = np.fromfunction(f, (3, 4), dtype=int); h
Out[2]:
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
In[3]: h[1, 2], h[0, 3], h[2, 2]
Out[3]: (7, 4, 11)
In[4]: h[1:3], h[1:3,], h[1:3,:], h[0]
Out[4]:
(array([[ 5,  6,  7,  8],
```

```
        [ 9, 10, 11, 12]]),
 array([[ 5,  6,  7,  8],
        [ 9, 10, 11, 12]]),
 array([[ 5,  6,  7,  8],
        [ 9, 10, 11, 12]]),
 array([1, 2, 3, 4]))
In[5]: h[:, 1:4:2], h[:, 3:1:-1], h[:, -2]
Out[5]:
(array([[ 2,  4],
        [ 6,  8],
        [10, 12]]),
 array([[ 4,  3],
        [ 8,  7],
        [12, 11]]),
 array([ 3,  7, 11]))
In[6]: for row in h: print(row)
Out[6]:
[1 2 3 4]
[5 6 7 8]
[ 9 10 11 12]
In[7]: for element in h.flat: print(element, end=' ')
Out[7]: 1 2 3 4 5 6 7 8 9 10 11 12
In[8]: h[np.ix_([0,2], [1])]  # 行索引值为0和2并且列索引值为1
Out[8]:
array([[ 2],
       [10]])
In[9]: h[np.ix_([0, 2], [0, 2])] # 行索引值为0和2并且列索引值为0和2
Out[9]:
array([[ 1,  3],
       [ 9, 11]])
In[10]: h[[0, 2], [0, 2]]
# 行索引值为0并且列索引值为0和行索引值为2并且列索引值为2
Out[10]: array([ 1, 11])
In[11]: h[[0, 2]]
Out[11]:
array([[ 1,  2,  3,  4],
       [ 9, 10, 11, 12]])
In[12]: h[:,[0,2]]
Out[12]:
```

```
array([[ 1,  3],
       [ 5,  7],
       [ 9, 11]])
In[13]: h[1:3, 0:3]
Out[13]:
array([[ 5,  6,  7],
       [ 9, 10, 11]])
In[14]: j = np.arange(24).reshape(3, 2, 4); j
Out[14]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],

       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]],

       [[16, 17, 18, 19],
        [20, 21, 22, 23]]])
In[15]: j[2, ...]
Out[15]:
array([[16, 17, 18, 19],
       [20, 21, 22, 23]])
In[16]: j[:, 1:2, :]
Out[16]:
array([[[ 4,  5,  6,  7]],

       [[12, 13, 14, 15]],

       [[20, 21, 22, 23]]])
In[17]: j[..., 1:3]
Out[17]:
array([[[ 1,  2],
        [ 5,  6]],

       [[ 9, 10],
        [13, 14]],

       [[17, 18],
        [21, 22]]])
```

Listing 6.10: 使用整数数组作为索引

```
In[1]: a = np.arange(1, 16, 2)**2; a
Out[1]: array([  1,   9,  25,  49,  81, 121, 169, 225])
In[2]: i = np.array([3, 2, 7, 3, 5]); a[i]
Out[2]: array([ 49,  25, 225,  49, 121], dtype=int32)
In[3]: j = np.array([[3, 2, 4], [1, 5, 6]]); a[j]
Out[3]:
array([[ 49,  25,  81],
       [  9, 121, 169]], dtype=int32)
In[4]: b = a.reshape(4,2); b
Out[4]:
array([[  1,   9],
       [ 25,  49],
       [ 81, 121],
       [169, 225]], dtype=int32)
In[5]: b[np.array([2, 3, 1, 2])]
Out[5]:
array([[ 81, 121],
       [169, 225],
       [ 25,  49],
       [ 81, 121]], dtype=int32)
In[6]: b[np.array([[2, 3], [1, 2]])]
Out[6]:
array([[[ 81, 121],
        [169, 225]],

       [[ 25,  49],
        [ 81, 121]]], dtype=int32)
In[7]: i1 = np.array([[3, 2],  # row indices
                      [2, 1]])
In[8]: i2 = np.array([[0, 1],  # column indices
                      [1, 0]])
In[9]: b[i1, i2]
Out[9]:
array([[169, 121],
       [121,  25]], dtype=int32)
In[10]: b[i1, i2] = 36; a
Out[10]: array([  1,   9,  36,  49,  81,  36,  36, 225],
               dtype=int32)
```

对于一个二维数组，argmax 函数可以返回一个整数数组表示每列 (axis=0) 或每行 (axis=1) 的最大值的索引值。用这个整数数组作为索引可以获取二维数组中每列或每行的最大值。

Listing 6.11: 使用整数数组作为索引

```
In[1]: data = np.cos(np.arange(103, 123)).reshape(5, 4); data
Out[1]:
array([[-0.78223089, -0.94686801, -0.24095905,  0.68648655],
       [ 0.98277958,  0.3755096 , -0.57700218, -0.99902081],
       [-0.50254432,  0.4559691 ,  0.99526664,  0.61952061],
       [-0.32580981, -0.97159219, -0.7240972 ,  0.18912942],
       [ 0.92847132,  0.81418097, -0.04866361, -0.86676709]])
In[2]: maxind0 = data.argmax(axis=0); maxind0
Out[2]: array([1, 4, 2, 0], dtype=int32)
In[3]: data_max0 = data[maxind0, range(data.shape[1])]; data_max0
Out[3]: array([0.98277958, 0.81418097, 0.99526664, 0.68648655])
In[2]: maxind1 = data.argmax(axis=1); maxind1
Out[2]: array([3, 0, 2, 3, 0], dtype=int32)
In[3]: data_max1 = data[range(data.shape[0]), maxind1]; data_max1
Out[3]: array([0.68648655, 0.98277958, 0.99526664, 0.18912942,
               0.92847132])
```

Listing 6.12: 使用布尔值数组作为索引

```
In[1]: a = np.arange(1, 16, 2)**2; a
Out[1]: array([  1,   9,  25,  49,  81, 121, 169, 225],
              dtype=int32)
In[2]: g = a > 50; g
Out[2]: array([False, False, False, False, True, True, True,
               True])
In[3]: a[g] = 0; a
Out[3]:  array([ 1,  9, 25, 49,  0,  0,  0,  0], dtype=int32)
In[4]: b = a.reshape(2, 4); b
Out[4]:
array([[ 1,  9, 25, 49],
       [ 0,  0,  0,  0]], dtype=int32)
In[5]: i1 = np.array([False, True]); b[i1, :]
Out[5]: array([[0, 0, 0, 0]], dtype=int32)
In[6]: i2 = np.array([True, False, False, True]); b[:, i2]
Out[6]:
```

```
array([[ 1, 49],
       [ 0,  0]], dtype=int32)
```

6.4 复制和视图

在对数组进行运算时，有时元素会复制到一个新数组中，有时不发生复制。

Out[4] 行的输出结果是：k is h 的值为 True，并且 k 和 h 的 id 值相同。这说明 In[3] 行的赋值运算给已有数组 h 创建一个别名 k，而不发生元素的复制。Python 语言的函数调用对于可变实参是传引用而非传值，所以也不发生元素的复制。

view 方法从已有数组创建一个新的数组 (Out[6] 行 m is h 的值为 False)，可以理解为原数组的一个视图。新数组和已有数组共享元素 (Out[6] 行 m.base is h 的值为 True 并且 m.flags.owndata 的值为 False)，但可以有不同形状。从原数组切片得到的新数组也是原数组的一个视图。ravel 方法从一个多维数组生成一个一维数组，也是原数组的一个视图。reshape 方法从原数组生成一个不同形状的视图。

copy 方法将已有数组的元素复制到新创建的数组中，新数组和原数组不共享元素 (Out[12] 行 v.base is h 的值为 False 并且 v.flags.owndata 的值为 True)。copy 方法的一个用途是复制元素以后可以用 del 回收原数组占用的内存空间。

Listing 6.13: **复制和视图**

```
In[1]: def f(x, y): return x * 4 + y + 1
In[2]: h = np.fromfunction(f, (3, 4), dtype=int); h
Out[2]:
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
In[3]: k = h
In[4]: k is h, id(k), id(h)
Out[4]: (True, 186428160, 186428160)
In[5]: m = h.view()
In[6]: m is h, m.base is h, m.flags.owndata
Out[6]: (False, True, False)
In[7]: m.resize((2, 6)); h.shape
Out[7]: (3, 4)
In[8]: m[1, 3] = 16; m, h
Out[8]:
(array([[ 1,  2,  3,  4,  5,  6],
```

```
        [ 7,  8,  9, 16, 11, 12]]),
 array([[ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 16, 11, 12]]))
In[9]: t = h[0:2, 1:3]; t
Out[9]:
array([[2, 3],
       [6, 7]])
In[10]: t[1, 0] = 20; h
Out[10]:
array([[ 1,  2,  3,  4],
       [ 5, 20,  7,  8],
       [ 9, 16, 11, 12]])
In[11]: v = h.copy()
In[12]: v is h, v.base is h, v.flags.owndata
Out[12]: (False, False, True)
In[13]: v[1, 1] = 36; v[1, 1], h[1, 1]
Out[13]: (36, 20)
In[14]: p = h.ravel(); p
Out[14]: array([ 1,  2,  3,  4,  5, 20,  7,  8,  9, 16, 11, 12])
In[15]: p[9]=99; h
Out[15]:
array([[ 1,  2,  3,  4],
       [ 5, 20,  7,  8],
       [ 9, 99, 11, 12]])
In[16]: a = np.arange(1000000); b = a[:100].copy()
In[17]: del a  # 回收数组a占用的内存空间
```

6.5 矩阵计算

矩阵可以使用 numpy.array 类表示。SciPy 扩展库的 scipy.linalg 模块定义了常用的矩阵计算函数。*运算符表示两个矩阵的对应元素的乘法，不表示矩阵乘法。inv 函数计算一个矩阵的逆矩阵。det 函数计算一个矩阵的行列式。norm 函数对于一个矩阵 $\mathbf{A}$ 计算其 Frobenius 范数 $\sqrt{trace(\mathbf{A}^H\mathbf{A})}$，对于一个向量 $\mathbf{x}$ 计算其欧式范数 (各分量的平方和的平方根)$\sqrt{\sum_i |x_i|^2}$。
solve($\mathbf{A}$, $\mathbf{b}$) 函数求解线性方程组 $\mathbf{Ax} = \mathbf{b}$。
eig($\mathbf{A}$) 函数返回一个元组，由一个向量和一个矩阵组成。向量的第 i 个分量是矩阵的第 i 个特征值，矩阵的第 i 个列向量则是第 i 个特征值对应的特征向量。

一个秩为 r 的 $m \times n$ 的实矩阵 $\mathbf{A}$ 的奇异值分解 (singular value decomposition) [TL2019] 为 $\mathbf{A} = \mathbf{USV^T} = [\mathbf{u}_1, ..., \mathbf{u}_m]\mathbf{S}[\mathbf{v}_1, ..., \mathbf{v}_n]^T$，其中 $\mathbf{S} = \left(\begin{array}{c|c} diag(\sigma_1, ..., \sigma_r) & \mathbf{0}_{r,n-r} \\ \hline \mathbf{0}_{m-r,r} & \mathbf{0}_{m-r,n-r} \end{array}\right)$，奇异值为 $\sigma_1, ..., \sigma_r$，奇异向量为 $\mathbf{u}_1, ..., \mathbf{u}_m$ 和 $\mathbf{v_1}, ..., \mathbf{v_n}$，且

$$\mathbf{A}\mathbf{v}_i = \sigma_i \mathbf{u}_i, \quad i = 1, ..., r, \quad \mathbf{A}\mathbf{v}_i = 0, \quad i = r+1, ..., n, \tag{6.1}$$

$$\mathbf{A}^T\mathbf{u}_i = \sigma_i \mathbf{v}_i, \quad i = 1, ..., r, \quad \mathbf{A}^T\mathbf{u}_i = 0, \quad i = r+1, ..., m. \tag{6.2}$$

svd($\mathbf{A}$) 返回一个元组 ($\mathbf{U}$, $\mathbf{s}$, $\mathbf{V^T}$)，其中 $\mathbf{s}$ 是由所有奇异值组成的向量。diagsvd 函数返回 。矩阵的秩可通过计数非零奇异值的个数获得。

lu($\mathbf{A}$) 实现了 LU 分解 (LU decomposition)，即将一个矩阵 $\mathbf{A}$ 分解为一个置换矩阵 $\mathbf{P}$、一个对角线上元素均为 1 的下三角矩阵 $\mathbf{L}$ 和一个上三角矩阵 $\mathbf{U}$ 的乘积：$\mathbf{A} = \mathbf{PLU}$。np.allclose 函数判断两个数组在指定的相对误差 (用关键字参数 rtol 指定，默认值为 10^{-5}) 和绝对误差 (用关键字参数 atol 指定，默认值为 10^{-8}) 下是否相等。

scipy.linalg 模块还实现了 Cholesky 分解、QR 分解和 Schur 分解 [SciPyDoc]。

Listing 6.14: **矩阵计算**

```
In[1]: import numpy as np
In[2]: A = np.array([[4,3],[2,1]]); A
Out[3]:
array([[4, 3],
       [2, 1]])
In[3]: from scipy import linalg; linalg.inv(A)
Out[3]:
array([[-0.5,  1.5],
       [ 1. , -2. ]])
In[4]: b = np.array([[6,5]]); b  #  2D array
Out[4]: array([[6, 5]])
In[5]: b.T
Out[5]:
array([[6],
       [5]])
In[6]: A*b   # 并非矩阵乘法
Out[6]:
array([[24, 15],
       [12,  5]])
In[7]: A.dot(b.T) # 矩阵乘法
Out[7]:
```

```
array([[39],
       [17]])
In[8]: b = np.array([6,5]); b
Out[8]: array([6, 5])
In[9]: b.T   # 并非矩阵转置
Out[9]: array([6, 5])
In[10]: A.dot(b)
Out[10]: array([39, 17])
In[11]: A.dot(linalg.inv(A)
Out[11]:  array([[1., 0.],
                 [0., 1.]])
In[12]: linalg.det(A)
Out[12]: -2.0
In[13]: linalg.norm(A), linalg.norm(b)
Out[13]: (5.477225575051661, 7.810249675906654)
In[14]: x = np.linalg.solve(A, b); x
Out[14]: array([ 4.5, -4. ])
In[15]: A.dot(x) - b
Out[15]: array([0., 0.])
In[16]: la, v = linalg.eig(A)
In[17]: la
Out[17]: array([ 5.37228132+0.j, -0.37228132+0.j])
In[18]: v
Out[18]:
array([[ 0.90937671, -0.56576746],
       [ 0.41597356,  0.82456484]])
In[19]: A.dot(v[:, 0]) - la[0] * v[:, 0]
Out[19]: array([0.+0.j, 0.+0.j])
In[20]: np.sum(abs(v**2), axis=0)
Out[20]: array([1., 1.])
In[21]: A = np.array([[2,3,5],[7,9,11]])
In[22]: U,s,V = linalg.svd(A); s
Out[22]: array([16.96707058,  1.05759909])
In[23]: m, n = A.shape; S = linalg.diagsvd(s, m, n); S
Out[23]:
array([[16.96707058,  0.        ,  0.        ],
       [ 0.        ,  1.05759909,  0.        ]])
In[24]: U.dot(S.dot(V))
Out[24]:
```

```
array([[ 2.,  3.,  5.],
       [ 7.,  9., 11.]])
In[25]: tol = 1E-10; (abs(s) > tol).sum()  # 输出矩阵的秩
Out[25]: 2
In[26]: C = np.array([[2,3,5,7],[9,11,13,17],[19,23,29,31]])
In[26]: p, l, u = linalg.lu(C); p, l, u
Out[26]:
(array([[0., 1., 0.],
        [0., 0., 1.],
        [1., 0., 0.]]),
 array([[1.        , 0.        , 0.        ],
        [0.10526316, 1.        , 0.        ],
        [0.47368421, 0.18181818, 1.        ]]),
 array([[19.        , 23.        , 29.        , 31.        ],
        [ 0.        ,  0.57894737,  1.94736842,  3.73684211],
        [ 0.        ,  0.        , -1.09090909,  1.63636364]]))
In[27]: np.allclose(C - p @ l @ u, np.zeros((3, 4)))
Out[27]: True
```

6.6 稀疏矩阵

稀疏矩阵指大多数元素为 0 的矩阵，在求解微分方程等很多问题中经常出现。非稀疏的矩阵也称为稠密矩阵。按照稠密矩阵的方式对稀疏矩阵进行存储和计算会导致不必要的空间和时间开销，因此通常只存储稀疏矩阵中非零元的位置和值。scipy.sparse 模块提供了多个类，分别用不同的格式存储稀疏矩阵。它们都从一个共同的父类 spmatrix 继承，不同的存储格式体现为每个类的独特属性。对于一个 spmatrix 类或其子类的对象 mtx，mtx.A(等同于 mtx.toarray()) 返回其对应的 NumPy 数组，mtx.T(等同于 mtx.transpose()) 和 mtx.H 分别返回其转置和共轭转置，mtx.real 和 mtx.imag 分别返回其实部矩阵和虚部矩阵，mtx.size 返回其非零元的个数，mtx.shape 返回一个由行数和列数组成的元组。

稀疏矩阵的格式包括 CSR(Compressed Sparse Row, 压缩稀疏行)、CSC(Compressed Sparse Column, 压缩稀疏列)、COO(Coordinate list, 坐标列表), LIL(List of lists, 列表的列表), DOK (dictionary of keys, 键的字典), DIA(diagonal matrix, 对角矩阵) 和 BSR(Block-sparse matrix, 块稀疏矩阵)。CSR 和 CSC 适用于矩阵计算，COO, LIL 和 DOK 适用于创建和更新稀疏矩阵，DIA 适用于对角矩阵，BSR 适用于由多个形状相同的稠密子阵构成的稀疏矩阵。6.2 节介绍的对于 Numpy 数组的运算也适用于这些格式的稀疏矩阵。通过索引或切片访问稀疏矩阵的语法和 Numpy 数组相同，但某些格式的稀疏矩阵不提供切片运算或即使提供但运行效率低。不同格式的稀疏矩阵可以通过 toXXX 方法互相转换，例如 tocsr 方法转换成 CSR。

在解决问题时根据需要创建合适格式的稀疏矩阵，在进行计算时再转换成 CSR 或 CSC。程序 6.15 通过示例演示了这些不同格式的稀疏矩阵的用法和属性。直接生成 CSR 格式的方式有两种。In[29] 行通过提供三个数组生成：data 保存了所有非零元；row 保存了所有非零元的行索引值；col 保存了所有非零元的列索引值。关键字实参 shape 指定矩阵的行数和列数。In[33] 行通过提供三个数组生成：data 保存了所有非零元；indices 保存了所有非零元的列索引值；indptr 保存了每行的第一个非零元在 data 中的索引值。

scipy.sparse.linalg 模块提供了多个函数求解线性方程组 $\mathbf{Ax} = \mathbf{b}$，其中 $\mathbf{A}$ 是稀疏矩阵，$\mathbf{x}$ 和 $\mathbf{b}$ 都是稠密向量。spsolve 函数进行直接求解。进行迭代求解的函数有多个，例如 cg(共轭梯度) 和 bicg(双共轭梯度) 等，它们的返回值是一个元组 (x, info)，其中 x 是解，info 为 0 表示没有出错，info 为正数表示收敛错误，info 为负数表示输入错误。当一个稀疏矩阵的行数超过某一阈值 (如 100) 时，使用 scipy.sparse.linalg 模块的函数进行求解的运行时间短于 np.linalg.solve[RJ2019]。

Listing 6.15: **scipy.sparse 模块的稀疏矩阵类**

```
In[1]: import numpy as np
In[2]: import scipy.sparse as sps
In[3]: data = np.arange(12).reshape((3, 4)) + 1; data
Out[3]:
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
In[4]: offsets = np.array([0, 1, -2])
In[5]: dia = sps.dia_matrix((data, offsets), shape=(4, 4)); dia
Out[5]:
<4x4 sparse matrix of type '<class␣'numpy.int32'>'
        with 9 stored elements (3 diagonals) in DIAgonal format>
# 生成了一个DIA格式的稀疏矩阵，data中的每一行被排布在矩阵中的对角线
# 和其上方或下方的平行线上，offsets中的每个元素表示data中的对应行
# 的排布位置：0表示对角线，正整数和负整数分别表示对角线上方和
# 下方某一偏移量的平行线。
In[6]: print(dia.todense())
Out[6]:
[[ 1  6  0  0]
 [ 0  2  7  0]
 [ 9  0  3  8]
 [ 0 10  0  4]]
In[7]: lil = dia.tolil()  # 从一个DIA格式的矩阵生成一个LIL格式的矩阵
In[8]: lil.rows # 由多个列表组成，每个列表存储了每行非零元的列索引值
```

```
Out[8]:
array([list([0, 1]), list([1, 2]), list([0, 2, 3]),
       list([1, 3])], dtype=object)
In[9]: lil.data  # 由多个列表组成，每个列表存储了每行的非零元
Out[9]:
array([list([1, 6]), list([2, 7]), list([9, 3, 8]),
       list([10, 4])], dtype=object)
In[10]: coo = lil.tocoo() # 从一个LIL格式的矩阵生成一个COO格式的矩阵
In[11]: coo.row  # 所有非零元的行索引值
Out[11]: array([0, 0, 1, 1, 2, 2, 2, 3, 3])
In[12]: coo.col  # 所有非零元的列索引值
Out[12]: array([0, 1, 1, 2, 0, 2, 3, 1, 3])
In[13]: coo.data  # 所有的非零元
Out[13]: array([ 1,  6,  2,  7,  9,  3,  8, 10,  4], dtype=int32)
In[14]: dok = coo.todok() # 从一个COO格式的矩阵生成一个DOK格式的矩阵
In[15]: dok.items() # 每个键是一个非零元的位置元组，对应的值是非零元
Out[15]: dict_items([((0, 0), 1), ((2, 0), 9), ((0, 1), 6),
                     ((1, 1), 2), ((3, 1), 10), ((1, 2), 7),
                     ((2, 2), 3), ((2, 3), 8), ((3, 3), 4)])
In[16]: csr = dok.tocsr() # 从一个DOK格式的矩阵生成一个CSR格式的矩阵
In[17]: csr.data  # 所有的非零元
Out[17]: array([ 1,  6,  2,  7,  9,  3,  8, 10,  4], dtype=int32)
In[18]: csr.indices  # 所有非零元的列索引值
Out[18]: array([0, 1, 1, 2, 0, 2, 3, 1, 3], dtype=int32)
In[19]: csr.indptr  # 每行的第一个非零元在csr.data中的索引值
Out[19]: array([0, 2, 4, 7, 9], dtype=int32)
In[20]: csr * np.array([4, 3, 2, 1])  # 稀疏矩阵和一个向量的矩阵乘法
Out[20]: array([22, 20, 50, 34], dtype=int32)
In[21]: import scipy.sparse.linalg as spla
In[22]: b = np.array([4, 2, 1, 3])
In[23]: x = spla.spsolve(csr, b); x  # 直接求解
Out[23]:
array([ 0.64503817,  0.55916031,  0.1259542 , -0.64790076])
In[24]: x = np.linalg.solve(csr.todense(), b); x
Out[24]: array([ 0.64503817,  0.55916031,  0.1259542 ,
                -0.64790076])
In[25]: x = spla.bicg(csr, b); x  # 双共轭梯度迭代求解
Out[25]: (array([ 0.64503817,  0.55916031,  0.1259542 ,
                 -0.64790076]), 0)
```

```
In[26]: row = np.array([0, 0, 1, 2, 2, 2])
In[27]: col = np.array([0, 2, 1, 0, 1, 2])
In[28]: data = np.array([1, 2, 3, 4, 5, 6])
In[29]: print(sps.csr_matrix((data, (row, col)),
                shape=(3, 3)).toarray())
Out[29]: [[1 0 2]
          [0 3 0]
          [4 5 6]]
In[30]: indptr = np.array([0, 2, 3, 6])
In[31]: indices = np.array([0, 2, 1, 0, 1, 2])
In[32]: data = np.array([1, 2, 3, 4, 5, 6])
In[33]: print(sps.csr_matrix((data, indices, indptr),
                shape=(3, 3)).toarray())
Out[33]: [[1 0 2]
          [0 3 0]
          [4 5 6]]
```

scipy.sparse.linalg 模块的 eigs 函数求解稀疏矩阵的特征值和特征向量，eigsh 求解实对称矩阵和复 Hermitian 矩阵的特征值和特征向量。由于稀疏矩阵的规模较大，这些函数只返回指定数量的特征值和特征向量。svds 函数进行奇异值分解。

程序 6.17 的第 7 行调用 diags 函数创建了一个对角矩阵，它的第一个实参是一个存储了一些非零元的列表，第二个实参是一个存储了偏移量的列表，其中的每个元素表示第一个列表中的对应元素所在位置相对于对角线的偏移量，第三个实参是一个表示矩阵形状的元组，第四个实参指定了格式。第 9 行至第 11 行使用索引和切片语法修改了矩阵的一些元素。randint 函数生成一个指定形状的数组，由服从某一区间内的均匀分布的随机整数组成，区间的范围由前两个参数指定，数组的形状由第三个参数指定。图 6.1 的左子图和右子图分别显示了修改前后的稀疏矩阵的结构，其中矩阵的非零元用蓝色小方块表示。第 14 行调用 eigs 函数时提供的关键字实参 k 指定需要返回的特征值的个数，关键字实参 which 指定返回的特征值需要满足的条件：LM 和 SM 分别表示最大和最小幅度 (幅度即复数的模或实数的绝对值), LR 和 SR 分别表示最大和最小实部，LI 和 SI 分别表示最大和最小虚部。第 15 行的输出结果显示返回值 evals 是一个包含 5 个特征值的一维数组，返回值 evecs 是一个 100×5 的二维数组。evecs 的第 j 列是 evals 中第 j 个特征值对应的特征向量，第 17 行至第 19 行根据定义检查了这些特征值和特征向量。

Listing 6.16: **scipy.sparse 模块的稀疏矩阵类**

```
1 import numpy as np
2 import scipy.sparse as sps; import scipy.sparse.linalg as spla
3 import matplotlib.pyplot as plt
4
```

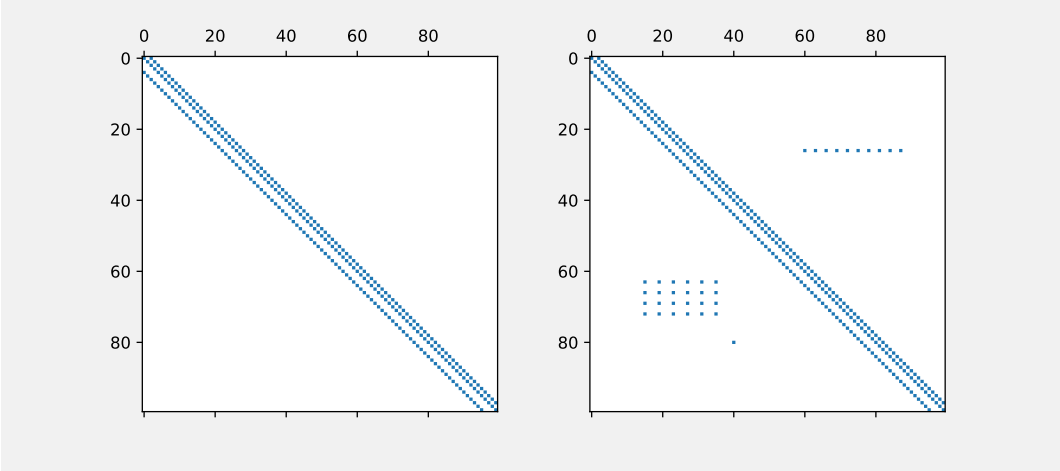


图 6.1: 稀疏矩阵的结构

```
fig, axes = plt.subplots(1, 2, figsize=(9, 4), dpi = 300)
N = 100
m = sps.diags([1, -2, 3], [-4, 0, 2], [N, N], format='dok')
axes[0].spy(m, markersize=1)
m[63:75:3, 15:39:4] = np.random.randint(10, 20, (4, 6))
m[80, 40] = 9
m[26, 60:90:3] = np.random.randint(40, 80, 10)
axes[1].spy(m, markersize=1)

evals, evecs = spla.eigs(m, k=5, which='LM')
print(evals.shape, evecs.shape) # (5,) (100, 5)
t = [np.allclose(m.dot(evecs[:,i]), evals[i] * evecs[:,i])
     for i in range(5)]
print(np.all(t)) # True
```

6.7 实验 6：NumPy 数组和矩阵计算

实验目的

本实验的目的是掌握使用 NumPy 库和 scipy.sparse 模块进行矩阵计算。

提交方式

在 Blackboard 提交一个文本文件 (txt 后缀)，文件中记录每道题的源程序和运行结果。

实验内容

1. 求解线性方程组和矩阵的基本运算

生成一个由实数组成的秩为 4 的 4 行 4 列的矩阵 $\mathbf{A}$ 和一个由实数组成的包含 4 个元素的列向量 $\mathbf{b}$。求解线性方程组 $\mathbf{Ax} = \mathbf{b}$。计算矩阵 $\mathbf{A}$ 的转置、行列式、秩、逆矩阵、特征值和特征向量。

2. 计算最小二乘解和矩阵分解

生成一个由实数组成的秩为 4 的 6 行 4 列的实矩阵 $\mathbf{B}$ 和一个由实数组成的包含 6 个元素的列向量矩阵 $\mathbf{b}$。计算线性方程组 $\mathbf{Bx} = \mathbf{b}$ 的最小二乘解 $(\mathbf{B}^T\mathbf{B})^{-1}\mathbf{B}^T\mathbf{b}$。计算矩阵 $\mathbf{B}$ 的奇异值分解，并验证 6.5 节的奇异值分解满足的方程 6.1。计算矩阵 $\mathbf{B}$ 的 LU 分解。

3. 填写以下程序中所有标注了???之处，使之输出已提供的输出结果。

Listing 6.17: **CSR 格式的稀疏矩阵**

```
import scipy.sparse as sps
import numpy as np
import scipy.sparse.linalg as spla

row = np.array([???])
col = np.array([???])
data = np.array([???])
csr1 = sps.csr_matrix((data, (row, col)), shape=(5, 5))
print(csr1.toarray())
'''
[[1 0 2 0 0]
 [0 0 0 3 0]
 [0 4 0 0 5]
 [0 0 6 0 7]
 [0 0 0 8 9]]
'''

indptr = np.array([???])
indices = np.array([???])
data = np.array([???])
csr2 = sps.csr_matrix((data, indices, indptr), shape=(5, 5))
print(csr2.toarray())
'''
[[1 0 2 0 0]
 [0 0 0 3 0]
```

```
 [0 4 0 0 5]
 [0 0 6 0 7]
 [0 0 0 8 9]]
'''

b = np.array([4, 2, 1, 3, 5])
print(???(csr2, b))  # 求解线性方程组 csr2*x = b
# [ 2.91358025  0.2962963   0.54320988  0.66666667 -0.03703704]
```

第 7 章　错误处理和文件读写

7.1　错误处理

7.1.1　错误的分类

程序发生的错误可分为三大类：语法错误、逻辑错误和运行时错误。

- 语法错误是指程序违反了程序设计语言的语法规则，例如语句“if 3>2 print('3>2')”因冒号缺失导致语法解析器 (parser) 报错“SyntaxError: invalid syntax”。
- 逻辑错误是指程序可以正常运行，但结果不正确。例如程序 7.1 本意是对从 1 至 10 的所有整数求和，但由于对 range 函数的错误理解，实际是从 1 至 9 的所有整数求和。

Listing 7.1: **for 语句输出从 1 至 10 的所有整数的和**

```
sum = 0
for i in range(1, 10):
    sum += i
print("The␣sum␣of␣1␣to␣10␣is␣%d" % sum)
# The sum of 1 to 10 is 45
```

- 运行时错误也称为异常 (exception), 是指程序在运行过程中发生了意外情形而无法继续运行。例如语句“a = 1/0 + 3”在运行过程中报错“ZeroDivisionError: division by zero”并终止。

语法错误和运行时错误都有明确的出错信息，改正这两类错误比较容易。相比之下，改正逻辑错误的难度更大。

避免发生错误的基本方法列举如下。

1. 在编写较复杂程序之前应构思一个设计方案，把要完成的任务分解成为一些子任务，各子任务分别由一个模块完成。每个模块内部根据需要再进行功能分解，实现一些类和函数。这样使得整个程序有清晰合理的结构，容易修改和维护。
2. 对于每个类、函数和模块进行充分的测试。

3. 实现某一功能之前，先了解 Python 标准库和扩展库是否已经实现了该功能。如果有，则可以直接利用。这些库由专业软件开发人员实现，在正确性和运行效率上优于自己编写的程序。

7.1.2 调试

对于比较简单的程序，可以通过反复阅读程序和输出中间步骤的变量值查找逻辑错误。对于比较复杂的程序，上述方法的效率较低，更为有效的方法是设断点调试程序。

设断点调试程序所依据的原理是基于命令式编程范式编写的程序的运行过程可以理解为状态转换的过程。状态包括程序中所有变量的值和正在运行的语句编号。每条语句的运行导致某些变量的值发生变化，可以理解为发生了一步状态转换。程序的输出结果是最终状态。从程序开始运行到结束经历了多次状态转换。如果程序结束时的输出结果有错，则错误必定发生在某一次状态转换中。在可能出错的每一条语句之前设断点。程序运行到断点停下以后，单步运行程序以观察每一步状态转换并与预期结果对照，这样最终一定会找到出错的语句。下面以实现高斯消去法的程序为例说明在 Spyder 中设断点调试程序的方法。

高斯消去法可用来求解线性方程组 $\mathbf{Ax} = \mathbf{b}$。通过一系列的初等行变换，高斯消去法将增广矩阵 $(\mathbf{A}, \mathbf{b})$ 转变成一个上三角矩阵。设矩阵 $\mathbf{A}$ 的行数和列数分别为 m 和 n。程序的设计方案如下：

1. 外循环对第 j 列 $(0 \le j \le n-2)$ 运行，每次循环完成后第 j 列处于主对角线下方的元素变为 0。
   (a) 内循环对第 i 行 $(j+1 \le i \le m-1)$ 运行，将第 j 行乘以 $-a_{ij}/a_{jj}$ 的结果从第 i 行减去，目的是使得 a_{ij} 变为 0。

程序 7.2 的运行结果 (7.3) 显示它对矩阵 $\mathbf{A}$ 输出了正确的结果，但对 $\mathbf{B}$ 输出的结果有错误并且报告在第 7 行出现除数 M[j, j] 为 0 的警告，这个警告提示了出错的可能原因。

Listing 7.2: 高斯消去法第一个版本

```
1  import numpy as np
2
3  def Gaussian_elimination_v1(M):
4      m, n = np.shape(M)
5      for j in range(n - 1):
6          for i in range(j + 1, m):
7              M[i, :] -= (M[i, j] / M[j, j]) * M[j, :]
8      return M
9
10 A = np.array([[2.0,3,5,7],[11,13,17,19],[23,29,31,37]])
11 print(Gaussian_elimination_v1(A))
```

```
12
13 B = np.array([[2.0,3,5,7],[12,18,17,19],[23,29,31,37]])
14 print(Gaussian_elimination_v1(B))
```

Listing 7.3: 程序 **7.2** 的运行结果

```
1  [[  2.          3.          5.          7.        ]
2   [  0.         -3.5       -10.5       -19.5       ]
3   [  0.          0.        -10.        -12.85714286]]
4  [[  2.   3.   5.   7.]
5   [  0.   0. -13. -23.]
6   [ nan  nan -inf -inf]]
7  c:\users\user\.spyder-py3\untitled0.py:7: RuntimeWarning:
8  divide by zero encountered in scalar divide
9    M[i, :] -= (M[i, j] / M[j, j]) * M[j, :]
10 c:\users\user\.spyder-py3\untitled0.py:7: RuntimeWarning:
11 invalid value encountered in multiply
12   M[i, :] -= (M[i, j] / M[j, j]) * M[j, :]
```

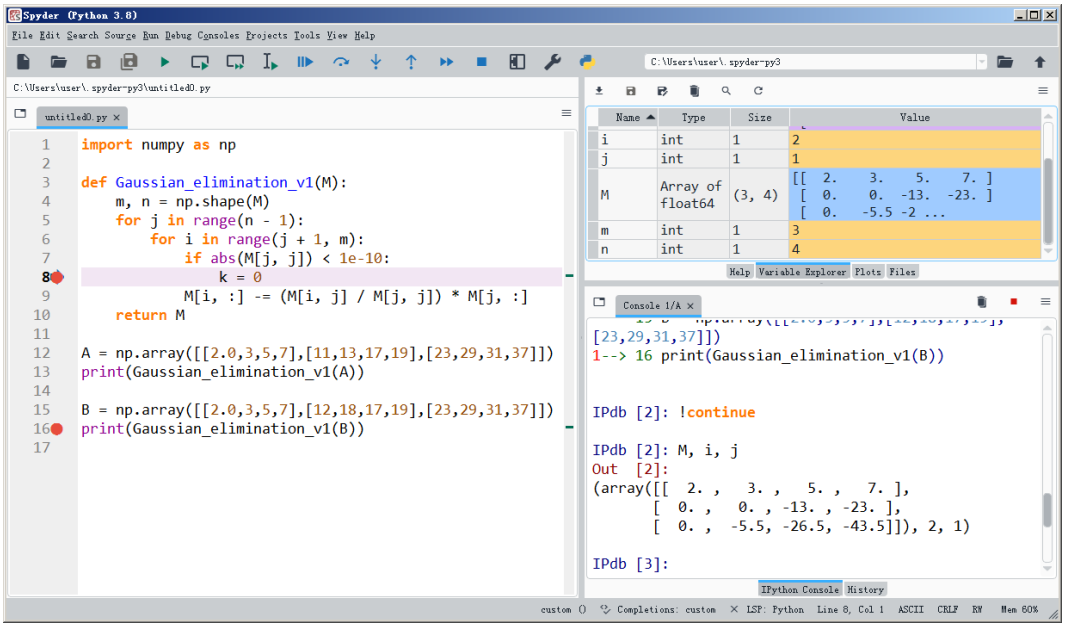


图 7.1: 设置断点进行调试

为了能设置条件断点使程序在 A[j, j] 为 0 时暂停运行，在第 6 行后面加上以下语句

```
if abs(M[j, j]) < 1e-10:
    k = 0
```

这里语句 k = 0 不会影响程序的运行结果。

由于程序只对 **B** 的输出有错，需要先在第 16 行以 **B** 作为实参调用函数的语句处设置断点。在左边编辑窗口中的某一行设置断点的步骤是：首先用鼠标点击该行使得光标在该行跳动，然后点击 Debug 菜单的菜单项“Set/Clear breakpoint”或按下快捷键 F12。此时该行的行号的右边出现了一个红色的圆点，表示已在这一行设置断点。设置断点的操作类似电灯的开关，再进行一次以上操作则会取消断点。

在第 16 行设置断点以后，需要进行调试运行。和正常运行不同，调试运行会在当前设置的所有断点处停下，以便检查程序的中间状态。Debug 菜单提供了多个菜单项用于调试运行，表 7.1 说明了一些常用菜单项的用途。

菜单项	用途
Set/Clear breakpoint	点击一次在当前行设置断点，再点击一次则清除断点 。
Debug	开始调试运行。调试运行和普通运行的区别在于遇到断点会停止。
Step	单步运行，如果当前语句是函数调用语句则执行完函数调用，不会进入函数内部。
Step Into	单步运行，如果当前语句是函数调用语句则进入函数内部，并停止在第一条语句。
Continue	继续运行直至遇到下一个断点，然后停止。
Stop	结束调试运行。
Clear breakpoints in all files	清除所有断点。

表 7.1: Debug 菜单的常用菜单项

点击 Debug 菜单的菜单项“Debug”或使用快捷键 Ctrl+F5 使程序开始调试运行。此时第 16 行行号的右边出现一个蓝色箭头 (部分被红色圆点遮挡)，表示已在这一行停下。点击右上角窗口下边界处的“Variable explorer”可使右上角窗口自动显示所有变量的值。有时矩阵无法完整显示，可以用鼠标右键点击窗口，在弹出的菜单中选中“Resize rows to contents”或“Resize columns to contents”。

下一步需要在第 8 行设置断点。为了使程序继续运行直至下一个断点 (即第 8 行)，可以进行多次单步运行。进行单步运行的方法是点击 Debug 菜单的菜单项“Step Into”或使用快捷键 Ctrl+F11。效率更高的方式是点击 Debug 菜单的菜单项“Continue”或使用快捷键 Ctrl+F5，这样程序可以继续运行直至下一个断点。程序停下后，为了观察程序中变量的值，在控制台输入“M, i, j”，输出结果如图 7.1 所示。由于 M[1, 1] 的值为 0，接下来在运行第 9 行时会发生除数 A[j, j] 为 0 的情形。为了避免出现某行在主对象线上的元素为 0 的情形，可以将该行与其他行进行交换，使得主对象线上的元素的绝对值尽可能大。已经找到出错原因后，为了终止调试运行，点击 Debug 菜单的菜单项“Stop”或使用快捷键 Ctrl+Shift+F12。

程序 7.4 实现了以上修改。第 6 行在位于第 j 列的第 j 行及以下的元素中找到绝对值最大的元素所在行的索引值 p，这个索引值是相对于 j 的。第 7 行判断如果 p 大于 0，则说明绝对值最大的元素不在第 j 行，需要在第 8 行交换第 j 行和第 p+j 行。第 9 行判断若 M[j, j] 为 0 则退出外循环，否则运行内循环。

Listing 7.4: 高斯消去法第二个版本

```
1  import numpy as np
2
3  def Gaussian_elimination_v2(M, tol = 1e-10):
4      m, n = np.shape(M)
5      for j in range(n - 1):
6          p = np.argmax(abs(M[j:m, j]))
7          if p > 0:
8              M[[j, p + j]] = M[[p + j, j]]
9          if abs(M[j, j]) < tol: break
10         for i in range(j + 1, m):
11             M[i, :] -= (M[i, j] / M[j, j]) * M[j, :]
12     return M
13
14 B = np.array([[2.0,3,5,7],[12,18,17,19],[23,29,31,37]])
15 print(Gaussian_elimination_v2(B))
```

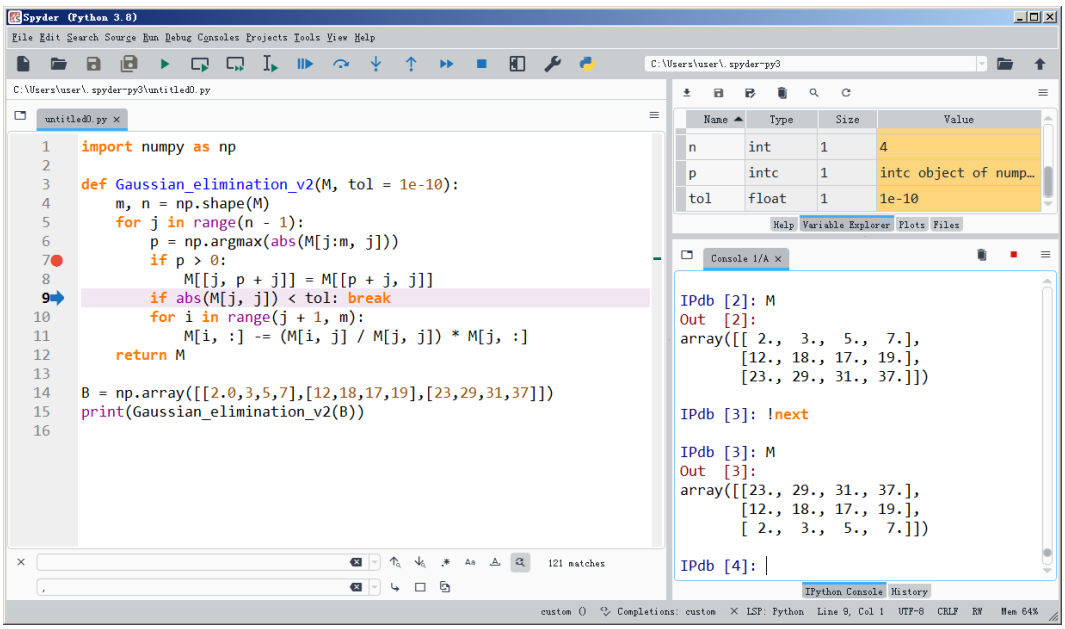


图 7.2: 设置断点进行调试

图 7.2 显示了程序 7.4 在单步运行时输出的中间结果。

7.1.3 异常处理

若程序中某一语句块在运行过程中可能发生运行时错误，可以利用 if 语句对每一种出错情形进行判断和处理。当出错情形较多时，这些 if 语句导致程序结构不清晰并难于理解。异常处理是比 if 语句更好的错误处理方式，体现在将程序的主线和错误处理分离。异常处理为每种出错的情形定义一种异常，然后将可能出错的语句置于 try 语句块中。如果这些语句在运行时出错，运行时系统会抛出异常，导致程序跳转到对应这种异常的 except 语句块中处理异常。else 语句块是可选的，包含不发生任何异常时必须运行的语句，必须位于所有 except 语句块之后。

例如程序 7.5 要求用户在命令行输入两个整数作为参数，然后计算它们的最大公约数。如果用户输入的参数个数少于两个，或者某个参数不是整数，都会导致错误。sys.argv 是一个列表，存储了用户在命令行输入的所有字符串。索引值为 0 的字符串是程序的名称，其余字符串是用户输入的参数。如果输入的参数个数少于两个，则读取 sys.argv[2] 导致 IndexError，第 13 行至第 14 行的 except 语句块对这种异常进行处理，即输出具体的出错信息。如果某个参数不是整数，则 int 函数报错 ValueError，第 15 行至第 16 行的 except 语句块对这种异常进行处理。如果用户输入了至少两个参数，并且前两个都是整数，则程序不会发生异常，第 12 行运行完成以后跳转到第 17 行至第 19 行的 else 语句块。else 语句块调用 gcd 函数计算前两个整数的最大公约数，然后输出。

Listing 7.5: **异常处理**

```
1  def gcd(a, b):
2      while a != b:
3          if a > b:
4              a -= b
5          else:
6              b -= a
7      return a
8
9  import sys
10 try:
11     x = int(sys.argv[1])
12     y = int(sys.argv[2])
13 except IndexError:
14     print('Two␣arguments␣must␣be␣supplied␣on␣the␣command␣line')
15 except ValueError:
16     print('Each␣argument␣should␣be␣an␣integer.')
17 else:
18     print('The␣greatest␣common␣divisor␣of␣%d␣and␣%d␣is␣%d' %\
```

```
            (x, y, gcd(x, y)))
```

Listing 7.6: **程序 7.5 的运行结果**

```
In[1]: run d:\python\src\gcd_ex.py 4
Out[1]: Two arguments must be supplied on the command line
In[2]: run d:\python\src\gcd_ex.py 4 6o
Out[2]: Each argument should be an integer.
In[3]: run d:\python\src\gcd_ex.py 4 60
Out[3]: The greatest common divisor of 4 and 60 is 4
```

Python 标准库定义了很多内置异常类，它们都是 Exception 类的直接或间接子类，构成一个继承层次结构。这些异常类针对的错误类型包括算术运算、断言、输入输出和操作系统等。except 语句块中声明某一个异常类时，可以处理对应于该异常类或其子类的运行时错误。

用户在程序中可以使用这些内置异常类，也可以根据需要自定义异常类，自定义的异常类以 Exception 作为父类。例如程序 7.7 的第 1 行至第 7 行针对用户输入的整数为负数的出错情形定义了异常类 InputRangeError。InputRangeError 类只有一个属性，即出错信息。gcd 函数在第 10 行判断用户输入的两个整数中是否存在负数，若是则在第 11 行抛出 InputRangeError 异常，因为这种情形下 while 循环不会终止。异常导致 gcd 函数返回，该异常对象被第 29 行的 except 语句块捕获，然后在第 30 行输出出错信息。如果用户输入了至少两个参数，并且前两个都是正整数，则程序不会发生异常。

第 31 行至第 32 行的 finally 语句块是可选的。finally 语句块必须位于所有其他语句块之后。无论是否发生异常，finally 语句块都会运行，通常用于回收系统资源等善后工作。finally 语句块的语义规则如下：

1. 如果 try 语句块在运行过程中抛出了异常，且未被任何 except 语句块处理，则运行 finally 语句块后会重新抛出该异常。
2. 如果 except 语句块和 else 语句块在运行过程中抛出了异常，则运行 finally 语句块后会重新抛出该异常。
3. 如果 try 语句块中即将运行 break、continue 或 return 等跳转语句，则会先运行 finally 语句块再运行跳转语句。

Listing 7.7: **自定义异常类**

```
class InputRangeError(Exception):
    """Raised when an input is not in suitable range
       Attributes:
           message -- explanation of suitable range
```

```
    """
    def __init__(self, message):
        self.message = message

def gcd(a, b):
    if a <= 0 or b <= 0:
        raise InputRangeError('Each␣integer␣should␣be␣positive')
    while a != b:
        if a > b:
            a -= b
        else:
            b -= a
    return a

import sys
try:
    x = int(sys.argv[1])
    y = int(sys.argv[2])
    print('The␣greatest␣common␣divisor␣of␣%d␣and␣%d␣is␣%d' %\
            (x, y, gcd(x, y)))
except IndexError:
    print('Two␣arguments␣must␣be␣supplied␣on␣the␣command␣line')
except ValueError:
    print('Each␣argument␣should␣be␣an␣integer.')
except InputRangeError as ex:
    print(ex.message)
finally:
    print("executing␣finally␣clause")
```

Listing 7.8: 程序 **7.7** 的运行结果

```
In[1]: run d:\python\src\gcd_ex.py -48 126
Out[1]: Each integer should be positive
        executing finally clause
In[2]: run d:\python\src\gcd_ex.py 48 126
Out[2]: The greatest common divisor of 48 and 126 is 6
        executing finally clause
```

7.2 文件读写

7.2.1 打开和关闭文件

如果程序需要输入大量数据，应从文件中读取。如果程序需要输出大量数据，应写入文件中。文件可分为两类：文本文件和二进制文件。文本文件存储采用特定编码方式 (例如 UTF-8、GBK 等) 编码的文字信息，以字符作为基本组成单位，可在文本编辑器中显示内容。二进制文件存储图片、视频、音频、可执行程序或其他格式的数据，以字节作为基本组成单位，在文本编辑器中显示为乱码。

读写文件之前，先要使用“f = open(filename, mode)”语句打开文件名为 filename 的文件并创建文件对象 f。其中 mode 是打开方式，可以是'r'(读)、'w'(写) 或'a'(追加)。mode 中如果有 'b' 表示以二进制方式打开。读写一个文件 f 完成以后，需要使用“f.close() ”语句关闭文件。使用
“with open(filename, mode) as f:”语句块打开的文件 f 会在语句块运行结束时自动关闭。

7.2.2 读写文本文件

从普通文本文件读取数据的基本方法是分析文件中的数据格式，采用合适的方法提取有效数据。文件 rainfall.dat(7.9) 记录了合肥市每月的平均降水量。需要从中读取这些数据，然后计算最大值、最小值和平均值并写入文件 rainfall_stat.dat 中。文件中的有效数据在第 2 行至第 13 行，每行的格式是“月份名称+空格+降水量”。

Listing 7.9: **文本文件 rainfall.dat**

```
1  Average rainfall (in mm) in HEFEI: 459 months between 1951 and 1990
2  Jan  32.2
3  Feb  53.2
4  Mar  71.8
5  Apr  92.5
6  May  101.5
7  Jun  117.3
8  Jul  175.7
9  Aug  117.7
10 Sep  85.6
11 Oct  60.7
12 Nov  51.2
13 Dec  27.6
14 Year 988.7
```

程序 7.10 的前 9 行定义了一个函数 extract_data，它的形参为文件名。第 3 行读取文件的第 1 行。第 4 行定义了一个空的字典 rainfall。第 5 行至第 8 行的循环从文件的第 2 行开始每次

读取一行。第 6 行判断当前行是否包含子串'Year'。若存在，则已读完所有月份的数据，可以跳出循环。第 7 行将当前行以空格作为分隔符分解成为两部分 (月份和对应的降水量)，然后存储在一个列表 words 中。第 8 行将从月份到降水量的映射添加到 rainfall 中。

主程序的第 12 行调用函数 extract_data 获取从文件中提取的字典。第 14 行至第 20 行的循环计算最大值、最小值和平均值，并保存最大值和最小值的对应月份。第 22 行至第 25 行输出以上信息到文件 rainfall_stat.dat(7.11) 中。

Listing 7.10: **读写文本文件**

```
1  def extract_data(filename):
2      with open(filename, 'r') as infile:
3          infile.readline()
4          rainfall = {}
5          for line in infile:
6              if line.find('Year') >= 0: break
7              words = line.split()
8              rainfall[words[0]] = float(words[1])
9      return rainfall
10
11 import sys; min = sys.float_info.max; max = -min
12 rainfall = extract_data('D:/Python/src/rainfall.dat')
13 sum = 0
14 for month in rainfall.keys():
15     rainfall_month = rainfall[month]
16     sum += rainfall_month
17     if max < rainfall_month:
18         max = rainfall_month; max_month = month
19     if min > rainfall_month:
20         min = rainfall_month; min_month = month
21
22 with open('D:/Python/src/rainfall_stat.dat', 'w') as outfile:
23     outfile.write('The␣maximum␣rainfall␣of␣%.1f␣occurs␣in␣%s\n' %\
24                   (max, max_month))
25     outfile.write('The␣minimum␣rainfall␣of␣%.1f␣occurs␣in␣%s\n' %\
26                   (min, min_month))
27     outfile.write('The␣average␣rainfall␣is␣%.1f' % (sum / 12))
```

Listing 7.11: **rainfall_stat.dat**

```
1  The maximum rainfall of 175.7 occurs in Jul
```

```
The minimum rainfall of 27.6 occurs in Dec
The average rainfall is 82.3
```

7.2.3 读写 CSV 文件

CSV 是一种简单的电子表格文件格式，其中的数据值之间用逗号分隔。办公软件 (如 Excel 和 LibreOffice Calc 等) 可以读入 CSV 文件并显示为电子表格。Python 标准库的 csv 模块可将 CSV 文件中的数据读入一个嵌套列表中，也可以将一个嵌套列表写入 CSV 文件中。

程序 7.12 的第 2 行至第 3 行从文件 scores.csv(7.13) 中读取四位学生在三个科目上的考试成绩数据并存入嵌套列表 table 中。其中的每个元素都是字符串。第 7 行至第 9 行的双重循环将每个元素转换为 float 类型。第 12 行至第 16 行的双重循环计算每位学生的总分并将其追加到列表中对应这个学生的行。第 18 行创建一个新的列表 row。第 19 行至第 23 行的双重循环计算每个科目的平均成绩并将其追加到 row 中。第 24 行将 row 追加到 table 中。第 26 行使用 pprint 模块的 pprint 函数输出 table。第 28 行至第 31 行将 table 写入文件 scores2.csv(7.14) 中。

Listing 7.12: **读写 CSV 文件**

```
1  import csv, pprint
2  with open('D:/Python/src/scores.csv', 'r') as infile:
3      table = [row for row in csv.reader(infile)]
4
5  nrow = len(table); ncol = len(table[0])
6  for r in range(1, nrow):
7      for c in range(1, ncol):
8          table[r][c] = float(table[r][c])
9
10 table[0].append('Total')
11 for r in range(1, nrow):
12     total = 0
13     for c in range(1, ncol):
14         total += table[r][c]
15     table[r].append(total)
16
17 row = ['Average']
18 for c in range(1, ncol):
19     avg = 0
20     for r in range(1, nrow):
21         avg += table[r][c]
```

```
22     row.append(avg / (nrow - 1))
23 table.append(row)
24
25 pprint.pprint(table)
26
27 with open('D:/Python/src/scores2.csv', 'w', newline='') as outfile:
28     writer = csv.writer(outfile)
29     for row in table:
30         writer.writerow(row)
```

Listing 7.13: **scores.csv**

```
1 Name,Math,Physics,English
2 Tom,95,91,81
3 Jerry,89,82,86
4 Mary,83,80,96
5 Betty,88,96,93
```

Listing 7.14: **scores2.csv**

```
1 Name,Math,Physics,English,Total
2 Tom,95.0,91.0,81.0,267.0
3 Jerry,89.0,82.0,86.0,257.0
4 Mary,83.0,80.0,96.0,259.0
5 Betty,88.0,96.0,93.0,277.0
6 Average,88.75,87.25,89.0
```

7.2.4 读写 JSON 文件

JSON 是“JavaScript Object Notation”的缩写，是一种常用的应用程序间数据交换格式。Python 标准库的 json 模块可将结构化数据 (字典、列表或它们的组合) 转换成为一个 JSON 格式的字符串并写入一个文件中，也可以从一个 JSON 文件中读取结构化数据。

程序 7.15 的第 3 行至第 7 行定义了一组通讯录数据 contacts，它是一个字典的列表。第 9 行至第 10 行打开一个文件 contacts.json，并将 contacts 的内容以 JSON 格式写入其中 (7.16)。第 12 行至第 13 行从 contacts.json 中读取数据，然后由第 15 行输出 (7.17)。

Listing 7.15: **读写 JSON 文件**

```
1 import json, pprint
2
3 contacts = [
4     {"Name":"Tom", "Phone":12345, "Address":"100 Wall St."},
5     {"Name":"Jerry", "Phone":54321, "Address":"200 Main St."},
6     {"Name":"Mary", "Phone":23415, "Address":"300 Fifth Ave."}
7 ]
8
9 with open('D:/Python/src/contacts.json', 'w') as outfile:
```

```
10     json.dump(contacts, outfile)
11
12 with open('D:/Python/src/contacts.json', 'r') as infile:
13     x = json.load(infile)
14
15 pprint.pprint(x)
```

Listing 7.16: **contacts.json**

```
1 [{"Name": "Tom", "Phone": 12345, "Address": "100␣Wall␣St."},
2  {"Name": "Jerry", "Phone": 54321, "Address": "200␣Main␣St."},
3  {"Name": "Mary", "Phone": 23415, "Address": "300␣Fifth␣Ave."}]
```

Listing 7.17: **程序 7.15 的输出结果**

```
1 [{'Address': '100␣Wall␣St.', 'Name': 'Tom', 'Phone': 12345},
2  {'Address': '200␣Main␣St.', 'Name': 'Jerry', 'Phone': 54321},
3  {'Address': '300␣Fifth␣Ave.', 'Name': 'Mary', 'Phone': 23415}]
```

7.2.5 读写 pickle 文件

pickle 是一种 Python 定义的数据格式。Python 标准库的 pickle 模块可将结构化数据 (字典、列表或它们的组合以及类的对象) 转换成为一个字节流并写入一个二进制文件中，也可以从一个 pickle 文件中读取结构化数据。JSON 格式适用于使用多种程序设计语言编写的程序之间的数据交换，而 pickle 格式只适用于使用 Python 语言编写的程序之间的数据交换。

程序 7.18 的第 3 行至第 7 行定义了一组通讯录数据 contacts，它是一个字典的列表。第 9 行至第 10 行打开一个文件 contacts.pickle，并将 contacts 的内容以 pickle 格式写入其中。第 12 行至第 13 行从 contacts.pickle 中读取数据，然后由第 15 行输出 (7.19)。

Listing 7.18: **读写 pickle 文件**

```
1 import pickle, pprint
2
3 contacts = [
4     {"Name":"Tom", "Phone":12345, "Address":"100␣Wall␣St."},
5     {"Name":"Jerry", "Phone":54321, "Address":"200␣Main␣St."},
6     {"Name":"Mary", "Phone":23415, "Address":"300␣Fifth␣Ave."}
7     ]
8
9 with open('D:/Python/src/contacts.pickle', 'wb') as outfile:
10     pickle.dump(contacts, outfile)
```

```

with open('D:/Python/src/contacts.pickle', 'rb') as infile:
    x = pickle.load(infile)

pprint.pprint(x)
```

Listing 7.19: **程序 7.18 的输出结果**

```
[{'Address': '100␣Wall␣St.', 'Name': 'Tom', 'Phone': 12345},
 {'Address': '200␣Main␣St.', 'Name': 'Jerry', 'Phone': 54321},
 {'Address': '300␣Fifth␣Ave.', 'Name': 'Mary', 'Phone': 23415}]
```

7.2.6 读写 NumPy 数组的文件

np.savetxt 函数可以把一个 NumPy 数组保存为一个文本文件。np.loadtxt 函数可以从一个文本文件中读入一个数组。

np.save 函数可以把一个数组保存为一个后缀为“npy”的二进制文件。np.load 函数可以从一个后缀为“npy”的二进制文件中读入一个数组。

np.savez 函数可以把多个数组保存为一个后缀为“npz”的二进制文件。np.savez_compressed 可以把多个数组保存为一个后缀为“npz”的压缩二进制文件。

Listing 7.20: **读写 NumPy 数组的文件**

```
In[1]: import numpy as np; a = np.arange(1, 16, 2)**2; a
Out[1]: array([  1,   9,  25,  49,  81, 121, 169, 225],
              dtype=int32)
In[2]: b = a.reshape(2, 4); b
Out[2]:
array([[  1,   9,  25,  49],
       [ 81, 121, 169, 225]], dtype=int32)
In[3]: np.savetxt('D:/Python/dat/b.txt', b)
In[4]: c = np.loadtxt('D:/Python/dat/b.txt'); c
Out[4]:
array([[  1.,   9.,  25.,  49.],
       [ 81., 121., 169., 225.]])
In[5]: np.save('D:/Python/dat/b.npy', b)
In[6]: c = np.load('D:/Python/dat/b.npy'); c
Out[6]:
```

```
array([[  1,   9,  25,  49],
       [ 81, 121, 169, 225]])
In[7]: np.savez('D:/Python/dat/ab.npz', a, b)
In[8]: cd = np.load('D:/Python/dat/ab.npz')
In[9]: c = cd['arr_0']; c
Out[9]: array([  1,   9,  25,  49,  81, 121, 169, 225])
In[10]: d = cd['arr_1']; d
Out[10]:
array([[  1,   9,  25,  49],
       [ 81, 121, 169, 225]])
```

7.3 实验 7：错误处理和文件读写

实验目的

本实验的目的是掌握以下内容：程序调试，异常处理和文件读写。

提交方式

在 Blackboard 提交一个文本文件 (txt 后缀)，文件中记录第 2 题和第 3 题的源程序和运行结果。第 1 题无须提交。

实验内容

1. 程序调试
设断点单步运行本章的程序 7.4、7.10 和 7.12，观察变量的值。

2. 异常处理
编写程序读入用户在命令行输入的三个 float 类型的数值，判断它们是否能构成一个三角形的三条边。若可以，则使用以下公式计算并输出三角形的面积。公式中的 a, b, c 为三角形的三条边的长度。

$$area = \sqrt{s(s-a)(s-b)(s-c)},\ s = \frac{a+b+c}{2}$$

程序应处理以下类型的错误并输出出错信息：

- 用户在命令行输入的参数小于三个；
- 用户在命令行输入的三个参数不全是 float 类型；
- 用户在命令行输入的三个 float 类型的数值不能构成一个三角形的三条边，这里需要自定义异常类 InvalidTriangleError。

3. 文件读写

编写程序读入一个存储了程序 7.20 的文本文件，从中提取用户输入的代码然后输出到一个文本文件中。输出的文件的内容应为：

Listing 7.21: **程序 7.20 中用户输入的代码**

```
import numpy as np; a = np.arange(1, 16, 2)**2; a
b = a.reshape(2, 4); b
np.savetxt('D:/Python/dat/b.txt', b)
c = np.loadtxt('D:/Python/dat/b.txt'); c
np.save('D:/Python/dat/b.npy', b)
c = np.load('D:/Python/dat/b.npy'); c
np.savez('D:/Python/dat/ab.npz', a, b)
......
```

第 8 章　程序运行时间的分析和测量

一个用高级程序设计语言 (例如 Python) 编写的程序的运行时间取决于多种因素，例如求解问题的算法、问题的规模、输入数据的特点、编译器的代码生成和优化、运行时系统的效率和 CPU 执行指令的速度等。求解一个问题的算法可以有多种，为了缩短程序的运行时间，应选择时间性能最好的算法。通过测量一个程序中的各函数和语句的运行时间，可以找到程序在运行时间上的瓶颈部分，针对瓶颈部分进行改进 (例如改写为 Cython 扩展模块) 也可以缩短程序的运行时间。

8.1　算法和时间性能的分析

算法是求解问题的一系列计算步骤，用来将输入数据转换成输出结果。算法设计与分析是计算机科学的核心领域。算法的每个步骤应当是精确定义的，不允许出现歧义。算法应在运行有穷步后结束。如果一个算法对所有输入数据都能输出正确的结果并停止，则称它是正确的。3.2 节列出的求解某一给定自然数区间内的所有质数的设计方案描述了一个算法。评价算法优劣的一个重要指标是时间性能，即在给定的问题规模下运行算法所消耗的时间。

程序是使用某种程序设计语言对一个算法的实现。算法的时间性能是决定程序的运行时间的关键因素。时间性能的分析对算法在计算机上的运行过程进行各种简化和抽象，把算法的运行时间定义为问题规模的函数，称为时间复杂度。当问题规模增长时，时间复杂度也会增长。分析的主要结果是时间复杂度的增长阶 (order of growth)，即时间复杂度的增长速度有多快。当问题规模充分大时，增长阶决定了算法的时间性能。有些算法的运行时间受问题输入数据的影响很大，例如排序算法。此时需要对最坏、平均和最好三种情形进行分析。这里只对最坏情形进行分析，即估计时间复杂度的上界。

时间复杂度通常使用三种记号描述。O 记号的定义如下：设函数 $f(n)$ 和 $g(n)$ 是定义在非负整数集合上的正函数, 如果存在两个正常数 c 和 n_0, 使得当 $n \geq n_0$ 时 $f(n) \leq cg(n)$ 成立，则记为 $f(n) = O(g(n))$。O 记号的含义是：当 n 增长到充分大以后，$g(n)$ 是 $f(n)$ 的一个上界。例如：$n^2 + 10n = O(n^3)$；$n^{100} + 100n^{99} = O(1.01^n)$。和 O 记号对称的是 Ω 记号：$f(n) = \Omega(g(n))$ 当且仅当 $g(n) = O(f(n))$。第三种记号是 Θ 记号：$f(n) = \Theta(g(n))$ 当且仅当 $f(n) = O(g(n))$ 并且 $g(n) = O(f(n))$。例如：$10n^2 + 1000n = \Theta(n^2)$；$0.01n^3 + 9n^2 = \Theta(n^3)$。这些记号提供

了一种抽象，即只关注一个时间复杂度函数的表达式中增长阶最高的项并且忽略它的常数系数，该项的增长阶就是时间复杂度的增长阶。

8.2 算法的时间复杂度

算法由一些不同类别的的基本运算组成，包括算术运算、关系运算、逻辑运算、数组 (列表) 元素的访问和流程控制等。这些基本运算的运行时间都是常数。算法的时间复杂度等于每种基本运算的运行时间和其对应运行次数的乘积的总和，其中运行次数是问题规模的函数。

为了简化分析，一般只考虑运行次数最多的基本运算，因为当问题规模较大时它们的运行时间是时间复杂度中增长阶最高的项。对于单层循环或嵌套的多层循环，运行次数最多的基本运算位于最内层循环。对于问题规模 n，用 $f(n)$ 表示最内层循环的运行次数。设最内层循环包含了 k 种基本运算，其中第 i 种基本运算在最内层循环里出现的次数和运行时间分别是 n_i 和 $c_i(i=1,...,k)$。这些基本运算的总计运行时间是 $f(n)\sum_{i=1}^{k} n_i c_i$。由于 $\sum_{i=1}^{k} n_i c_i$ 是常数，$f(n)\sum_{i=1}^{k} n_i c_i$ 的增长阶和 $f(n)$ 的增长阶相同。因此，循环的时间复杂度的增长阶由最内层循环的运行次数决定。

对于由递归结构构成的算法，根据递归公式可以得到时间复杂度满足的递归方程 $T(n)=aT(n/b)+f(n)$，其中 $T(n)$ 表示求解规模为非负整数 n 的原问题的时间复杂度。原问题被分解成 a 个规模为 n/b 的与原问题结构类似的子问题，其中 $a \geq 1$ 和 $b>1$ 是常数，n/b 等于 $\lfloor n/b \rfloor$ 或 $\lceil n/b \rceil$。原问题的解可由这些子问题的解合并而成，合并过程的时间复杂度是一个函数 $f(n)$。该方程可由 Master 定理 [CL2009] 求解。

- 若 $f(n)=O(n^{\log_b a-\epsilon})$ 对于常数 $\epsilon>0$ 成立，则 $T(n)=\Theta(n^{\log_b a})$；
- 若 $f(n)=\Theta(n^{\log_b a})$ 成立，则 $T(n)=\Theta(n^{\log_b a}\log_2 n)$；
- 若 $f(n)=\Omega(n^{\log_b a+\epsilon})$ 对于常数 $\epsilon>0$ 成立，并且当 n 充分大时 $af(n/b)\leq cf(n)$ 对于常数 $c<1$ 成立，则 $T(n)=\Theta(f(n))$。

大多数算法的时间复杂度属于以下七类，按照增长阶从低到高的次序依次为：$O(1)$，$O(\log n)$，$O(n)$，$O(n\log n)$，$O(n^2)$，$O(n^3)$ 和 $O(a^n)(a>1)$。一般认为增长阶为 $O(n^k)$(k 是一个常数) 的算法是可行的。增长阶为 $O(a^n)(a>1)$ 的算法只适用于规模较小的问题。解决一个问题的算法通常不止一种，在保证正确性的前提下应尽量选择时间复杂度的增长阶最低的算法。以下分析一些常用算法的时间复杂度。

8.2.1 插入排序

程序 8.1 实现了从小到大排序的插入排序算法。插入排序的基本思想是将待排序列表中的每个元素依次插入到合适的位置。第 3 行的外循环的循环变量 i 是列表中待插入元素的索引值。第 6 行至第 8 行的内循环将待插入元素 value 依次与索引值为 i-1,i-2,...,0 的元素进行比较，将比 value 大的元素向后移动，直至找到比 value 小的元素或者 pos=0 为止。第 9 行将 value 写

入索引值为 pos 的位置，完成插入。对于长度为 n 的列表 s，内层循环在最坏情况下 (待排序列表是从大到小的顺序) 的运行次数为 $1+2+...+(n-1)=n(n-1)/2$。因此插入排序算法的时间复杂度是 $O(n^2)$。

Listing 8.1: **插入排序**

```
def insertion_sort(s):
    n = len(s)
    for i in range(1, n):
        value = s[i]; print('insert␣%2d:␣' % value, end = '␣')
        pos = i
        while pos > 0 and value < s[pos - 1] :
            s[pos] = s[pos - 1]
            pos -= 1
        s[pos] = value
        print(s)

s = [21, 73, 6, 67, 99, 60, 77, 5, 51, 32]; print(s)
insertion_sort(s)
```

Listing 8.2: **插入排序的运行过程**

```
[21, 73, 6, 67, 99, 60, 77, 5, 51, 32]
insert 73:  [21, 73, 6, 67, 99, 60, 77, 5, 51, 32]
insert  6:  [6, 21, 73, 67, 99, 60, 77, 5, 51, 32]
insert 67:  [6, 21, 67, 73, 99, 60, 77, 5, 51, 32]
insert 99:  [6, 21, 67, 73, 99, 60, 77, 5, 51, 32]
insert 60:  [6, 21, 60, 67, 73, 99, 77, 5, 51, 32]
insert 77:  [6, 21, 60, 67, 73, 77, 99, 5, 51, 32]
insert  5:  [5, 6, 21, 60, 67, 73, 77, 99, 51, 32]
insert 51:  [5, 6, 21, 51, 60, 67, 73, 77, 99, 32]
insert 32:  [5, 6, 21, 32, 51, 60, 67, 73, 77, 99]
```

8.2.2 归并排序

程序 8.3 实现了从小到大排序的归并排序算法。算法的主函数是 merge_sort，它把待排序的列表等分成左右两个子列表，分别对它们递归调用 merge_sort 进行排序，然后调用辅助函数 merge_ordered_lists 把两个已经排好序的子列表归并在一起成为一个排好序的列表。归并过程中使用指示变量 i 和 j 分别指示第一个列表 s1 和第二个列表 s2 的待归并元素的索引值。第 5 行至第 8 行的 if 语句块将 s1[i] 和 s2[j] 中的最小值追加到列表 t 中，然后将最小值所在列表的指示变量加 1。如果 s1 或 s2 中的所有元素都已经追加到列表 t 中，则第 4 行至第 8 行的循

环结束，只需将另一个列表中的所有剩余元素追加到列表 t 中。归并过程包含三个循环：第 4 行至第 8 行的 while 循环、第 9 行将 s1 中的所有剩余元素追加到列表 t 中、第 10 行将 s2 中的所有剩余元素追加到列表 t 中。这些循环的运行次数的总和是两个子列表的长度之和。归并排序的运行时间包括三部分，即递归调用左子列表、递归调用右子列表和归并排好序的两个子列表。对于长度为 n 的列表，时间复杂度 $T(n)$ 满足方程 $T(n) = 2T(n/2) + \Theta(n)$，根据 Master 定理可知归并排序算法的时间复杂度是 $O(n \log n)$。

Listing 8.3: 归并排序

```
1  def merge_ordered_lists(s1, s2):
2      t = []
3      i = j = 0
4      while i < len(s1) and j < len(s2):
5          if s1[i] < s2[j]:
6              t.append(s1[i]); i += 1
7          else:
8              t.append(s2[j]); j += 1
9      t += s1[i:]
10     t += s2[j:]
11     print('%s␣+␣%s␣=>␣%s' % (s1, s2, t));
12     return t
13
14 def merge_sort(s):
15     if len(s) <= 1:
16         return s
17     mid = len(s) // 2
18     print('%s␣->␣%s␣+␣%s' % (s, s[:mid], s[mid:]));
19     left = merge_sort(s[:mid])
20     right = merge_sort(s[mid:])
21     return merge_ordered_lists(left, right)
22
23 s = [21, 73, 6, 67, 99, 60, 77, 5, 51, 32]
24 print(s); print(merge_sort(s))
```

Listing 8.4: 归并排序的运行过程

```
1 [21, 73, 6, 67, 99, 60, 77, 5, 51, 32]
2 [21, 73, 6, 67, 99, 60, 77, 5, 51, 32] ->
3     [21, 73, 6, 67, 99] + [60, 77, 5, 51, 32]
4 [21, 73, 6, 67, 99] -> [21, 73] + [6, 67, 99]
5 [21, 73] -> [21] + [73]
```

```
[21] + [73] => [21, 73]
[6, 67, 99] -> [6] + [67, 99]
[67, 99] -> [67] + [99]
[67] + [99] => [67, 99]
[6] + [67, 99] => [6, 67, 99]
[21, 73] + [6, 67, 99] => [6, 21, 67, 73, 99]
[60, 77, 5, 51, 32] -> [60, 77] + [5, 51, 32]
[60, 77] -> [60] + [77]
[60] + [77] => [60, 77]
[5, 51, 32] -> [5] + [51, 32]
[51, 32] -> [51] + [32]
[51] + [32] => [32, 51]
[5] + [32, 51] => [5, 32, 51]
[60, 77] + [5, 32, 51] => [5, 32, 51, 60, 77]
[6, 21, 67, 73, 99] + [5, 32, 51, 60, 77] =>
    [5, 6, 21, 32, 51, 60, 67, 73, 77, 99]
[5, 6, 21, 32, 51, 60, 67, 73, 77, 99]
```

8.2.3 线性查找

程序 8.5 实现了线性查找算法。第 2 行至第 3 行的循环在列表 s 中查找指定元素 k 是否出现，若出现则返回其索引值，否则在第 4 行返回-1 表示未找到。循环在最坏情形下的运行次数是列表 s 的长度 n，因此线性查找算法的时间复杂度是 $O(n)$。

Listing 8.5: **线性查找**

```
1 def linear_search(s, k):
2     for i in range(len(s)):
3         if s[i] == k: return i
4     return -1
```

8.2.4 二分查找

程序 8.6 实现了二分查找算法。假定列表已经按照从小到大的次序排好序，查找范围的索引值的下界和上界分别为 low 和 high，则可计算中间位置的索引值 mid。二分查找算法采用迭代方法，将指定元素 k 和列表 s 的中间位置的元素进行比较。如果比较的结果是相等，则已找到并返回。如果比较的结果是小于，则只需在左半边的子列表中继续查找。否则，只需在右半边的子列表中继续查找。循环的终止条件是下界大于上界，如果此条件满足则表示未找到。每进行一次迭代，查找范围缩小一半。对于长度为 n 的列表 s，循环次数不超过 $\lceil \log n \rceil$，因此二分查找算法的时间复杂度是 $O(\log n)$。

Listing 8.6: 二分查找

```
def binary_search(s, k):
    low = 0; high = len(s) - 1
    while low <= high:
        mid = (high + low) // 2
        print('(%2d,␣␣%2d)␣low␣=␣%d,␣mid␣=␣%d,␣high␣=␣%d'
              % (k, s[mid], low, mid, high))
        if k == s[mid]:
            return mid
        elif k < s[mid]:
            high = mid - 1
        else:
            low = mid + 1
    return -1

s = [5, 6, 21, 32, 51, 60, 67, 73, 77, 99]
print(binary_search(s, 77)); print(binary_search(s, 31))
```

Listing 8.7: 二分查找的运行过程

```
(77,  51) low = 0, mid = 4, high = 9
(77,  73) low = 5, mid = 7, high = 9
(77,  77) low = 8, mid = 8, high = 9
8
(31,  51) low = 0, mid = 4, high = 9
(31,   6) low = 0, mid = 1, high = 3
(31,  21) low = 2, mid = 2, high = 3
(31,  32) low = 3, mid = 3, high = 3
-1
```

8.2.5 穷举法求解 3-sum 问题

3-sum 问题的描述如下：给定一个整数 x 和一个由整数构成的集合 S，从 S 中找一个由三个元素构成的子集，该子集中的三个元素之和必须等于 x。穷举法是一种简单直接的算法，即列举所有由三个元素构成的子集，逐个检查其是否满足条件。程序 8.8 实现了穷举法求解 3-sum 问题。三重循环的最内层循环的运行次数是 $\sum_{i=0}^{n-3}\sum_{j=i+1}^{n-2}(n-j-1) = n(n-1)(n-2)/6$，因此该算法的时间复杂度是 $O(n^3)$。

Listing 8.8: 穷举法求解 3-sum 问题

```
def exhaustive_search_3_sum(S, x):
```

```
    n = len(S)
    for i in range(n - 2):
        for j in range(i + 1, n - 1):
            for k in range(j + 1, n):
                if S[i] + S[j] + S[k] == x:
                    return S[i], S[j], S[k]
    return ()

S = [21, 73, 6, 67, 99, 60, 77, 5, 51, 32]
print(exhaustive_search_3_sum(S, 152))  #  (21, 99, 32)
```

对以上穷举法可以进行改进得到时间复杂度为 $O(n^2 \log n)$ 的算法 [SW2011]。假定存在一个子集 $\{a, b, c\}$ 满足 $x = a + b + c$，则 $x - a = b + c$。原问题可以转换为另一个问题：对于 S 中的任意两个元素 b 和 c，在 $S - \{b, c\}$ 中查找一个等于 $x - b - c$ 的元素。为了提高查找的效率，可以先将所有元素按照从小到大的顺序排序，然后使用二分查找。

8.2.6 穷举法求解 subset-sum 问题

subset-sum 问题的描述如下：给定一个整数 x 和一个由整数构成的集合 S，从 S 中找一个子集，该子集中的所有元素之和必须等于 x。使用穷举法列举 S 的所有子集，逐个检查其是否满足条件。设 S 包含 n 个元素，则 S 的每个子集 T 和 n 位二进制数存在一一映射。n 位二进制数的第 k 位为 1 表示第 k 个元素在子集 T 中，为 0 则表示不在。

程序 8.9 实现了穷举法求解 subset-sum 问题。第 6 行的表达式 i >> j 使用整数的比特移位运算符将 i 的二进制形式向右移动 j 次，然后判断其是否是奇数。如果是，则从右边数的第 j 位为 1，将第 j 个元素追加到子集中。程序包含两个内循环：第 5 行至第 6 行的 for 循环和第 7 行的 sum(subset)。它们的运行次数都不超过 $n(2^n - 1)$，因此该算法的时间复杂度是 $O(n2^n)$。

Listing 8.9: **穷举法求解 subset-sum 问题**

```
1  def exhaustive_search_subset_sum(S, x):
2      n = len(S)
3      for i in range(1, 2 ** n):
4          T = []
5          for j in range(n):
6              if (i >> j) % 2 == 1: T.append(S[j])
7          if sum(T) == x: return T
8      return []
9
10 S = [21, 73, 6, 67, 99, 60, 77, 5, 51, 32]
11 print(exhaustive_search_subset_sum(S, 135))  #  [73, 6, 5, 51]
```

8.3 程序运行时间的测量

timeit 模块的 timeit 函数测量一个程序重复运行多次所需时间。程序 8.10 定义的两个字符串 s1 和 s2 表示两个程序。程序 s1 向一个空列表中添加 10 万个元素。程序 s2 生成一个指定长度的由随机数构成的列表，然后对其排序。第 18 行测量了程序 s1 运行 10 次的平均运行时间。第 18 行测量了程序 s1 运行 10 次的平均运行时间。第 20 行至第 22 行的循环使用不同的长度值运行程序 s2，测量其运行 10 次的平均运行时间。

Listing 8.10: **timeit 函数测量一个程序重复运行多次所需时间**

```
1  s1 = """\
2  a = []
3  for i in range(100000):
4      a.append(i)
5  """
6
7  s2 = """\
8  import random
9  def sort_random_list(n):
10     alist = [random.random() for i in range(n)]
11     alist.sort()
12
13 sort_random_list(%d)
14 """
15
16 import timeit
17 N = 10
18 print('%.4f' % (timeit.timeit(stmt=s1, number=N) / N)) # 0.0091
19
20 for n in [10000, 20000, 40000, 80000]:
21     t = timeit.timeit(stmt=s2 % n, number=N) / N
22     print('%d␣:␣%.4f' % (n, t), end = '␣')
23 # 10000 : 0.0025 20000 : 0.0053 40000 : 0.0126 80000 : 0.0283
```

性能分析工具 line_profiler 可测量程序中加了@profile 标记的函数中每一行语句的运行时间。在 Spyder 的 IPython 窗口中运行命令“pip install line_profiler”安装 line_profiler。以计算 Julia 集 [SL2018] 并绘图显示的程序 8.11 为例 [MG2017] 说明以行为单位的时间性能测量。

对于复平面上的每个点 z，Julia 集由迭代过程 $z_0 = z, z_{n+1} = z_n^2 + c$ 定义。迭代过程的终止条件是 $|z| \geq 2$ 或者 $n \geq N$，其中 N 是预先设定的最大迭代次数。通过迭代过程，可计算每个点 z 在终止时的迭代次数 n。将以原点为中心的正方形区域内定义的等距网格上的每个点 的迭

代次数映射到某一灰度或颜色，即可生成一个灰度或彩色图片。图 8.1 展示了选取六个不同的 c 值所得到的有趣图片。程序 8.11 的 calc_z_python 函数计算每个点的迭代次数，calc_Julia 函数对每个点调用 calc_z_python 函数。time 模块的 time 函数返回从一个时间起点 (1970 年 1 月 1 日世界标准时间 00:00:00) 到当前时刻所经历的秒数。在运行一个函数之前和之后分别使用 time 函数记录两个数值，它们的差值即为该函数的运行时间。函数 show_color 将每个点的迭代次数转换为一个六进制的三位整数，它的每一位乘以 50 分别为组成一种颜色的三原色之一的数值。为了生成图片，需要将程序 8.11 的第 47 行的 False 改为 True 并且删除第 6 行和第 16 行的@profile 标记。

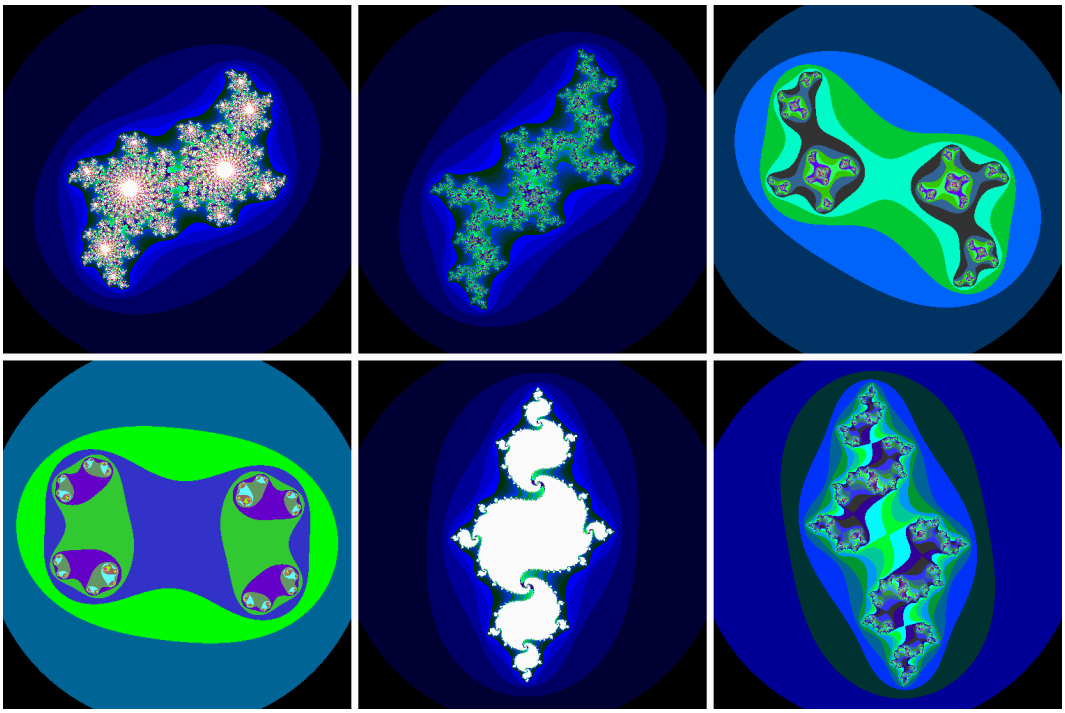


图 8.1: Julia 集生成的图片

Listing 8.11: **julia_set.py**

```
1  import time; import array; import numpy as np
2
3  x1, x2, y1, y2 = -1.8, 1.8, -1.8, 1.8     # range of complex space
4  c_real, c_imag = -0.62772, -.42193
5
6  @profile
7  def calc_z_python(max_iter, zs, c):
8      output = [0] * len(zs)
9      for i in range(len(zs)):
10         z = zs[i];  n = 0
```

```
11          while abs(z) < 2 and n < max_iter:
12              z = z * z + c; n += 1
13          output[i] = n
14      return output
15
16  @profile
17  def calc_Julia(show, length, max_iter):
18      xs = np.linspace(x1, x2, length)
19      ys = np.linspace(y1, y2, length)
20      zs = []; c = complex(c_real, c_imag)
21      for x in xs:
22           for y in ys:
23               zs.append(complex(x, y))
24      start_time = time.time()
25      output = calc_z_python(max_iter, zs, c)
26      end_time = time.time()
27      print("%.4fs" % (end_time - start_time))  #  1.3954s
28      if show: show_image(output, length, max_iter)
29
30  from PIL import Image
31  def show_image(output_raw, length, max_iter):
32      # rescale output_raw to be in the inclusive range [0..215]
33      max_value = float(max(output_raw))
34      output_raw_limited = [int(float(o) / max_value * 215) \
35                            for o in output_raw]
36      rgb = array.array('B')
37      for o in output_raw_limited:
38          r = o // 36; o = o % 36; g = o // 6; b = o % 6
39          rgb.append(r*50); rgb.append(g*50); rgb.append(b*50);
40      im = Image.new("RGB", (length, length));
41      im.frombytes(rgb.tobytes(), "raw", "RGB")
42      im.show()
43
44  calc_Julia(show=False, length=500, max_iter=200)
```

假定 Anaconda 的安装路径是"D:\Anaconda"，在操作系统的命令行窗口进入文件"julia_set.py"所在目录，然后运行命令"D:\Anaconda\Scripts\kernprof -l -v julia_set.py"可显示程序中 calc_z_python 和 calc_Julia 这两个函数的每行语句的运行时间 (图 8.2)。

```
Total time: 9.59807 s
File: julia_set.py
Function: calc_z_python at line 6

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     6                                           @profile
     7                                           def calc_z_python(max_iter, zs, c):
     8         1        632.0    632.0      0.0      output = [0] * len(zs)
     9    250001     118595.0      0.5      1.2      for i in range(len(zs)):
    10    250000     145802.0      0.6      1.5          z = zs[i];  n = 0
    11   6028026    4992046.0      0.8     52.0          while abs(z) < 2 and n < max_iter:
    12   5778026    4203592.0      0.7     43.8              z = z * z + c; n += 1
    13    250000     137400.0      0.5      1.4          output[i] = n
    14         1          0.0      0.0      0.0      return output

Total time: 15.2133 s
File: julia_set.py
Function: calc_Julia at line 16

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    16                                           @profile
    17                                           def calc_Julia(show, length, max_iter):
    18         1        221.0    221.0      0.0      xs = np.linspace(x1, x2, length)
    19         1         97.0     97.0      0.0      ys = np.linspace(y1, y2, length)
    20         1          3.0      3.0      0.0      zs = []; c = complex(c_real, c_imag)
    21       501        354.0      0.7      0.0      for x in xs:
    22    250500     152269.0      0.6      1.0           for y in ys:
    23    250000     216797.0      0.9      1.4               zs.append(complex(x, y))
    24         1          5.0      5.0      0.0      start_time = time.time()
    25         1   14843542.0 14843542.0     97.6        output = calc_z_python(max_iter, zs, c)
    26         1          4.0      4.0      0.0      end_time = time.time()
    27         1         47.0     47.0      0.0      print("%.4f seconds" % (end_time - start_time))
    28         1          2.0      2.0      0.0      if show: show_image(output, length, max_iter)
```

图 8.2: 以行为单位的运行时间测量

calc_Julia 函数的第 25 行的函数调用消耗了整个函数大部分的运行时间，而 calc_z_python 函数中的内循环消耗了整个函数大部分的运行时间，是整个程序的性能瓶颈。

8.4 Cython

C 等编译型语言的语法要求程序中的所有变量必须具有类型声明，类型的分析和检查由编译器在编译程序时完成。Python 语言的语法要求程序中的变量不能有类型声明，类型的分析和检查由 Python 解释器在运行程序时完成。与 C 语言相比，这种动态特性虽然降低了编写程序的工作量，但也同时降低了程序的运行效率。Cython 语言 [CythonDoc] 是 Python 语言的扩展，其语法允许为程序中的变量提供类型声明。Cython 编译器可以将添加了类型声明的 Python 程序 (即 Cython 程序) 编译成 C 语言程序，再使用 C 语言编译器将其编译成可从 Python 程序中调用的扩展模块，从而提高整个程序的运行效率。下面以计算圆周率 $\pi = 4(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + ...)$ 的程序为例说明在 Windows 操作系统中使用 Cython 的主要步骤。

1. 从微软公司网站 https://visualstudio.microsoft.com/downloads/下载和安装 Visual Studio 或 Build Tools for Visual Studio。

2. 假定 Anaconda 的安装路径是“D:\Anaconda”，将“D:\Anaconda”和“D:\Anaconda\Library\bin”这两个路径追加到操作系统的 path 环境变量中。

3. 编写 Python 程序，按照 Cython 要求使用 cdef 关键字给变量添加类型声明，保存为文件 calc_pi_cython.pyx(程序 8.12)。设文件所在目录为“D:\src\julia”。

Listing 8.12: **calc_pi_cython.pyx**

```
import cython; import numpy as np

@cython.cdivision(True)
def calc_pi(int n):
    cdef double pi = 0
    cdef int i
    for i in range(1, n, 4):
        pi += 4.0 / i
    for i in range(3, n, 4):
        pi -= 4.0 / i
    return pi
```

4. 在“D:\src\julia”目录下编写一个文件 setup_pi.py(程序 8.13)。

Listing 8.13: **setup_pi.py**

```
from distutils.core import setup
from Cython.Build import cythonize
import numpy as np
setup(ext_modules=cythonize('calc_pi_cython.pyx'),
include_dirs=[np.get_include()],
requires=['Cython', 'numpy'])
```

5. 打开命令行窗口 (控制台)，进入“D:\src\julia”目录，然后运行如下命令 (8.14) 将程序 8.12 编译成为扩展模块文件 calc_pi_cython.cp38-win32.pyd。

Listing 8.14: **将程序 8.12 编译成为扩展模块**

```
python setup_pi.py build_ext --inplace
```

6. 在“D:\src\julia”目录下编写程序 8.15。s1 表示的程序调用从扩展模块 calc_pi_cython 导入的 calc_pi 函数计算 π。s2 表示的程序使用 Numpy 数组计算 π。输出结果显示前者的运行时间是后者的 51.0%。

Listing 8.15: **calc_pi_time.py**

```
s1 = """\
from calc_pi_cython import calc_pi
calc_pi(%d)
```

```
"""

s2 = """\
import numpy as np
n = %d
np.sum(4.0 / np.r_[1:n:4, -3:-n:-4])
"""

import timeit
N = 10; n = 10000000
print("%.4f" % (timeit.timeit(stmt=s1%n, number=N) / N))
# 0.0367
print("%.4f" % (timeit.timeit(stmt=s2%n, number=N) / N))
# 0.0720
```

如果在 Linux 操作系统中使用 Cython，主要步骤和上述类似，只是不需要前两步，因为 Linux 操作系统使用 GCC 编译器并且环境变量在安装 Anaconda 时已自动设置。生成的扩展模块文件为 calc_pi_cython.cpython-38-x86_64-linux-gnu.so。

按照同样的步骤编写程序 8.16 用 Cython 实现 calc_z_python 函数。

Listing 8.16: **calc_z_cython.pyx**

```
def calc_z(int maxiter, zs, c):
    """calc output list using Julia update rule"""
    cdef unsigned int i, n
    cdef double complex z
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        while n < maxiter and \
            (z.real * z.real + z.imag * z.imag) < 4:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

编写一个文件 setup_z.py(程序 8.17)，然后在控制台运行命令 8.18 将程序 8.16 编译成为扩展模块文件 calc_z_cython.cp38-win32.pyd。

Listing 8.17: **setup_z.py**

```
from distutils.core import setup
from Cython.Build import cythonize
setup(ext_modules=cythonize("calc_z_cython.pyx"))
```

Listing 8.18: **将程序 8.16 编译成为扩展模块**

```
python setup_z.py build_ext --inplace
```

在程序 8.11 中添加语句“from calc_z_cython import calc_z”，然后删除 calc_z_python 函数，将对这个函数的调用改为对扩展模块中 calc_z 函数的调用，保存为 julia_set_cython.py(程序 8.19)。运行 julia_set_cython.py 的输出结果显示 calc_z 函数的运行时间是 0.2891 秒，而运行 julia_set.py 的输出结果显示 calc_z_python 函数的运行时间是 1.3954 秒。

Listing 8.19: **julia_set_cython.py**

```
import time; import array; import numpy as np
from calc_z_cython import calc_z

...

def calc_Julia(show, length, max_iter):
    ...
    start_time = time.time()
    output = calc_z(max_iter, zs, c)
    end_time = time.time()
    print("%.4fs" % (end_time - start_time))  #  0.2891s
    if show: show_image(output, length, max_iter)

...
```

本节列举的两个 Cython 程序较简单。Cython 文档 [CythonDoc] 对 Cython 技术进行了全面介绍，包括内存分配、扩展类型、调用 C 函数、调试和性能剖析等。

8.5 实验 8：程序运行时间的分析和测量

实验目的

本实验的目的是掌握程序运行时间的分析和测量。

提交方式

在 Blackboard 提交一个文本文件 (txt 后缀)，文件中记录每道题的源程序和运行结果。

实验内容

1. 实现一个时间复杂度为 $O(n^2 \log n)$ 的算法，解决 3-sum 问题。

2. 生成长度为 100,200,...,900,1000 的由随机数构成的列表，使用 timeit 函数分别测量插入排序 (8.1)、归并排序 (8.3) 和快速排序 (4.15) 这三种排序算法的运行时间。对于每个长度测量 10 次计算平均值。在测量之前需要删除程序中的 print 语句。

3. 使用 timeit 函数测量归并排序 (8.3) 的两个函数中的每条语句的运行时间。在测量之前需要删除程序中的 print 语句。

第 9 章　Matplotlib 库简介

Matplotlib(https://matplotlib.org/stable/index.html) 是一个 Python 扩展库，它提供了丰富的绘图功能，可绘制多种二维和三维图示，直观地呈现科学计算的输入数据和输出结果。

Matplotlib 绘图在图片 (Figure) 对象上进行。每个图片对象包含一个或多个独立的绘图区域 (Axes)。在每个绘图区域里，需要绘制的点用二维或三维坐标表示。

Matplotlib 的使用方式有两种。一种是面向对象的方式，即创建 Figure 和 Axes 对象，然后调用它们的方法。另一种是基于状态的方式，即调用 matplotlib.pyplot 的函数，这些函数调用会保存状态。前者提供了丰富的功能，适合绘制较为复杂的图示。后者易于使用，适合绘制较为简单的图示。

9.1　基于状态的绘图方式

matplotlib.pyplot 模块提供了一些函数，实现了类似 MATLAB 的绘图功能。这些函数作用于当前的绘图区域。每个函数调用完成以后会保存对状态的改变。

plot 函数可绘制二维平面上由若干数据点组成的图示，可以仅显示孤立的点，也可以把点连成线。程序 9.1 绘制了一些时间复杂度函数随问题规模增长的变化情况 (图 9.1)。第 4 行创建一个 Numpy 数组存储了数据点的 x 坐标。第 5 行创建了一个图片对象，其中 figsize 参数表示以英寸为单位的图片的宽度和高度，dpi 参数表示以每英寸的点数表示的分辨率。第 6 行设置 x 轴的刻度为数组 x 中的元素。第 7 行至第 14 行分别调用 plot 函数绘制了四个时间复杂度函数。plot 函数的前两个参数是分别存储了数据点的 X 坐标和 Y 坐标的两个数组，其余参数都有名称。表 9.1 列出了一些常用参数的说明。第 15 行给两个坐标轴添加标签。第 16 行给图添加标题。第 17 行显示图例和添加背景网格线。

Listing 9.1: **plot 函数绘制时间复杂度函数**

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.arange(10, 31, 1.0)
```

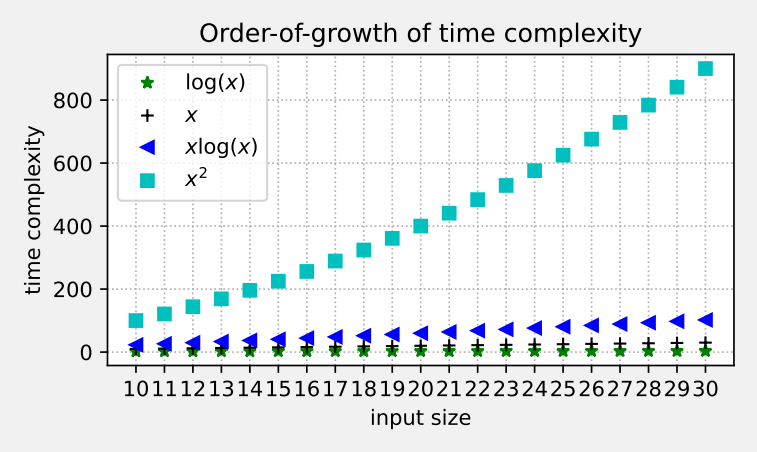


图 9.1: plot 函数绘制时间复杂度函数

```
5  plt.figure(figsize=(5, 3), dpi = 300)
6  plt.xticks(x)
7  plt.plot(x, np.log(x), color='g', linestyle='None',
8           marker = '*', label=r'$\log(x)$')
9  plt.plot(x, x, color='k', linestyle='None',
10          marker = '+', label=r'$x$')
11 plt.plot(x, x * np.log(x), color='b', linestyle='None',
12          marker = '<', label=r'$x\log(x)$')
13 plt.plot(x, x ** 2, color='c', linestyle='None',
14          marker = 's', label=r'$x^{2}$')
15 plt.xlabel('input␣size'); plt.ylabel('time␣complexity')
16 plt.title("Order-of-growth␣of␣time␣complexity")
17 plt.legend(); plt.grid(linestyle=':')
18 plt.tight_layout(); plt.show()
```

如果在程序 9.1 中增加绘制比平方函数增长更快的函数，则前四个函数的图像几乎重合，原因是不同函数的函数值的变化范围差别太大。为了清晰显示时间复杂度函数的差别，程序 9.2 中的第 7 行将 y 轴设为对数尺度，得到图 9.2。

Listing 9.2: **plot 函数绘制时间复杂度函数 (对数尺度)**

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  x = np.arange(10.0, 205.0, 10.0)
5  plt.figure(figsize=(8, 4), dpi = 300)
6  plt.xticks(x)
```

参数名称	参数定义和示例
color	颜色，可以是蓝色 ('b')、绿色 ('g')、红色 ('r')、蓝绿色 ('c')、洋红色 ('m')、黄色 ('y')、黑色 ('k') 和白色 ('w')，或十六进制自定义"#rrggbb"
linestyle	线的风格，可以是实线 ('-')、虚线 ('–')、点划线 ('-.')、点线 (':') 和不连线 ('None')
marker	数据点的标志，可以是'+'、'_'、'*'、'o'、'v'、'∧'、'<'、'>'、's'、'p'、'x'、'D'、'd'、'1'、'2'、'3' 和'4' 等
markersize	数据点的标志的尺寸
label	图例中显示的标签，可以用 r'$...$' 的形式写 Latex 公式

表 9.1: plot 函数的一些常用参数

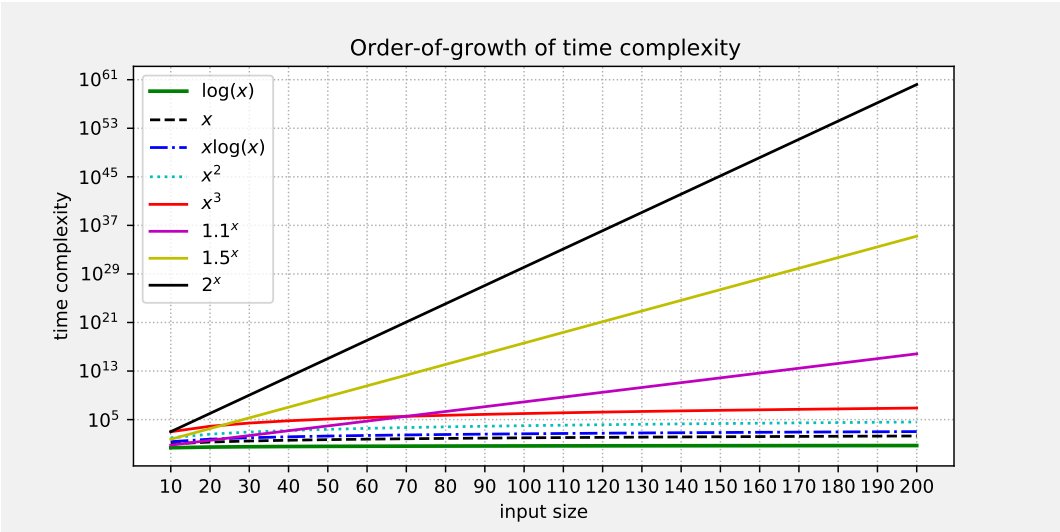


图 9.2: plot 函数绘制时间复杂度函数 (对数尺度)

```
7  plt.yscale('log')
8  plt.plot(x, np.log(x), color='g', linestyle='-',
9           linewidth = 2, label=r'$\log(x)$')
10 plt.plot(x, x, color='k', linestyle='--', label=r'$x$')
11 plt.plot(x, x * np.log(x), color='b',
12          linestyle='-.', label=r'$x\log(x)$')
13 plt.plot(x, x ** 2, color='c', linestyle=':', label=r'$x^{2}$')
14 plt.plot(x, x ** 3, color='r', label=r'$x^{3}$')
15 plt.plot(x, 1.2 ** x, color='m', label=r'$1.1^{x}$')
16 plt.plot(x, 1.5 ** x, color='y', label=r'$1.5^{x}$')
17 plt.plot(x, 2 ** x, color='k', label=r'$2^{x}$')
18 plt.xlabel('input␣size'); plt.ylabel('time␣complexity')
19 plt.title("Order-of-growth␣of␣time␣complexity")
20 plt.legend(); plt.grid(linestyle=':'); plt.show()
```

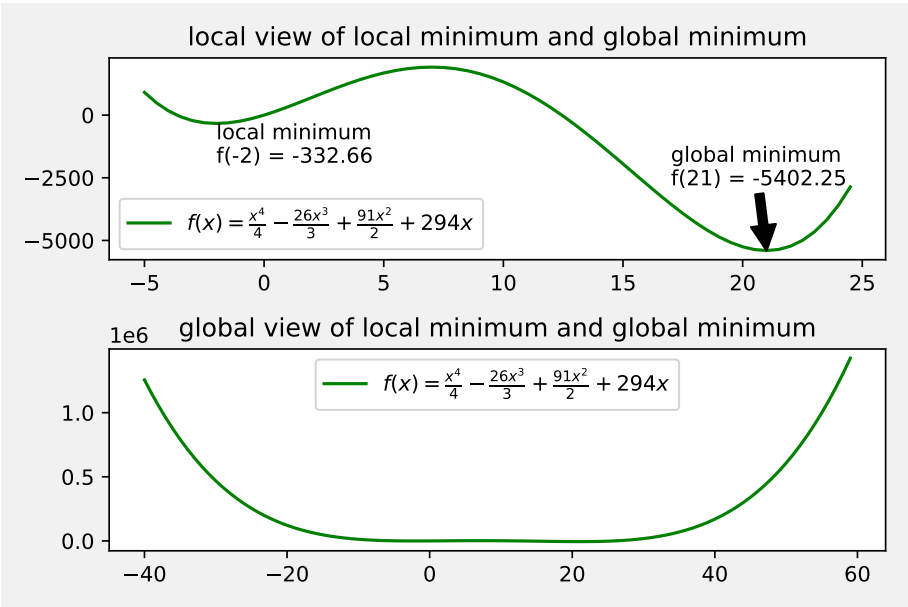


图 9.3: plot 函数绘制多项式函数的局部和全局极小值

程序 9.3 绘制了函数 $f(x) = \frac{x^4}{4} - \frac{26x^3}{3} + \frac{91x^2}{2} + 294x$ 在两个不同区间 [-5, 24] 和 [-40, 59] 上的图像 (图 9.3)。subplot 函数可以在一个图片中绘制一个子图。通常情况下，属于同一个图片的所有子图的尺寸相同，并且按行和列均匀排布。subplot 函数的前两个参数定义了子图的行数和列数，第三个参数指定当前子图的位置编号。第 15 行使用 text 函数在第一个子图中的指定位置标示了 $x = 2$ 时的局部极小值。第 16 行至第 18 行调用 annotate 函数在第一个子图中的指定位置标示了 $x = 21$ 时的全局极小值，annotate 函数可以显示一个箭头指向指定坐标并用文字注解。第 26 行调用的 tight_layout 函数可以调整两个子图的间距，以防止文字发生重叠。

Listing 9.3: **plot 函数绘制多项式函数的局部和全局极小值**

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(-5, 25, 0.5)
def f(x):
    return x**4 / 4 - 26 * x**3 / 3 + 91 * x ** 2 / 2 + 294 * x

plt.figure(figsize=(6, 4), dpi = 300)

plt.subplot(2,1,1)
flb = \
  r'$f(x)␣=␣\frac{x^4}{4}-\frac{26x^3}{3}+\frac{91x^2}{2}+294x$'
plt.plot(x, f(x), color='g', linestyle='-', label=flb)
plt.title("local␣view␣of␣local␣minimum␣and␣global␣minimum")
plt.text(-2, -1900, 'local␣minimum\nf(-2)␣=␣-332.66')
plt.annotate('global␣minimum\nf(21)␣=␣-5402.25',
             xy=(21, -5402.25), xytext=(17, -2800),
             arrowprops=dict(facecolor='black'))
plt.legend();

plt.subplot(2,1,2)
x = np.arange(-40, 60)
plt.plot(x, f(x), color='g', linestyle='-', label=flb)
plt.title("global␣view␣of␣local␣minimum␣and␣global␣minimum")
plt.legend()
plt.tight_layout(); plt.show()
```

程序 9.4 使用多种风格绘制了 6 个子图，依次是条状图、误差条图、以日期为横坐标的随机游走连线图、x 轴和 y 轴均为对数尺度的连线图、仅 x 轴为对数尺度的连线图和仅 y 轴为对数尺度的连线图 (图 9.4)。

Listing 9.4: **在子图中使用多种风格绘图**

```
import numpy as np
import matplotlib as mpl; import matplotlib.pyplot as plt

plt.figure(figsize = (9, 6))

plt.subplot(2, 3, 1)
```

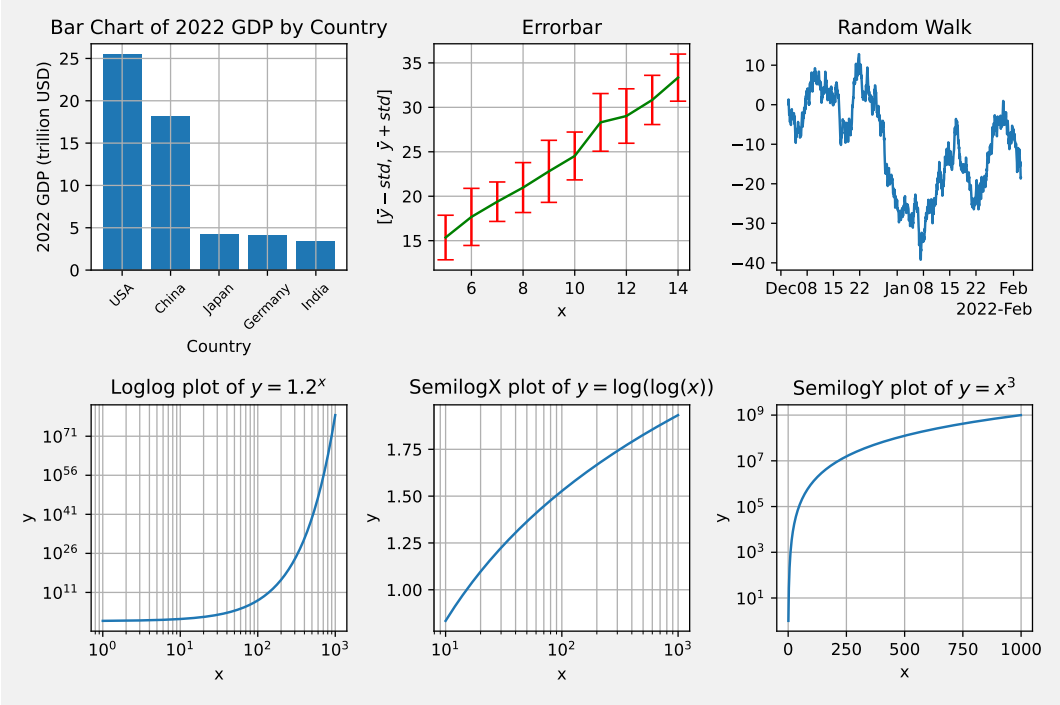


图 9.4: 在子图中使用多种风格绘图

```
x = ['USA', 'China', 'Japan', 'Germany', 'India']
y = [25.46, 18.10, 4.23, 4.08, 3.39]
plt.bar(x,y); plt.grid(); ax = plt.gca()
ax.set_xticks(x); ax.set_xticklabels(x, rotation=45)
ax.tick_params(axis='x', labelsize=8)
plt.title('Bar Chart of 2022 GDP by Country')
plt.xlabel('Country'); plt.ylabel('2022 GDP (trillion USD)')

plt.subplot(2, 3, 2)
np.random.seed(10); xr = np.arange(5, 15, 1); n = 20
y = np.zeros(len(xr)); yerr = np.zeros(len(xr))
for i, x in enumerate(xr):  # i是元素的索引值, x是对应的元素
    d = 2*(np.zeros(n) + x) + 5 + np.random.randn(n)*3
    y[i] = np.mean(d); yerr[i] = np.std(d) # 均值和标准差
plt.errorbar(xr, y, yerr=yerr, color='g', ecolor='r', capsize=5)
plt.xlabel('x'); plt.ylabel(r'$[\bar{y}-std, \; \bar{y}+std]$');
plt.title('Errorbar'); plt.grid()

plt.subplot(2, 3, 3)
dates = np.arange(np.datetime64('2021-12-03'),
```

```
                    np.datetime64('2022-02-03'),
                    np.timedelta64(1, 'h'))
data = np.cumsum(np.random.randn(len(dates))) # 累积和
plt.plot(dates, data)
cdf = mpl.dates.ConciseDateFormatter(plt.gca().xaxis.
                                     get_major_locator())
plt.gca().xaxis.set_major_formatter(cdf);
plt.title('Random␣Walk')

plt.subplot(2, 3, 4)
x = np.arange(1, 1000); y = 1.2**x
plt.loglog(x,y)
plt.title('Loglog␣plot␣of␣' + r'$y=1.2^x$')
plt.xlabel('x'); plt.ylabel('y'); plt.grid(which='both')

plt.subplot(2, 3, 5)
x = np.arange(10, 1000); y =  np.log(np.log(x))
plt.semilogx(x,y)
plt.title('SemilogX␣plot␣of␣' + r'$y=\log(\log(x))$')
plt.xlabel('x'); plt.ylabel('y'); plt.grid(which='both')

plt.subplot(2, 3, 6)
x = np.arange(1, 1000); y = x**3
plt.semilogy(x,y)
plt.title('SemilogY␣plot␣of␣' + r'$y=x^3$')
plt.xlabel('x'); plt.ylabel('y'); plt.grid(which='both')

plt.tight_layout(); plt.show()
```

9.2 面向对象的绘图方式

程序 9.5 采用面向对象的方式绘制了程序 9.2 中的几个函数。第 5 行创建了 Figure 和 Axes 对象，然后调用 Axes 对象 ax 的方法。

Listing 9.5: **面向对象的方式绘制时间复杂度函数**

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.arange(5, 25)
```

```
5  fig, ax = plt.subplots(figsize=(6, 4), dpi = 300)
6  ax.set_yscale('log')
7  ax.plot(x, np.log(x), color='g', linestyle='-',
8          linewidth = 2, label=r'$\log(x)$')
9  ax.plot(x, x, color='k', linestyle='--', label=r'$x$')
10 ax.plot(x, x * np.log(x), color='b',
11         linestyle='-.', label=r'$x\log(x)$')
12 ax.plot(x, x ** 2, color='c', linestyle=':', label=r'$x^{2}$')
13 ax.plot(x, x ** 3, color='r', label=r'$x^{3}$')
14 ax.plot(x, 1.2 ** x, color='m', label=r'$1.1^{x}$')
15 ax.plot(x, 1.5 ** x, color='y', label=r'$1.5^{x}$')
16 ax.plot(x, 2 ** x, color='k', label=r'$2^{x}$')
17 ax.set_xlabel('input␣size'); ax.set_ylabel('time␣complexity')
18 ax.set_title("Order-of-growth␣of␣time␣complexity")
19 ax.legend(); plt.show()
```

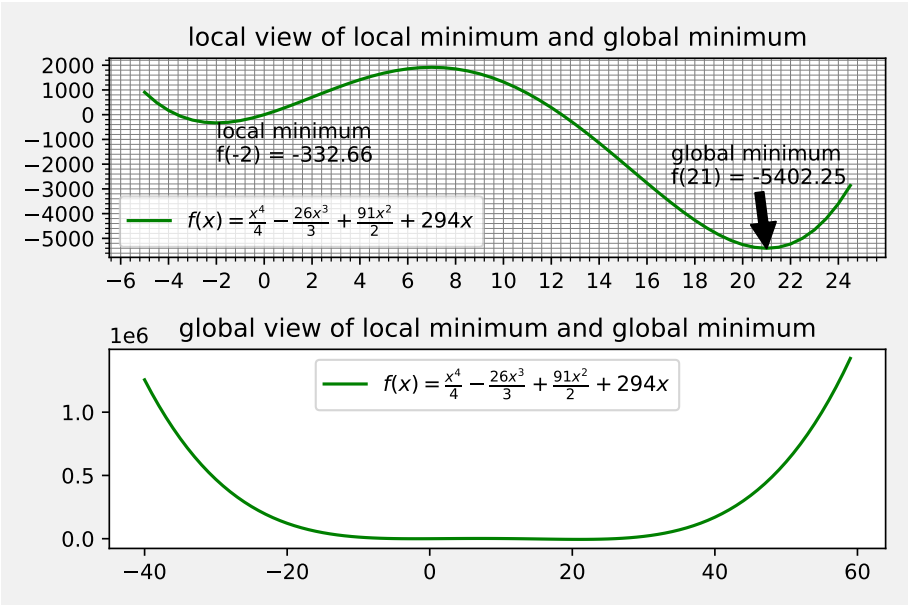


图 9.5: 面向对象的方式绘制多项式函数的局部和全局极小值

程序 9.6 采用面向对象的方式绘制了多项式函数的局部和全局极小值 (图 9.5)。第 9 行创建了 Figure 和 Axes 对象，两个 Axes 对象分别为 axs[0] 和 axs[1]。第 15 行至第 18 行指定了 x 轴和 y 轴的主刻度和次刻度的数量。第 19 行至第 23 行根据主刻度和次刻度绘制了网格线。第 35 行使用 300dpi(每英寸 300 个点) 的分辨率保存图像为指定的路径和名称的 PNG 格式文件。如果将文件的后缀改为“pdf”，则可保存为 PDF 格式文件。

Listing 9.6: **面向对象的方式绘制多项式函数的局部和全局极小值**

```
1 import numpy as np
2 import matplotlib as mpl; import matplotlib.pyplot as plt
3
4 x = np.arange(-5, 25, 0.5)
5 def f(x):
6     return x**4 / 4 - 26 * x**3 / 3 + 91 * x ** 2 / 2 + 294 * x
7 title = "local minimum and global minimum"
8
9 fig, axs = plt.subplots(2, 1, figsize=(6, 4))
10
11 flb = \
12   r'$f(x) = \frac{x^4}{4}-\frac{26x^3}{3}+\frac{91x^2}{2}+294x$'
13 axs[0].plot(x, f(x), color='g', linestyle='-', label=flb)
14 axs[0].set_title("local view of " + title)
15 axs[0].xaxis.set_major_locator(mpl.ticker.MaxNLocator(20))
16 axs[0].xaxis.set_minor_locator(mpl.ticker.MaxNLocator(100))
17 axs[0].yaxis.set_major_locator(mpl.ticker.MaxNLocator(10))
18 axs[0].yaxis.set_minor_locator(mpl.ticker.MaxNLocator(50))
19 grid_width = {"major":0.5, "minor":0.25}
20 for axis in ('x','y'):
21     for gt in grid_width.keys():
22         axs[0].grid(color="grey", which=gt, axis=axis,
23                     linestyle='-', linewidth=grid_width[gt])
24 axs[0].annotate('global minimum\nf(21) = -5402.25',
25                 xy=(21, -5402.25), xytext=(17, -2800),
26                 arrowprops=dict(facecolor='black'))
27 axs[0].legend();
28
29 x = np.arange(-40, 60)
30 axs[1].plot(x, f(x), color='g', linestyle='-', label=flb)
31 axs[1].set_title("global view of " + title)
32 axs[1].legend();
33
34 fig.tight_layout()
35 plt.savefig("D:/images/dg.png", dpi=300, facecolor="#f1f1f1")
36 plt.show()
```

程序 9.7 演示了如何绘制多个共享 x 轴的函数的图像 (图 9.6)。第 9 行创建了两个子图片，它们的布局方式是一行两列，宽度各占总宽度的 55%和 40%。左边的子图片只有一个绘图区域，

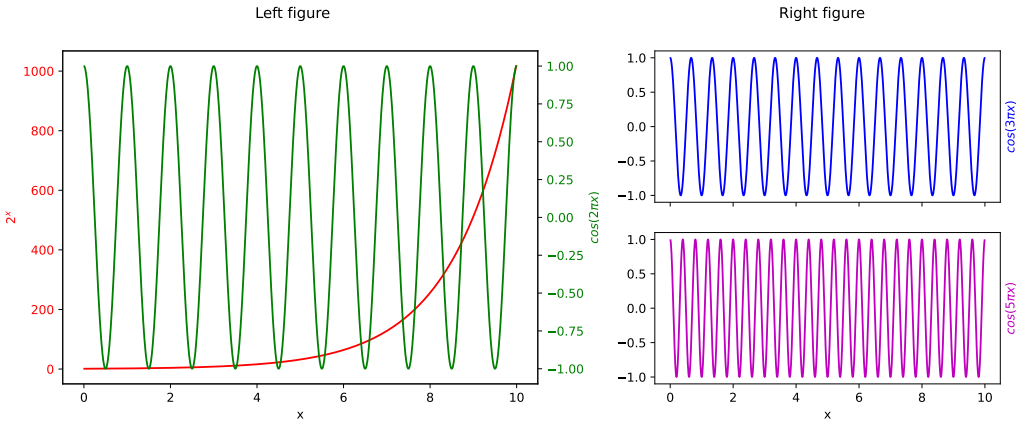


图 9.6: 多个共享 x 轴的函数的图像

在相同的自变量取值范围 [0,10] 内分别用红色和绿色绘制了函数 $y = 2^x$ 和 $y = cos(2\pi x)$ 的图像，左边框和右边框分别用红色和和绿色标注了对应于这两个函数的 y 轴刻度。右边的子图片有两个共享 x 轴的绘图区域，它们的布局方式是两行一列，在相同的自变量取值范围 [0,10] 内分别用蓝色和洋红色绘制了函数 $y = cos(3\pi x)$ 和 $y = cos(5\pi x)$ 的图像。

Listing 9.7: **绘制多个共享 x 轴的函数的图像**

```
import numpy as np
import matplotlib.pyplot as plt
x = np.arange(0.01, 10.0, 0.01)
data1 = 2 ** x; data2 = np.cos(2 * np.pi * x)
data3 = np.cos(3 * np.pi * x); data4 = np.cos(5 * np.pi * x)

color1 = 'r'; color2 = 'g'; color3 = 'b'; color4 = 'm'
fig = plt.figure(figsize=(12, 5))
subfigs = fig.subfigures(1, 2, width_ratios=[0.55, 0.4])
# 创建两个子图片，宽度各占总宽度的55\%和40\%
subfigs[0].suptitle('Left␣figure')
ax1 = subfigs[0].subplots()  # 创建第一个绘图区域
ax1.set_xlabel('x'); ax1.set_ylabel(r'$2^{x}$', color=color1)
ax1.plot(x, data1, color=color1)
ax1.tick_params(axis='x', labelcolor='black')
ax1.tick_params(axis='y', labelcolor=color1)
ax1_2 = ax1.twinx() # 创建第二个绘图区域，它和第一个绘图区域共享$x$轴
ax1_2.yaxis.set_label_position("right")
ax1_2.set_ylabel(r'$cos(2␣\pi␣x)$', color=color2)
ax1_2.plot(x, data2, color=color2)
```

```
ax1_2.tick_params(axis='y', labelcolor=color2)

subfigs[1].suptitle('Right figure')
# 创建两个共享$x$轴的绘图区域
ax2 = subfigs[1].subplots(2, 1, sharex=True)
ax2[0].plot(x, data3, color=color3)
ax2[0].yaxis.set_label_position("right")
ax2[0].set_ylabel(r'$cos(3 \pi x)$', color=color3)
ax2[1].plot(x, data4, color=color4)
ax2[1].set_xlabel('x')
ax2[1].yaxis.set_label_position("right")
ax2[1].set_ylabel(r'$cos(5 \pi x)$', color=color4)

plt.show()
```

9.3 三维数据的图示

有两个自变量和一个因变量的函数的数据点分布在三维空间，可以通过二维平面上的颜色映射图或等高线图显示，也可以通过表面图、线框图和三维等高线图显示。默认情况下，在 spyder 中运行程序生成的图片显示在右上角窗口。如果需要在一个独立窗口中显示图片，需要按照以下步骤操作。首先点击 spyder 的 Tools 菜单，选择 Preference 菜单项，此时出现一个对话框 (图 1.3)。然后在对话框左边的列表中选中“IPython console”，再点击右边窗口的“Graphics”标签。此时中间的“Graphics backend”部分有一个下拉列表。其默认选项是“Inline”，表示图片嵌入右上角窗口。选中其中的“IPython console”选项即可使得图片显示在一个独立窗口中。当一个三维图片显示在一个独立窗口中时，可以通过拖动鼠标旋转坐标轴，实现从多个视角的观察。

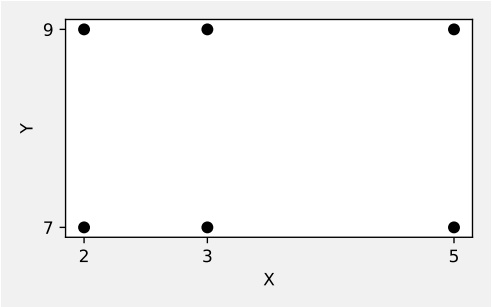
9.3.1 颜色映射图和等高线图

对于函数 $z = f(x, y)$，颜色映射图将对应于二维平面上的每个坐标 (x, y) 的 z 值映射到一种颜色显示在这一位置，等高线图将 z 值相同的 (x, y) 坐标连成线。

给定一个 x 坐标值的集合 $\{x_1, x_2, ..., x_m\}$ 和一个 y 坐标值的集合 $\{y_1, y_2, ..., y_n\}$，网格定义为 $\{(x_i, y_j)\}, \quad 1 \le i \le m, \quad 1 \le j \le n$。np.meshgrid 函数从 x 坐标值和 y 坐标值的数组生成一个网格，其返回值是两个二维数组，分别存储了网格上所有点的 x 坐标值和 y 坐标值。程序 9.10 演示了 np.meshgrid 函数生成的网格。

Listing 9.8: **np.meshgrid 函数生成的网格**

```
In[1]: x=[2,3,5]; y=[7,9]
In[2]: np.meshgrid(x,y)
Out[2]:
[array([[2, 3, 5],
        [2, 3, 5]]),
 array([[7, 7, 7],
        [9, 9, 9]])]
```



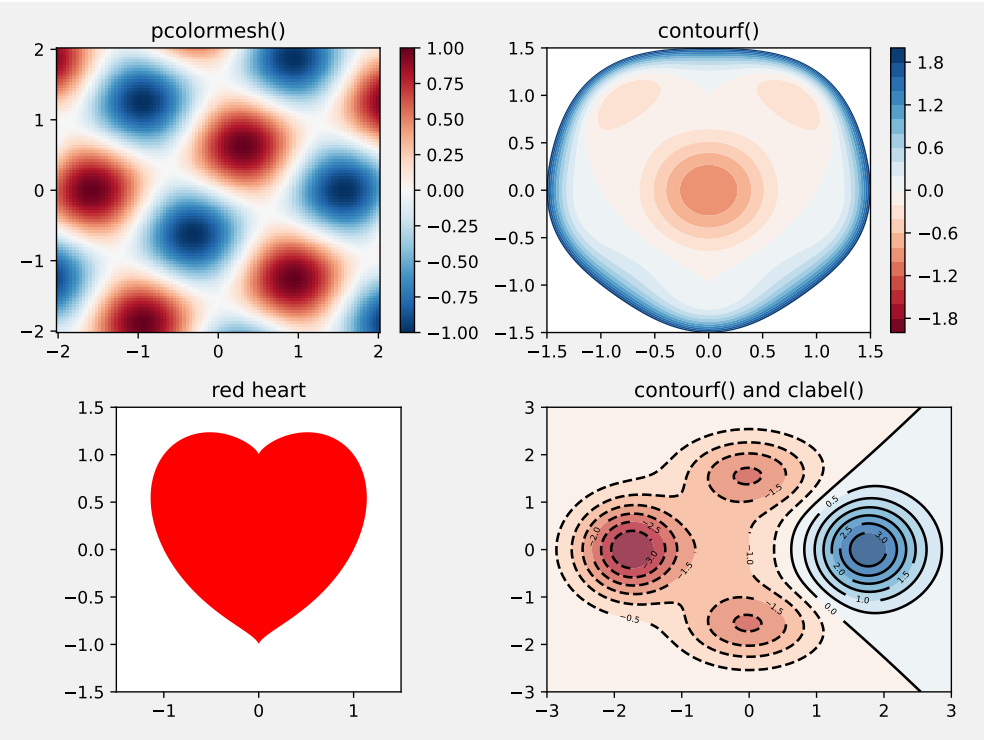


图 9.7: 颜色映射图和等高线图

图 9.7 由程序 9.9 绘制。第 4 行至第 5 行定义了一个由指定范围内的 (x, y) 坐标值构成的网格。第 5 行定义了函数 $z = \sin(x + 2y)sin(2x - y)$。第 9 行至第 10 行绘制了颜色映射图，其中参数 vmin 和 vmax 指定了映射到颜色的 z 值范围。第 16 行至第 17 行调用 contourf 函数绘制了函数 $z = (x^2 + y^2 - 1)^3 - x^2y^3$ 的等高线图，其中参数 levels 指定了所绘制的各等高线的 z 值，参数 cmap 指定了填充相邻等高线之间的区域的颜色映射。第 21 行用红色填充 $z = -1$ 和 $z = 0$ 这两条等高线之间的区域得到了心形图案。第 31 行绘制了函数 $z = (x^7 - y^6 + x^5 - y^4 + x^3 - y^2 + x - 1)e^{-x^2-y^2}$ 的 16 条等高线，并在第 32 行标注了每条线的 z 值。第 33 行至第 34 行使用指定的颜色映射填充相邻等高线之间的区域。

Listing 9.9: **颜色映射图和等高线图**

```
import numpy as np
import matplotlib.pyplot as plt

X, Y = np.meshgrid(np.linspace(-2, 2, 100),
                   np.linspace(-2, 2, 100))
Z = np.sin(X + 2 * Y) * np.cos(2 * X - Y)

fig, axs = plt.subplots(2, 2, figsize=(8, 6))
pc = axs[0, 0].pcolormesh(X, Y, Z, vmin=-1, vmax=1,
                          cmap='RdBu_r', shading='auto')
fig.colorbar(pc, ax=axs[0, 0])
axs[0, 0].set_title('pcolormesh()')

x, y = np.mgrid[-1.5:1.5:500j, -1.5:1.5:500j]
z = (x**2 + y**2 - 1)**3 - x**2 * y**3
co = axs[0, 1].contourf(x, y, z, levels=np.linspace(-2, 2, 21),
                        cmap=plt.cm.RdBu)
fig.colorbar(co, ax=axs[0, 1])
axs[0, 1].set_title('contourf()')

axs[1, 0].contourf(x, y, z, levels=[-1, 0], colors=["red"])
axs[1, 0].set_aspect("equal");
axs[1, 0].set_title('red␣heart')

def f(x,y):
    return (x**7 - y**6 + x**5 - y**4 + x**3 - y**2 + x - 1) * \
            np.exp(-x**2 -y**2)

x = y = np.linspace(-3, 3, 256)
X,Y = np.meshgrid(x, y)
C = axs[1, 1].contour(X, Y, f(X, Y), 16, colors='black')
axs[1, 1].clabel(C, inline=1, fontsize=5)
axs[1, 1].contourf(X, Y, f(X, Y), 16, alpha=.75,
                   cmap=plt.cm.RdBu)
axs[1, 1].set_title('contourf()␣and␣clabel()')

fig.tight_layout(); plt.show()
```

9.3.2 表面图、线框图和三维等高线图

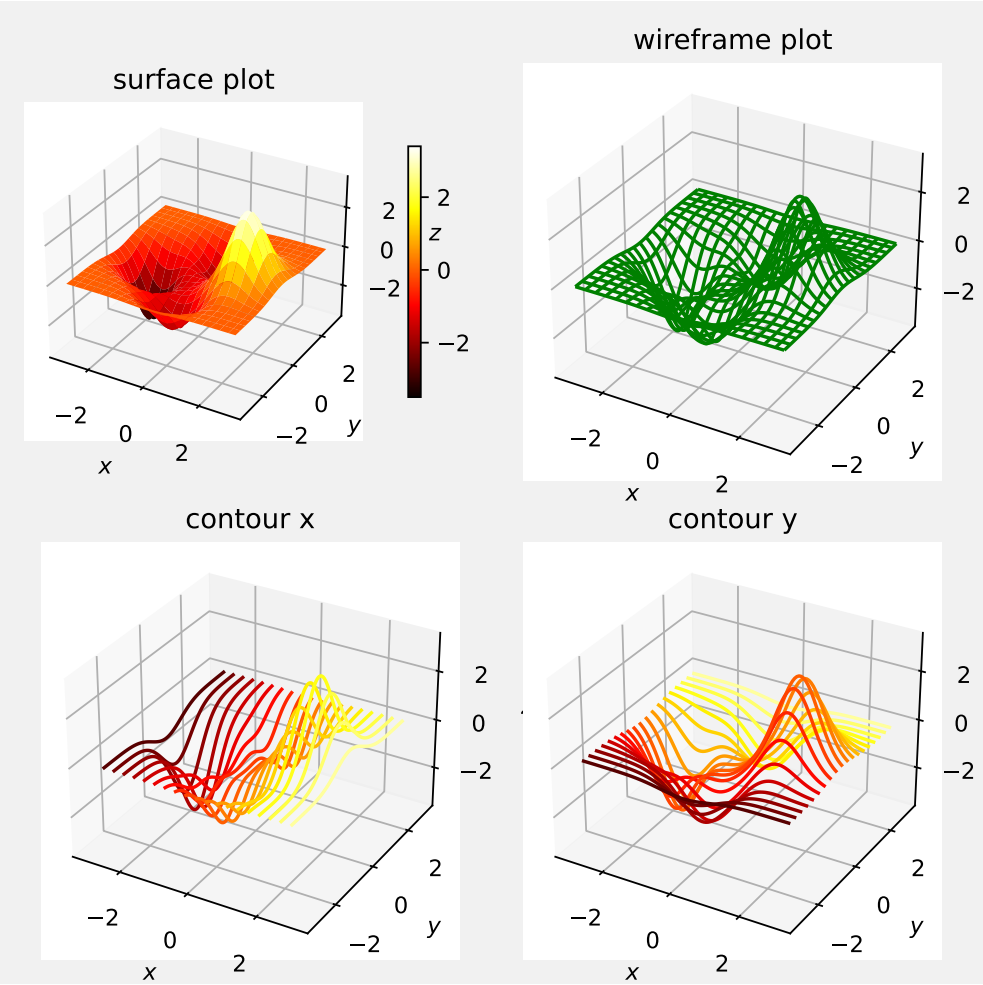


图 9.8: 表面图、线框图和三维等高线图

程序??绘制了函数 $z = (x^7 - y^6 + x^5 - y^4 + x^3 - y^2 + x - 1)e^{-x^2-y^2}$ 的三维图示 (图 9.8)。第 8 行至第 8 行创建了 Figure 和 Axes 对象，并通过 subplot_kw 参数指定绘制三维图示。第 16 行至第 21 行绘制了根据 z 值染色的表面图，在 x 轴方向和 y 轴方向上的采样次数都是 20。第 23 行至第 25 行绘制了线框图。第 27 行至第 29 行绘制了曲面上 x 坐标值相同的点连线得到的三维等高线图。第 31 行至第 33 行绘制了曲面上 y 坐标值相同的点连线得到的三维等高线图。

Listing 9.10: 表面图、线框图和三维等高线图

```
1 import numpy as np
2 import matplotlib as mpl; import matplotlib.pyplot as plt
3
4 def title_and_labels(ax, title):
5     ax.set_title(title); ax.set_xlabel("$x$")
```

```
6     ax.set_ylabel("$y$"); ax.set_zlabel("$z$")
7
8 fig, axes = plt.subplots(2, 2, figsize=(6, 6),
9                          subplot_kw={'projection': '3d'})
10
11 x = y = np.linspace(-3, 3, 100)
12 X, Y = np.meshgrid(x, y)
13 Z = (X**7 - Y**6 + X**5 - Y**4 + X**3 - Y**2 + X - 1) * \
14     np.exp(- X**2 - Y**2)
15
16 norm = mpl.colors.Normalize(vmin = Z.min(), vmax = Z.max())
17 p = axes[0, 0].plot_surface(X, Y, Z, linewidth=0, rcount=20,
18                             ccount=20, norm=norm,
19                             cmap=mpl.cm.hot)
20 cb = fig.colorbar(p, ax=axes[0, 0], pad=0.1, shrink = 0.6)
21 title_and_labels(axes[0, 0], "surface␣plot")
22
23 p = axes[0, 1].plot_wireframe(X, Y, Z, rcount=20, ccount=20,
24                               color="green")
25 title_and_labels(axes[0, 1], "wireframe␣plot")
26
27 cset = axes[1, 0].contour(X, Y, Z, zdir='x', levels = 20,
28                           norm=norm, cmap=mpl.cm.hot)
29 title_and_labels(axes[1, 0], "contour␣x")
30
31 cset = axes[1, 1].contour(X, Y, Z, zdir='y', levels = 20,
32                           norm=norm, cmap=mpl.cm.hot)
33 title_and_labels(axes[1, 1], "contour␣y")
34
35 fig.tight_layout(); plt.show()
```

9.4 二维和三维向量场

quiver 函数可绘制二维和三维向量场。程序 9.11 的第 6 行至第 11 行绘制了 Holling-Tanner 模型 [SL2018]

$$\dot{x} = x(1 - \frac{x}{7}) - \frac{6xy}{7 + 7x}, \quad \dot{y} = 0.2y(1 - \frac{0.5y}{x})$$

在一个二维网格上的每个点 (x, y) 处的向量 $(\dot{x}, \dot{y})$，即图 9.6 的左子图。第 10 行至第 11 行调用 quiver 函数的前两个参数 x 和 y 为网格上每个点的 x 和 y 坐标，第三个参数 u 和第四个参

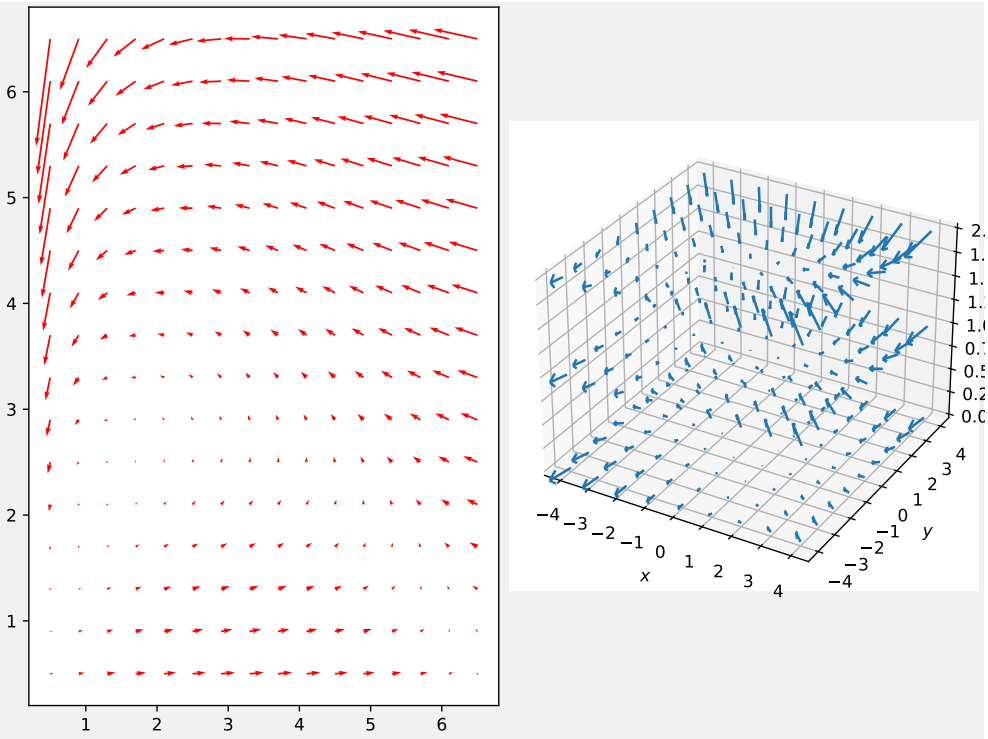


图 9.9: 二维和三维向量场

数 v 为该点的向量的两个分量，参数 angles='xy' 表示绘制的箭头从 (x, y) 指向 $(x+u, y+v)$，参数 scale_units='xy' 表示绘制向量时使用和坐标相同的尺度。第 13 行创建了绘图区域 ax2，并设置参数 projection='3d' 表示三维图示。第 21 行调用 quiver 函数绘制了三维 Lotka-Volterra 模型 [SL2018]

$$\dot{x} = x(1 - 2x + y - 5z), \quad \dot{y} = y(1 - 5x - 2y - z), \quad \dot{z} = z(1 + x - 3y - 2z)$$

在一个三维网格上的每个点 (x, y, z) 处的向量 $(\dot{x}, \dot{y}, \dot{z})$，即图 9.6 的右子图。quiver 函数的前三个参数为网格上每个点的 x、y 和 z 坐标，第四个、第五个和第六个参数依次为该点的向量的三个分量，参数 length 设置绘制的向量的长度。

Listing 9.11: **二维和三维向量场**

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 fig = plt.figure(figsize=(8, 6), constrained_layout=True)
5 gs = fig.add_gridspec(1, 2)
6
7 ax1 = fig.add_subplot(gs[0, 0])
8 x, y = np.mgrid[0.5:6.5:16j, 0.5:6.5:16j]
9 dx_dt = x*(1 - x/7) - 6*x*y/(7+7*x)
```

```
dy_dt = 0.2*y*(1 - 0.5*y/x)
ax1.quiver(x, y, dx_dt, dy_dt, color='r', angles='xy',
           scale_units='xy')

ax2 = fig.add_subplot(gs[0, 1], projection='3d')
ax2.set_xlabel("$x$"); ax2.set_ylabel("$y$");
ax2.set_zlabel("$z$")
grid1 = np.linspace(-4, 4, 9); grid2 = np.linspace(0, 2, 3)
x, y, z = np.meshgrid(grid1, grid1, grid2)
dx_dt = x * (1 - 2 * x + y - 5 * z)
dy_dt = y * (1 - 5 * x - 2 * y - z)
dz_dt = z * (1 + x - 3 * y - 2 * z)
ax2.quiver(x, y, z, dx_dt, dy_dt, dz_dt, length=0.008)
plt.show()
```

9.5 实验 9：Matplotlib 库简介

实验目的

本实验的目的是掌握使用 Matplotlib 库绘制图示。

提交方式

在 Blackboard 提交一个文本文件 (txt 后缀)，文件中记录每道题的源程序。

实验内容

1. 编写程序从一个等比数列中依次选取 n，绘制实验 5 习题 2 的三个计算数值定积分公式的计算结果和精确结果之间的相对误差随 n 变化的情况。绘图时两个坐标轴都使用对数尺度，并标注图例。

2. 绘制实验 8 习题 2 的三种排序算法的运行时间随列表长度变化的情况并标注图例。

3. 在两行两列的四个子图中依次绘制函数 $z = (3x^3 - 2x + 4x^2y + 2y^2)e^{-x^2-y^2}(-2 \le x \le 2, -2 \le y \le 2)$ 的四个图示：根据 z 值染色的三维表面图、x 坐标值相同的点连线得到的三维等高线图、y 坐标值相同的点连线得到的三维等高线图和 z 坐标值相同的点连线得到的三维等高线图。

第 10 章　SymPy 库简介

符号计算指不使用数值代入变量的情况下按照规则对数学表达式进行精确演算。符号计算和数值计算是两种不同类型的计算。例如要计算函数 $f(x) = x^3 - 2x^2 + 7x - 9$ 的导函数在 $x = 2$ 时的函数值。符号计算根据求导规则获得导函数为 $f'(x) = 3x^2 - 4x + 7$，然后将 $x = 2$ 代入求函数值 $f'(2)$。数值计算则根据数值求导公式 5.2 对 $x = 2$ 近似计算 $f'(2)$。虽然很多实际问题无法通过精确演算获得解析解 (例如一元五次方程求根)，符号计算可以用来简化问题的复杂度或缩小问题的规模。

可进行符号计算的软件称为计算机代数系统 (computer algebra systems)。在众多类似软件中，SymPy 是一个完全用 Python 实现的轻量级开源 Python 库。SageMath 是一个使用 Python 语言的庞大开源数学软件系统，包含 SymPy 在内的很多开源软件包。Maple 和 Mathematica 则是付费使用的商业软件。

本章简要介绍 SymPy[SymPyDoc] 的常用功能，包括矩阵计算、表达式展开和化简、微积分和方程求解。使用 SymPy 模块之前需要执行“import sympy as sym”将其导入，使用别名 sym 以便于输入。

10.1　符号表达式

符号表达式由符号变量、符号数值、运算符和符号函数按照规则组合而成。

符号变量在使用前需要声明。sym.Symbol 函数可声明单个符号变量，并可用关键字参数指定变量表示的数值的性质。sym.symbols 函数可同时声明多个符号变量。程序 10.1 中声明了一些符号变量：b 表示一个实数，c 和 d 分别表示一个正实数和一个负实数，e、f 和 g 都表示一个整数，j 和 k 分别表示一个奇数和一个偶数。

Listing 10.1: 符号变量

```
In[1]: a = sym.Symbol("a"); b = sym.Symbol("b", real=True)
In[2]: c = sym.Symbol("c", positive=True)
In[3]: d = sym.Symbol("d", negative=True)
In[4]: a.is_real, b.is_real, c.is_real and c.is_positive
```

```
Out[4]: (None, True, True)
In[5]: d.is_real and d.is_negative
Out[5]: True
In[6]: e, f, g = sym.symbols("e, f, g", integer=True);
In[7]: f.is_integer
Out[7]: True
In[8]: j = sym.Symbol("j", odd=True)
In[9]: k = sym.Symbol("k", even=True)
In[10]: j.is_integer and j.is_odd, k.is_integer and k.is_even
Out[10]: (True, True)
```

SymPy 的三个类 Integer、Float 和 Rational 分别表示符号数值中的整数、浮点数和有理数，并可通过函数 int 和 float 转换为 Python 中的 int 和 float 类型。SymPy 使用 mpmath 模块，可以进行精确的浮点数计算。SymPy 中定义复数时用 sym.I 表示虚部符号。

Listing 10.2: **符号数值类型**

```
In[1]: p = sym.Integer(123); (p.is_Integer, p.is_real, p.is_odd)
Out[1]: (True, True, True)
In[2]: q = sym.Float('0.4'); (q.is_Integer, q.is_real, q.is_odd)
Out[2]: (False, True, False)
In[3]: r = sym.Float('0.3'); q - r, float(q)-float(r)
Out[3]: (0.100000000000000, 0.10000000000000003)
In[4]: s = sym.Rational(6, 7); t = sym.Rational(19, 8)
In[5]: s+t, s-t, s*t, s/t
Out[5]: (181/56, -85/56, 57/28, 48/133)
In[6]: (p + sym.I * 5) * (q -sym.I * r)
Out[6]: (0.4-0.3i)(123+5i)
In[7]: sym.simplify((p + sym.I * 5) * (q -sym.I * r))
Out[7]: 50.7-34.9i
```

符号函数分为三类：sym.Function 声明的未定义的函数、SymPy 内建的常用数学函数和 sym.Lambda 定义的匿名函数 (表 10.1)。

10.2 符号表达式的变换

sym.simplify 使用各种经验规则对符号表达式进行化简 (表 10.2)，适合交互使用，但有时不能得到预期的结果。对于每一类数学函数，sympy 都定义了专门的化简函数。

sym.sympify 可从一个表示数学表达式的字符串生成一个符号表达式。subs 函数可将一个符号表达式中出现的所有指定的变量替换为一个指定的表达式 (表 10.3)，替换的结果保存在一个新的符号表达式中。subs 函数的参数是指定的变量和指定的表达式。如果需要对多个变量

输入	输出
x, y = sym.symbols(”x, y”); f = sym.Function(”f”)(x, y); f	$f(x, y)$
j = sym.Symbol(”j”, odd=True); sym.tan(j*sym.pi)	0
k = sym.Symbol(”k”, even=True); sym.cos(k*sym.pi)	1
sym.sqrt(x*x)	$\sqrt{x^2}$
d = sym.Symbol(”d”, negative=True); sym.sqrt(d*d)	$-d$
sym.log(sym.E ** sym.Float(’-23.12345678987654321’))	-23.12345678987654321
g = sym.Lambda(x, 3 * x**2 - 6 * x + 15); g	$(x \mapsto 3x^2 - 6x + 15)$
g(p), sym.simplify(g(q - sym.I * 2))	(44664, 1.08 + 7.2*I)

表 10.1: 符号函数

输入	输出
sym.simplify(sym.sin(x)**2 + sym.cos(x)**2)	1
sym.simplify((x**3 - 1)/(x**2 + x + 1))	$x - 1$
expr = sym.exp((x + sym.pi/2) * sym.I); expr	$e^{i(x+\frac{\pi}{2})}$
sym.simplify(expr)	ie^{ix}

表 10.2: sym.simplify

替换，可提供一个列表作为参数，列表中的每个元素是由变量和表达式组成的元组。

evalf 函数可计算符号表达式的数值。evalf 函数的参数 subs 是一个字典，其中的每个元素由变量和对应的取值组成。evalf 函数的参数 n 指定有效数字的位数，参数 chop 取值为 True 可以去除小于指定精度的舍入误差。

sym.lambdify 函数可将一个符号表达式 (第二个参数) 转换成为以指定的变量 (第一个参数) 作为自变量的 Python 函数以便调用，并通过第三个参数指定表达式中的数学函数由哪个数学库提供。

输入	输出
expr1 = sym.sympify('sin(x)-tan(x)'); expr1	$\sin(x) - \tan(x)$
expr1.subs(x, sym.pi/3)	$-\frac{\sqrt{3}}{2}$
expr = x*y + 2*y*z - 3*z*x + 9; expr	$xy - 3xz + 2yz + 9$
expr2 = expr.subs([(x, 2*u+v), (y, 3*w-u), (z, w)]); expr2	$2w(-u+3w) - 3w(2u+v) + (-u+3w)(2u+v) + 9$
expr3 = sym.simplify(expr2); expr3	$-2u^2 - uv - 2uw + 6w^2 + 9$
expr3.evalf(subs={u: 2.4, v: sym.pi, w: sym.E})	21.226761448365
expr3.evalf(subs={u: 2.4, v: sym.pi, w: sym.E}, n = 25)	21.22676144836498207588746
expr = sym.cos(sym.E)**2 + sym.sin(sym.E)**2; expr	$\sin^2(e) + \cos^2(e)$
(expr - 1).evalf(), (expr - 1).evalf(chop=True)	(-0.e-125, 0)
f = sym.Function("f"); f = sym.lambdify([u, v, w], expr3)	
f(1,2,3)	53
h = sym.lambdify(x, expr1, 'numpy')	
import numpy as np; h(np.pi/3)	-0.8660254037844382

表 10.3: sym.sympify 函数、subs 函数、evalf 函数和 sym.lambdify 函数

10.2.1 幂函数的变换 (表 10.4)

幂函数的变换涉及三个公式。公式 $x^a x^b = x^{a+b}$ 总是成立。公式 $x^a y^a = (xy)^a$ 成立的一个充分条件是 x 和 y 都是非负实数并且 a 是实数，不成立的一个反例是 $x = y = -1, a = 1/2$。公式 $(x^a)^b = x^{ab}$ 成立的一个充分条件是 x 是非负实数并且 a 和 b 都是实数，成立的另一个充分条件是 b 是整数，不成立的一个反例是 $x = -1, a = 2, b = 1/2$。在确定公式成立的前提下，sym.powsimp 从左向右使用这三个公式，sym.expand_power_exp、sym.expand_power_base 和 sym.powdenest 从右向左使用这三个公式。

输入	输出
x, y = sym.symbols('x y', positive=True)	
a, b = sym.symbols('a b', real=True)	
u, v, c, d = sym.symbols('u, v, c, d')	
sym.powsimp(u**c * u**d), sym.powsimp(u**c * v**c), sym.powsimp((u**c)**d)	$(u^{c+d}, u^c v^c, (u^c)^d)$
sym.powsimp(x**a * x**b), sym.powsimp(x**a * y**a), sym.powsimp((x**a)**b)	$(x^{a+b}, (xy)^a, x^{ab})$
n = sym.symbols("n", integer=True); sym.powsimp(u**n * v**n)	$(uv)^n$
sym.expand_power_exp(x**(a + b)), sym.expand_power_exp(u**(c + d))	$(x^a x^b, u^c u^d)$
sym.expand_power_base((x*y)**a), sym.expand_power_base((u*v)**c)	$(x^a y^a, (uv)^c)$
sym.powdenest((x**a)**b), sym.powdenest((u**a)**b)	$(x^{ab}, (u^a)^b)$

表 10.4: 幂函数的变换

10.2.2 对数函数的变换 (表 10.5)

对数函数的变换涉及两个公式。公式 $\log(x*y) = logx + logy$ 成立的一个充分条件是 x 和 y 都是正实数。公式 $\log x^a = a\log x$ 成立的一个充分条件是 x 是正实数并且 a 是实数。在确定公式成立的前提下，sym.expand_log 从左向右使用这两个公式，sym.logcombine 从右向左使用这两个公式。

输入	输出
x, y = sym.symbols('x y', positive=True)	
a = sym.symbols('a', real=True)	
u, v, c = sym.symbols('u, v, c')	
sym.expand_log(sym.log(x*y)), sym.expand_log(sym.log(u*v))	$(\log(x) + \log(y), \log(uv))$
sym.expand_log(sym.log(x**a)), sym.expand_log(sym.log(u**c))	$(a\log(x), \log(u^c)$
sym.logcombine(sym.log(x) + sym.log(y)), sym.logcombine(sym.log(u) + sym.log(v)))	$(\log(xy), \log(u) + \log(v))$
sym.logcombine(a*sym.log(x)), sym.logcombine(c*sym.log(u))	$(\log(x^a), c\log(u))$

表 10.5: 对数函数的变换

10.2.3 三角函数和双曲函数的变换 (表 10.6)

sym.trigsimp 对三角函数和双曲函数进行化简。sym.expand_trig 对三角函数和双曲函数进行展开。如果只需要对一个表达式中的某些项 (而非所有项) 进行操作，可以使用 subs 函数。

输入	输出
sym.trigsimp(sym.cos(x-y)-sym.cos(x+y))	$2\sin(x)\sin(y)$
sym.trigsimp(4 * sym.cos(x)**3 - 3 * sym.cos(x))	$\cos(3x)$
expr = sym.sinh(x)*sym.cosh(y) - sym.sinh(y)*sym.cosh(x); expr	$\sinh(x)\cosh(y) - \sinh(y)\cosh(x)$
sym.trigsimp(expr)	$\sinh(x-y)$
sym.expand_trig(sym.tan(x + y))	$\frac{\tan(x)+\tan(y)}{-\tan(x)\tan(y)+1}$
expr = sym.cos((x+y)/2) * sym.sin((x-y)/2); expr	$\sin\left(\frac{x}{2}-\frac{y}{2}\right)\cos\left(\frac{x}{2}+\frac{y}{2}\right)$
sym.trigsimp(sym.expand_trig(expr))	$\frac{\sin(x)}{2}-\frac{\sin(y)}{2}$
sym.expand_trig(sym.cosh(x - y))	$-\sinh(x)\sinh(y)+\cosh(x)\cosh(y)$
expr = sym.sin(2*x) - sym.cos(3*x); sym.expand_trig(expr)	$2\sin(x)\cos(x)-4\cos^3(x)+3\cos(x)$
expr.subs(sym.sin(2*x), 2*sym.sin(x)*sym.cos(x))	$2\sin(x)\cos(x)-\cos(3x)$

表 10.6: 三角函数和双曲函数的变换

10.2.4 有理系数多项式函数的变换 (表 10.7)

sym.expand 将多项式函数展开成各单项的和 。sym.factor 对多项式函数进行因式分解。sym.collect 将多元多项式函数整理成指定变量的多项式函数，coeff 根据指定指数提取该项的系数。对于分子和分母均为多项式的分式，sym.cancel 消去分子和分母的公因子。sym.apart 实现了部分分式分解。

10.2.5 特殊函数的变换 (表 10.8)

sympy 定义了大量特殊函数，应用领域包括数学物理方程和组合数学等。sym.rewrite 可将当前函数表示成为实参指定的函数的表达式。sym.expand_func 可展开函数。sym.combsimp 可简化有组合函数的表达式。sym.gammasimp 可简化有伽马函数或组合函数的表达式。

输入	输出
sym.expand((x - 2) ** 3)	$x^3 - 6x^2 + 12x - 8$
sym.expand((x + 6)*(x - 8))	$x^2 - 2x - 48$
sym.expand((x + 4)*(x - 5) - (x + 21)*(x - 22))	442
expr = (x - sym.Rational(2, 3)) ** 2 * (y + sym.Rational(6, 7)); expr	$\left(x - \frac{2}{3}\right)^2 \left(y + \frac{6}{7}\right)$
sym.expand(expr)	$x^2y + \frac{6x^2}{7} - \frac{4xy}{3} - \frac{8x}{7} + \frac{4y}{9} + \frac{8}{21}$
sym.factor(x**2*z - 5*x*z*y + 6*y**2*z)	$z\left(x - 3y\right)\left(x - 2y\right)$
expr = x**2*y - sym.Rational(8, 7) * x + sym.Rational(8, 21) + y * sym.Rational(4, 9) - sym.Rational(4, 3) * x * y + sym.Rational(6, 7) * x * x)	
expr	$x^2y + \frac{6x^2}{7} - \frac{4xy}{3} - \frac{8x}{7} + \frac{4y}{9} + \frac{8}{21}$
sym.factor(expr)	$\frac{(3x-2)^2(7y+6)}{63}$
expr = 15 - 3*x**2 + 6*x*y*z - 8*x*y*2 + x**3*y collected_expr = sym.collect(expr, x); collected_expr	$x^3y - 3x^2 + x\left(6yz - 16y\right) + 15$
collected_expr.coeff(x, 1)	$6yz - 16y$
expr = (x**3*y - x**3 + 3*x**2*y - 3*x**2 - 10*x*y + 10*x - 24*y + 24)/(x**2*y - x**2 + x*y - x - 2*y + 2)	
expr	$\frac{x^3y-x^3+3x^2y-3x^2-10xy+10x-24y+24}{x^2y-x^2+xy-x-2y+2}$
sym.cancel(expr)	$\frac{x^2+x-12}{x-1}$
expr = (-64*x**4 + 176*x**3 + 439*x**2 - 542*x - 615)/(56*x**4 + 38*x**3 - 312*x**2 - 171*x + 270); expr	$\frac{-64x^4+176x^3+439x^2-542x-615}{56x^4+38x^3-312x^2-171x+270}$
sym.apart(expr)	$\frac{3x-2}{2x^2-9} - \frac{8}{7} + \frac{45}{7(7x+10)} + \frac{6}{4x-3}$

表 10.7: 有理系数多项式的变换

输入	输出
x, y, z = sym.symbols('x y z')	
n, k = sym.symbols('n k', integer = True)	
sym.tan(x).rewrite(sym.cos)	$\frac{\cos\left(x-\frac{\pi}{2}\right)}{\cos\left(x\right)}$
sym.factorial(z).rewrite(sym.gamma)	$\Gamma\left(z+1\right)$
sym.expand_func(sym.gamma(y + 4))	$y\left(y+1\right)\left(y+2\right)\left(y+3\right)\Gamma\left(y\right)$
sym.combsimp(sym.factorial(k)/sym.factorial(k - 4))	$k\left(k-3\right)\left(k-2\right)\left(k-1\right)$
sym.combsimp(sym.binomial(n+2, k+2)/sym.binomial(n, k))	$\frac{(n+1)(n+2)}{(k+1)(k+2)}$
sym.tan(x).rewrite(sym.cos)	$\frac{(n+1)(n+2)}{(k+1)(k+2)}$
sym.gammasimp(sym.gamma(y)*sym.gamma(2 - y))	$-\frac{\pi(y-1)}{\sin\left(\pi y\right)}$

表 10.8: 特殊函数的变换

10.3 微积分

10.3.1 求导数 (表 10.9)

sym.diff 可计算单变量函数或多变量函数的导数。它的第一个参数是要求导的函数，后面的一个或多个参数为依次求导的自变量序列。如果对同一个自变量求多次导数，也可以在自变量参数的后面用数字表示求导的次数。sym.Derivative 生成导数的未求值形式，可调用 doit 函数计算导数。

输入	输出
expr = x**4 * sym.E**x; expr	x^4e^x
sym.diff(expr, x, 3)	$x\left(x^3+12x^2+36x+24\right)e^x$
sym.diff(expr, x, x, x)	$x\left(x^3+12x^2+36x+24\right)e^x$
expr = sym.E**(x**2 * y**3) * (2 - x - z); expr	$\left(-x-z+2\right)e^{x^2y^3}$
sym.diff(expr, y, 2, x, z)	$-6xy\left(3x^4y^6+8x^2y^3+2\right)e^{x^2y^3}$
deriv = sym.Derivative(expr, z, y, 3, x); deriv	$\frac{\partial^5}{\partial x\partial y^3\partial z}\left(-x-z+2\right)e^{x^2y^3}$
deriv.doit()	$-6x\left(9x^6y^9+45x^4y^6+38x^2y^3+2\right)e^{x^2y^3}$

表 10.9: 求导数

10.3.2 求积分 (表 10.10)

sym.integrate 可计算单变量或多变量函数的定积分和不定积分。计算定积分需要提供自变量的积分范围。sym.Integral 生成积分的未求值形式，可调用 doit 函数计算积分。sym.oo 表示正无穷。对于无法计算的积分，sympy 返回其未求值形式。

10.3.3 求极限 (表 10.11)

sym.limit(f(x), x, x0) 计算极限 $\lim_{x\to x0} f(x)$。要计算单侧极限，添加第四个参数'+' 或'-'。sym.Limit 生成极限的未求值形式，可调用 doit 函数计算极限。

10.3.4 泰勒级数展开 (表 10.12)

series 函数可将一个函数在指定的自变量值 (第一个参数) 处进行泰勒级数展开到指定的阶数 (第二个参数)。removeO 函数可去除 series 函数的运行结果中的高阶无穷小项。

输入	输出
sym.integrate(sym.tan(x), x)	$-\log\left(\cos\left(x\right)\right)$
sym.integrate(sym.cos(x**2), x)	$\frac{\sqrt{2}\sqrt{\pi}C\left(\frac{\sqrt{2}x}{\sqrt{\pi}}\right)\Gamma\left(\frac{1}{4}\right)}{8\Gamma\left(\frac{5}{4}\right)}$
sym.integrate(x**x, x)	$\int x^{x}\,dx$
expr = sym.Integral(sym.exp(-x**2), (x, 0, sym.oo)); expr	$\int\limits_{0}^{\infty} e^{-x^{2}}\,dx$
expr.doit()	$\frac{\sqrt{\pi}}{2}$
expr = sym.Integral(x*sym.log(x), x); expr	$\int x\log\left(x\right)dx$
expr.doit()	$\frac{x^{2}\log\left(x\right)}{2} - \frac{x^{2}}{4}$
expr = sym.Integral(sym.exp(-x**2 - y**4), (x, -sym.oo, 0), (y, 0, sym.oo)); expr	$\int\limits_{0}^{\infty}\int\limits_{-\infty}^{0} e^{-x^{2}-y^{4}}\,dx\,dy$
expr.doit()	$\frac{\sqrt{\pi}\Gamma\left(\frac{1}{4}\right)}{8}$
_.evalf(20)	0.80327828046363949031
expr = sym.Integral(x**y*sym.exp(-x*x), (x, 0, sym.oo)); expr	$\int\limits_{0}^{\infty} x^{y}e^{-x^{2}}\,dx$
expr.doit()	$\begin{cases} \frac{\Gamma\left(\frac{y}{2}+\frac{1}{2}\right)}{2} & \text{for } \frac{\operatorname{re}\left(y\right)}{2} - \frac{1}{2} > -1 \\ \int\limits_{0}^{\infty} x^{y}e^{-x^{2}}\,dx & \text{otherwise} \end{cases}$

表 10.10: 求积分

输入	输出
sym.limit(sym.tan(x)/x, x, 0)	1
sym.limit(x**3/1.03**x, x, sym.oo)	0
expr = sym.Limit(x*(x-2)/sym.Abs(x-2), x, 2, '+'); expr	$\lim_{x\to 2^{+}}\left(\frac{x\left(x-2\right)}{\left\lvert x-2\right\rvert}\right)$
expr.doit()	2
expr = sym.Limit(x*(x-2)/sym.Abs(x-2), x, 2, '-'); expr	$\lim_{x\to 2^{-}}\left(\frac{x\left(x-2\right)}{\left\lvert x-2\right\rvert}\right)$
expr.doit()	-2

表 10.11: 求极限

In[1]:	expr = sym.tan(x); expr.series(x, 2, 6)
Out[1]:	$\tan(2) + \left(1+\tan^2(2)\right)(x-2) + (x-2)^2\left(\tan^3(2)+\tan(2)\right) + (x-2)^3\left(\frac{1}{3}+\frac{4\tan^2(2)}{3}+\tan^4(2)\right) + (x-2)^4\left(\tan^5(2)+\frac{5\tan^3(2)}{3}+\frac{2\tan(2)}{3}\right) + (x-2)^5\left(\frac{2}{15}+\frac{17\tan^2(2)}{15}+2\tan^4(2)+\tan^6(2)\right) + O\left((x-2)^6; x\to 2\right)$
In[2]:	expr.series(x, 2, 6).removeO()
Out[2]:	$(x-2)^5\left(\frac{2}{15}+\frac{17\tan^2(2)}{15}+2\tan^4(2)+\tan^6(2)\right) + (x-2)^4\left(\tan^5(2)+\frac{5\tan^3(2)}{3}+\frac{2\tan(2)}{3}\right) + (x-2)^3\left(\frac{1}{3}+\frac{4\tan^2(2)}{3}+\tan^4(2)\right) + (x-2)^2\left(\tan^3(2)+\tan(2)\right) + \left(1+\tan^2(2)\right)(x-2) + \tan(2)$
In[3]:	expr = sym.exp(sym.sin(x)*sym.cos(y)); expr.series(x, 0, 4)
Out[3]:	$1 + x\cos(y) + \frac{x^2\cos^2(y)}{2} + x^3\left(\frac{\cos^3(y)}{6}-\frac{\cos(y)}{6}\right) + O(x^4)$

表 10.12: 泰勒级数展开

10.4 矩阵

SymPy.Matrix 表示一个矩阵。创建一个矩阵的方式有多种 (表 10.13)。给定一个嵌套列表，SymPy.Matrix 生成一个矩阵，嵌套列表中的每个子列表构成一个行向量。给定一个非嵌套列表，SymPy.Matrix 生成一个列向量。eye(n) 生成一个 $n \times n$ 的单位阵。zeros(n, m) 生成一个 $n \times m$ 的所有元素为 0 的矩阵。ones(n, m) 生成一个 $n \times m$ 的所有元素为 1 的矩阵。diag 生成一个对角阵，它的参数可以是数值或矩阵，它们沿对角方向堆叠。

sym.shape 函数返回一个矩阵的 shape 属性，即由行数和列数组成的元组。row 函数可获取指定的行。col 函数可获取指定的列。row_del 可删除指定的行。col_del 可删除指定的列。row_insert 可在指定的位置插入一行并生成一个新矩阵。col_insert 可在指定的位置插入一列并生成一个新矩阵。

单个矩阵可进行转置和乘方运算。两个矩阵可进行加法、减法和乘法运算。det 函数可获取矩阵的行列式。rref 函数将一个矩阵转换成简化行阶梯形式 (reduced row echelon form)，返回值是一个元组，由转换结果和一个存储了主元列的索引值的元组构成。

nullspace 函数返回零空间的基向量。columnspace 函数返回列空间的基向量。eigenvals 函数返回一个字典，其中每个元素由一个特征值和其对应的代数重数组成。eigenvects 返回一个列表，其中的每个元素是一个元组，由一个特征值、对应的代数重数和特征向量的列表组成。charpoly 函数返回一个以指定变量为自变量的特征多项式。

输入	输出
sym.Matrix([[1, 2], [-3, -4], [5, 6]])	$\begin{bmatrix} 1 & 2 \\ -3 & -4 \\ 5 & 6 \end{bmatrix}$
sym.Matrix([7, -8, 9])	$\begin{bmatrix} 7 \\ -8 \\ 9 \end{bmatrix}$
sym.eye(3)	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
sym.zeros(2, 3)	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$
sym.ones(3, 2)	$\begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}$
sym.diag(2, -3, 6)	$\begin{bmatrix} 2 & 0 & 0 \\ 0 & -3 & 0 \\ 0 & 0 & 6 \end{bmatrix}$
sym.diag(sym.Matrix([7, -8, 9]), 24, sym.ones(2, 2))	$\begin{bmatrix} 7 & 0 & 0 & 0 \\ -8 & 0 & 0 & 0 \\ 9 & 0 & 0 & 0 \\ 0 & 24 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$

表 10.13: 创建矩阵

输入	输出
m = sym.Matrix([[1, 2], [-3, -4], [5, 6]]); m	$\begin{bmatrix} 1 & 2 \\ -3 & -4 \\ 5 & 6 \end{bmatrix}$
sym.shape(m), m.shape	((3, 2), (3, 2))
m.row(1)	$\begin{bmatrix} -3 & -4 \end{bmatrix}$
m.col(-2)	$\begin{bmatrix} 1 \\ -3 \\ 5 \end{bmatrix}$
m.row_del(1); m	$\begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix}$
m.col_del(-2); m	$\begin{bmatrix} 2 \\ 6 \end{bmatrix}$
m = m.row_insert(1, sym.Matrix([[7]])); m	$\begin{bmatrix} 2 \\ 7 \\ 6 \end{bmatrix}$
m = m.col_insert(-2, sym.Matrix([8, -9, 10])); m	$\begin{bmatrix} 8 & 2 \\ -9 & 7 \\ 10 & 6 \end{bmatrix}$
p = sym.Matrix([[6, -5], [4, -3], [1, 2]]); p	$\begin{bmatrix} 6 & -5 \\ 4 & -3 \\ 1 & 2 \end{bmatrix}$
m+p	$\begin{bmatrix} 14 & -3 \\ -5 & 4 \\ 11 & 8 \end{bmatrix}$
m-p	$\begin{bmatrix} 2 & 7 \\ -13 & 10 \\ 9 & 4 \end{bmatrix}$
q = p.T; q	$\begin{bmatrix} 6 & 4 & 1 \\ -5 & -3 & 2 \end{bmatrix}$
s = m * q; s	$\begin{bmatrix} 38 & 26 & 12 \\ -89 & -57 & 5 \\ 30 & 22 & 22 \end{bmatrix}$

表 10.14: 矩阵的基本运算

输入	输出
t = s - 3 * sym.ones(3, 3); t	$\begin{bmatrix} 35 & 23 & 9 \\ -92 & -60 & 2 \\ 27 & 19 & 19 \end{bmatrix}$
s.det(), t.det()	(0, -936)
t ** (-1)	$\begin{bmatrix} \frac{589}{468} & \frac{133}{468} & -\frac{293}{468} \\ -\frac{901}{468} & -\frac{211}{468} & \frac{449}{468} \\ \frac{16}{117} & \frac{11}{234} & -\frac{2}{117} \end{bmatrix}$
t ** 3	$\begin{bmatrix} 28852 & 19444 & 3468 \\ -82204 & -54468 & 6940 \\ 21720 & 15008 & 9272 \end{bmatrix}$
s ** (-1)	NonInvertibleMatrixError: Matrix det == 0; not invertible.
v = s.rref(); v[0]	$\begin{bmatrix} 1 & 0 & -\frac{11}{2} \\ 0 & 1 & \frac{17}{2} \\ 0 & 0 & 0 \end{bmatrix}$
v[1]	$(0, 1)$
w = sym.Matrix([[1, -2, 3, 4], [8, 6, 4, 9]]);	$\begin{bmatrix} 1 & -2 & 3 & 0 & 0 \\ 8 & 6 & 0 & 0 & 9 \end{bmatrix}$
w.nullspace()	[Matrix([[-13/11], [10/11], [1], [0]]), Matrix([[-21/11], [23/22], [0], [1]])]
w.columnspace()	[Matrix([[1], [8]]), Matrix([[-2], [6]])]
x = sym.Matrix([[-1, 0, 1, 2], [0, 1, 0, 1], [-1, -4, 1, -2], [0, 1, 0, 1]]); x	$\begin{bmatrix} -1 & 0 & 1 & 2 \\ 0 & 1 & 0 & 1 \\ -1 & -4 & 1 & -2 \\ 0 & 1 & 0 & 1 \end{bmatrix}$
x.eigenvals()	{2: 1, 0: 3}
x.eigenvects()	[(0, 3, [Matrix([[1], [0], [1], [0]]), Matrix([[2], [-1], [0], [1]])]), (2, 1, [Matrix([[-1], [1], [-5], [1]])])]
lamda = sym.symbols('lamda')	λ
p = x.charpoly(lamda); p	$\mathrm{PurePoly}\left(\lambda^4 - 2\lambda^3, \lambda, domain = \mathbb{Z}\right)$
sym.factor(p.as_expr())	$\lambda^3 (\lambda - 2)$

表 10.15: 矩阵的基本运算

10.5 方程求解

10.5.1 代数方程求解

线性方程组求解 (表 10.16)

linsolve 函数求解线性方程组。可以使用多种方式描述方程组：方程的列表、增广矩阵和 $Ax = b$ 形式。以求解以下线性方程组为例。

$$\begin{bmatrix} 1 & -2 & 3 \\ 4 & -5 & 6 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ 8 \end{bmatrix} \tag{10.1}$$

输入	输出
sym.linsolve([x - 2*y + 3*z - 4, 5*x - 6*y + 7*z - 8], (x, y, z))	$\{(z-2,\ 2z-3,\ z)\}$
sym.linsolve(sym.Matrix(([1, -2, 3, 4], [5, -6, 7, 8])), (x, y, z))	$\{(z-2,\ 2z-3,\ z)\}$
m = sym.Matrix(([1, -2, 3, 4], [5, -6, 7, 8])) system = A, b = m[:, :-1], m[:, -1] sym.linsolve(system, x, y, z)	$\{(z-2,\ 2z-3,\ z)\}$

表 10.16: 线性方程组求解

非线性方程求解 (表 10.17)

solveset 函数对于给定的变量 (第二个参数) 求解非线性方程 (第一个参数)，可通过 domain 参数指定根的取值范围。如果没有解，返回空集。如果无法求解，返回一个条件集合。对于一元多项式方程，sym.roots 返回一个字典，其中的每个元素包括一个根和其对应的重数。

输入	输出
sym.solveset(x**2 - 2*x + 2, x)	$\{1-i, 1+i\}$
sym.solveset(x**2 - 2*x + 2, x, domain=sym.S.Reals)	$\emptyset$
sym.solveset(sym.tan(x) - 1, x)	$\left\{2n\pi + \frac{5\pi}{4} \mid n \in \mathbb{Z}\right\} \cup \left\{2n\pi + \frac{\pi}{4} \mid n \in \mathbb{Z}\right\}$
sym.solveset(sym.sin(x) - x, x)	$\{x \mid x \in \mathbb{C} \wedge -x + \sin(x) = 0\}$
sym.solveset(x**5 - 4*x**4 + 14*x**2 - 17*x + 6)	$\{-2, 1, 3\}$
sym.roots(x**5 - 4*x**4 + 14*x**2 - 17*x + 6)	{3: 1, -2: 1, 1: 3}

表 10.17: 非线性方程求解

非线性方程组求解 (表 10.18)

nonlinsolve 函数可求解规模较小的非线性方程组，它的第一个参数是组成非线性方程组的各方程的列表，第二个参数是要求解的变量的列表。

In[1]:	sym.nonlinsolve([x*x + y*y - 2, x*y + 1], x, y)
Out[1]:	$\{(-1,\ 1), (1,\ -1)\}$
In[2]:	sym.nonlinsolve([x*x + y*y + 2, x*y + 1], x, y)
Out[2]	$\{(-i,\ -i), (i,\ i)\}$
In[3]:	system = [x**2 - 3*y**2 - 4, x*y + 2]; vars = [x, y]; sym.nonlinsolve(system, vars)
Out[3]:	$\left\{\left(-\sqrt{6},\ \frac{\sqrt{6}}{3}\right), \left(\sqrt{6},\ -\frac{\sqrt{6}}{3}\right), \left(-\sqrt{2}i,\ -\sqrt{2}i\right), \left(\sqrt{2}i,\ \sqrt{2}i\right)\right\}$
In[4]:	system = [sym.exp(x) + sym.cos(y), y*y - 3*y + 2]; sym.nonlinsolve(system, vars)
Out[4]:	$\{(\{i(2n\pi + \pi) + \log(\cos(1)) \mid n \in \mathbb{Z}\},\ 1), (\{2ni\pi + \log(-\cos(2)) \mid n \in \mathbb{Z}\},\ 2)\}$
In[5]:	system = [x*x + y*y, z*z + y - 1]; vars = [x, y]; sym.nonlinsolve(system, vars)
Out[5]:	$\{(-i(z-1)(z+1),\ -(z-1)(z+1)), (i(z-1)(z+1),\ -(z-1)(z+1))\}$

表 10.18: 非线性方程组求解

10.5.2 常微分方程求解

牛顿冷却定律 (Newton's law of cooling) 描述了温度高于周围环境的物体向环境传递热量逐渐冷却时所遵循的规律，即物体温度降低的速率正比于物体和环境的温差：$\frac{dT(t)}{dt} = -K(T(t) - T_s)$。公式中的 $T(t)$ 为要求解的物体温度随时间变化的函数，K 是一个常数，T_s 是环境温度。

dsolve 函数可求解常微分方程 (表 10.19)，其参数为方程的 SymPy 表示，方程中要求解的函数用 sym.Function 声明。求解这一方程的结果包括一个积分常数 C_1，需要通过代入初值条件 (物体的初始温度 $T(0) = Tb$) 确定。

输入	输出
t, K, Tb, Ts = sym.symbols("t, K, Tb, Ts")	
T = sym.Function("T")	
ode = T(t).diff(t) + K*(T(t) - Ts); sym.Eq(ode, 0)	$K\left(-Ts + T(t)\right) + \frac{d}{dt}T(t) = 0$
ode_sol = sym.dsolve(ode); ode_sol	$T(t) = C_1 e^{-Kt} + Ts$
ics = {T(0): Tb}; C_eq = ode_sol.subs(t, 0).subs(ics); C_eq	$Tb = C_1 + Ts$
C = sym.solve(C_eq); C	$[C1 : Tb - Ts]$
ode_sol.subs(C[0])	$T(t) = Ts + (Tb - Ts)\, e^{-Kt}$

表 10.19: 牛顿冷却定律

由于阻尼作用 (例如空气阻力)，阻尼谐振子的振幅不断缩小。设阻尼力为 $-b\frac{dx}{dt}$，则描述阻尼谐振子的位移 x 随时间 t 变化的常微分方程为 $m\frac{d^2}{dx^2}x(t) + b\frac{dx}{dt}x(t) + kx = 0$，其中 m、b 和 k 均为常数。设初始条件为 $x(0) = 1$ 和 $\frac{dx}{dt}x(0) = 0$，程序 10.3 求解方程得到的结果是 $x(t) = \left(-\frac{b}{2\sqrt{b^2-4km}} + \frac{1}{2}\right) e^{\frac{t\left(-b-\sqrt{b^2-4km}\right)}{2m}} + \left(\frac{b}{2\sqrt{b^2-4km}} + \frac{1}{2}\right) e^{\frac{t\left(-b+\sqrt{b^2-4km}\right)}{2m}}$。第 4 行至第 17 行的 solve_c 函数自动将所有初值条件代入微分方程的解以便确定所有积分常数 [RJ2019]。

为了直观显示 $x(t)$ 对于 m、b 和 k 的依赖关系，固定 $k = 9$ 和 $m = 1$，选取几个不同的 b 值绘制 $x(t)$ 随 t 变化的函数图像 (图 10.1)。当 $b^2 < 4km$ 时，振荡频率接近于无阻尼时的振荡频率 $\sqrt{\frac{k}{m}}$。当 $b^2 = 4km$ 时，即 $b = 6.0$ 时，不发生振荡。此时不能像第 35 行那样直接将 b 的值代入 $x(t)$，而是需要在第 38 行求极限。

Listing 10.3: **阻尼谐振子**

```
1 import numpy as np; import sympy as sym
2 import matplotlib.pyplot as plt
3
4 def solve_c(sol, ics, x, known_params):
5     """
6     Apply the initial conditions (ics), given as a dictionary
```

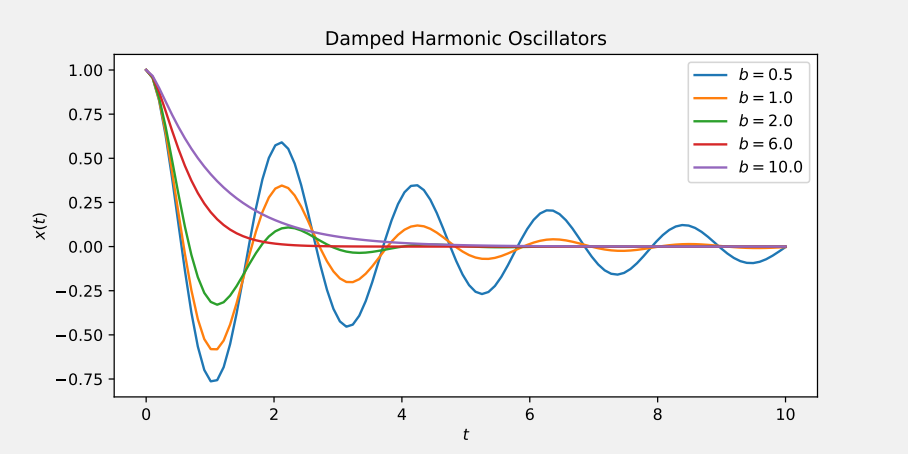


图 10.1: 阻尼谐振子

```
    of the form
    ics = {y(0): y0, y(x).diff(x).subs(x, 0): yp0, ...},
    to the solution of the ODE with independent variable x.
    The undetermined integration constants C1, C2, ... are
    extracted from the free symbols of the ODE solution,
    excluding symbols in the known_params list.
    """
    free_params = sol.free_symbols - set(known_params)
    eqs = [(sol.lhs.diff(x, n) - sol.rhs.diff(x, n))
           .subs(x, 0).subs(ics) for n in range(len(ics))]
    sol_params = sym.solve(eqs, free_params)
    return sol.subs(sol_params)

t, m, b, k = sym.symbols("t, m, b, k", positive=True)
x = sym.Function("x")
ode = m * x(t).diff(t, 2) + b * x(t).diff(t) + k * x(t)
sym.Eq(ode, 0)
ode_sol = sym.dsolve(ode); ode_sol
ics = {x(0): 1, x(t).diff(t).subs(t, 0): 0}; ics

x_t_sol = solve_c(ode_sol, ics, t, [m, b, k]); x_t_sol
x_t_sol_k_m = x_t_sol.rhs.subs({k:9, m:1})

fig, ax = plt.subplots(figsize=(8, 4));
tt = np.linspace(0, 10, 100)
```

```
bs = [0.5, 1, 2.0, 6.0, 10.0]

for i in range(len(bs)):
    if i != 3:
        x_t = sym.lambdify(t, x_t_sol_k_m.subs({b: bs[i]}),
                           'numpy')
    else:
        x_t = sym.lambdify(t, sym.limit(x_t_sol_k_m, b, 6.0),
                           'numpy')
    ax.plot(tt, x_t(tt).real, label=r"$b␣=␣%.1f$" % bs[i])

ax.set_xlabel(r"$t$"); ax.set_ylabel(r"$x(t)$"); ax.legend()
plt.title('Damped␣Harmonic␣Oscillators')
```

10.6 实验 10：SymPy 库简介

实验目的

本实验的目的是掌握使用 SymPy 库进行矩阵计算和方程求解。

提交方式

在 Blackboard 提交一个文本文件 (txt 后缀)，文件中记录每道题的源程序和运行结果。

实验内容

1. 计算矩阵 $\begin{bmatrix} 3 & -2 & a & a \\ 0 & 2 & b & b \\ -5 & 2 & -9 & -7 \\ a & -2 & a & 3 \end{bmatrix}$ 的逆矩阵、特征值和特征向量。其中 a 和 b 都是符号变量。

2. 求解以下线性方程组

$$\begin{pmatrix} 3 & -2 & a & a \\ 0 & 2 & b & b \\ -5 & 2 & -9 & -7 \\ a & -2 & a & 3 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} 1 \\ -2 \\ 3 \\ c \end{pmatrix}$$

其中 a、b 和 c 都是符号变量。

3. 以初值条件 $f(0) = 16$ 和 $f'(0) = 18$ 求解常微分方程 $f''(x) - 2f'(x) + 3f(x) - 4 = 0$。

第 11 章　SciPy 库简介

本章简要介绍使用 scipy 库实现以下功能：

- 使用 scipy.interpolate 模块进行一元插值和二元插值；
- 使用 scipy.integrate 模块计算数值积分；
- 使用 scipy.optimize 模块求解非线性方程和非线性方程组；
- 使用 scipy.integrate 模块求解常微分方程的初值问题；
- 使用 scipy.optimize 模块和 scipy.integrate 模块求解常微分方程的边值问题。

11.1　插值和数值积分

11.1.1　一元插值：多项式函数和样条函数

给定一些离散数据点，插值方法求解一个连续函数通过这些点，求解得到的函数称为插值函数 (interpolant)。插值的应用领域包括绘制平滑曲线和曲面、估算函数值、求解非线性方程和数值积分等。scipy.interpolate 模块提供了多个实现插值方法的类。

本节介绍使用多项式函数和样条函数进行一元插值。给定二维平面上的互不相同的 n 个点 $\{(x_i, y_i)\}_{i=1}^n$，拉格朗日插值公式可以计算以下的多项式插值函数：

$$f(x) = \sum_{i=1}^{n} y_i L_i(x), \quad L_i(x) = \prod_{j=1, j\neq i}^{n} (\frac{x - x_j}{x_i - x_j}),\ i = 1, 2, ..., n$$

也可定义一个次数为 $n-1$ 的多项式函数 $p(x) = a_0 + a_1 x + ... + a_{n-1}x^{n-1}$ 作为插值函数，然后求解线性方程组 $\{p(x_i) = y_i\}_{i=1}^n$ 得到所有系数 $\{a_i\}_{i=0}^{n-1}$。

程序 11.1 使用 numpy.polynomial 模块的 Polynomial 类的 fit 函数求解多项式函数的系数。图 11.1 的左上角子图绘制了对 6 个数据点进行插值的多项式函数的图像。当 n 较大时，插值得到的高次多项式函数在两个数据点之间可能出现大幅度波动现象。图 11.1 的左下角子图绘制了对 Runge 函数 $f(x) = \frac{1}{1+25x^2} \quad x \in [-1, 1]$ 上选取的若干数据点进行插值得到的 13 次和 14

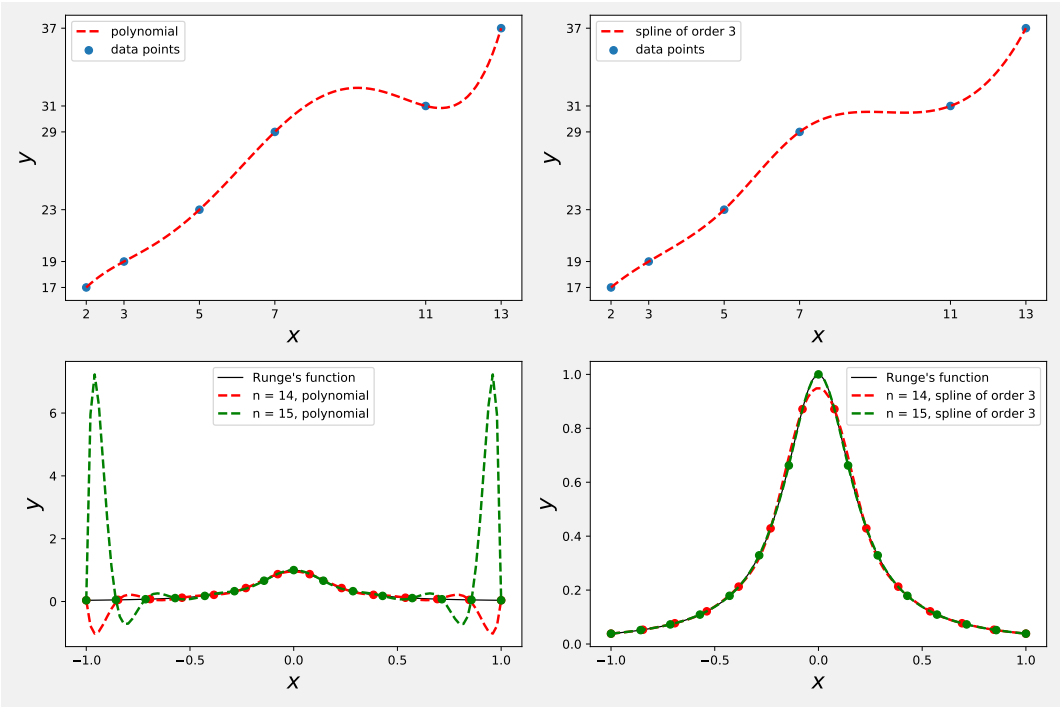


图 11.1: 一元插值

次多项式函数的图像，可见当 x 值接近 1 和-1 时两个函数都出现了大幅度波动。

为了避免高次多项式函数在插值时的病态性质，通常使用分段多项式函数作为插值函数，即把给定的数据点的 x 坐标值所在的区间划分为若干个子区间，在每个子区间上使用低次多项式函数作为插值函数。样条函数是一种特殊类型的分段多项式函数。一个 k 次分段多项式函数称为样条函数，如果它在所有相邻子区间的连接点 (称为节点) 处具有连续的 $k-1$ 阶导数。在实际应用中通常选择 $k=3$，三次样条函数在所有节点处具有连续的二阶导数。

scipy.interpolate 模块的 interp1d 类实现了一元样条函数插值。该类实现了___call___方法，可以作为函数调用。前两个参数分别是存储了数据点的 x 坐标值和 y 坐标值的数组，关键字实参 kind 可指定样条函数的次数。图 11.1 的右上子图和右下子图分别绘制了运行程序 11.1 进行样条插值得到的函数图像。可见三次样条函数的图像较为平滑，没有出现大幅度波动。

Listing 11.1: **一元插值：多项式函数和样条函数**

```
import numpy as np; from numpy import polynomial as pl
from scipy import interpolate; import matplotlib.pyplot as plt

fig, axes = plt.subplots(2, 2, figsize=(12, 8))

x = np.array([2,3,5,7,11,13])
```

```
y = np.array([17,19,23,29,31,37])
f1 = pl.Polynomial.fit(x, y, len(x) - 1)
f2 = interpolate.interp1d(x, y, kind='cubic')
colors = ('r', 'g'); labels = ('polynomial', 'spline of order 3')

xx = np.linspace(x.min(), x.max(), 100); f = (f1, f2)
for i in range(2):
    axes[0, i].set_xticks(x); axes[0, i].set_yticks(y)
    axes[0, i].scatter(x, y, label='data points')
    axes[0, i].plot(xx, f[i](xx), 'r--', lw=2, label=labels[i])

def runge(x): return 1/(1 + 25 * x**2)
def runge_interpolate(n):
    x = np.linspace(-1, 1, n + 1)
    f1 = pl.Polynomial.fit(x, runge(x), deg=n)
    f2 = interpolate.interp1d(x, runge(x), kind='cubic')
    return x, f1, f2

xx = np.linspace(-1, 1, 100)
for i in range(2):
    axes[1, i].set_xticks([-1, -0.5, 0, 0.5, 1])
    axes[1, i].plot(xx, runge(xx), 'k', lw=1,
                    label="Runge's function")

for n in (13, 14):
    x, f1, f2 = runge_interpolate(n); f = (f1, f2)
    for i in range(2):
        axes[1, i].plot(x, runge(x), colors[n-13]+'o');
        axes[1, i].plot(xx, f[i](xx), colors[n-13]+'--',
                        label='n = %d, ' % (n+1) + labels[i],
                        lw=2)

for i in range(2):
    for j in range(2):
        axes[i, j].set_xlabel(r"$x$", fontsize=18)
        axes[i, j].set_ylabel(r"$y$", fontsize=18)
        axes[i, j].legend();

fig.tight_layout(); plt.show()
```

11.1.2 二元插值

给定一些分布在三维空间的数据点，二元插值生成一个通过这些点的曲面。本节假设这些数据点在 x-y 平面上的投影分布在一个网格上。网格上 x 坐标值的集合为 $\{x_1, x_2, ..., x_m\}$，y 坐标值的集合为 $\{y_1, y_2, ..., y_n\}$，则数据点为
$\{(x_i, y_j, z_k)\}, \quad 1 \le i \le m, \quad 1 \le j \le n, \quad k = (i-1) \times n + j - 1$。

如果网格在 x 轴方向和 y 轴方向都是等间距的，可使用 RectBivariateSpline 函数进行二元样条函数插值。它的前两个参数是分别存储了网格上 x 坐标值和 y 坐标值的数组，第三个参数是网格上所有点的 z 坐标值，关键字实参 kx 和 ky 分别指定样条函数在 x 轴方向和 y 轴方向的次数。如果给定的数据点的网格不满足以上条件，可使用 interp2d 函数进行二元插值。它的前三个参数和 RectBivariateSpline 函数相同。指定关键字实参 kind 为'cubic' 表示使用三次样条函数。

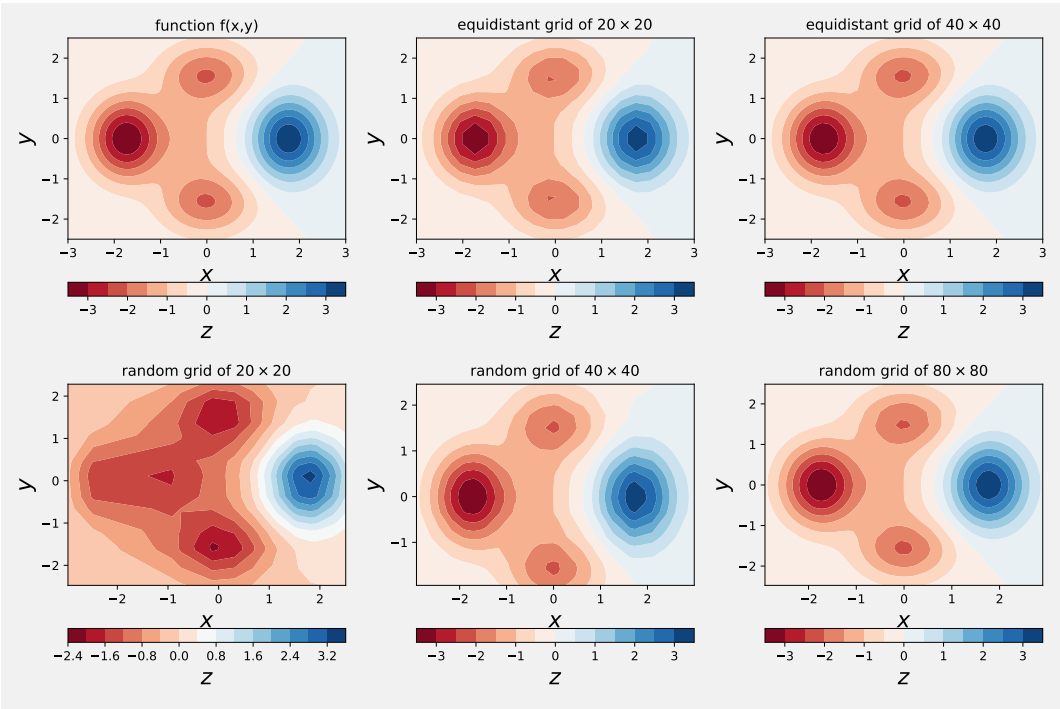


图 11.2: 等间距网格和随机网格上的二元插值

图 11.1 的左上子图绘制了函数 $z = (x^7 - y^6 + x^5 - y^4 + x^3 - y^2 + x - 1)e^{-x^2-y^2}$ 在区间 $[-3, 3] \times [-2.5, 2.5]$ 上的等高线图。中上子图和右上子图分别绘制了程序 11.2 生成的在 20×20 等间距网格和在 40×40 等间距网格上的二元样条插值函数的等高线图。左下子图、中下子图和右下子图分别绘制了程序 11.2 生成的在 20×20 随机网格、40×40 随机网格和

80×80 随机网格上的二元样条插值函数的等高线图。可见在网格的点数相同时，随机网格上的插值效果不如等间距网格。对于两种网格，增加网格的点数都可以改善插值的效果。

Listing 11.2: 等间距网格和随机网格上的二元插值

```
from numpy import polynomial as pl; import numpy as np
from scipy import interpolate; import matplotlib.pyplot as plt

def f(x,y):
    return (x**7 - y**6 + x**5 - y**4 + x**3 - y**2 + x - 1) *
            np.exp(-x**2 - y**2)

def create_colorbar(c,i,j):
    cb = fig.colorbar(c, ax=axes[i,j], orientation='horizontal')
    cb.set_label(r"$z$", fontsize=18);

fig, axes = plt.subplots(2, 3, figsize=(12, 8))

N = 20; xmin = -3; xmax = 3; ymin = -2.5; ymax = 2.5
xN = np.linspace(xmin, xmax, int(np.rint(N*(xmax-xmin))))
yN = np.linspace(ymin, ymax, int(np.rint(N*(ymax-ymin))))
XN, YN = np.meshgrid(xN, yN)
c = axes[0,0].contourf(XN, YN, f(XN, YN), 15, cmap=plt.cm.RdBu)
create_colorbar(c, 0, 0)
axes[0,0].set_title("function␣f(x,y)")

ns = (20, 40)
for i in range(2):
    n = ns[i]; title = "equidistant␣grid␣of␣"
    x = np.linspace(xmin, xmax, n)
    y = np.linspace(ymin, ymax, n)
    X, Y = np.meshgrid(x, y); Z = f(X, Y)
    f_i = interpolate.RectBivariateSpline(x, y, Z, kx=3, ky=3)
    c = axes[0, i+1].contourf(X, Y, f_i(x, y), 15,
                              cmap=plt.cm.RdBu)
    create_colorbar(c, 0, i+1)
    #axes[0, i+1].scatter(X, Y, marker='+', color="#f8f8f8")
    axes[0, i+1].set_title(title + (r"$%d␣\times␣%d$" % (n, n)))

ns = (20, 40, 80)
```

```
for i in range(3):
    n = ns[i]; title = "random grid of "
    x = np.sort(np.random.uniform(xmin, xmax, n))
    y = np.sort(np.random.uniform(ymin, ymax, n))
    X, Y = np.meshgrid(x, y); Z = f(X, Y)
    f_i = interpolate.interp2d(x, y, Z, kind='cubic')
    c = axes[1, i].contourf(X, Y, f_i(x, y), 15,
                            cmap=plt.cm.RdBu)
    create_colorbar(c, 1, i)
    #axes[1, i].scatter(X, Y, marker='+', color="#f8f8f8")
    axes[1, i].set_title(title + (r"$%d \times %d$" % (n, n)))

for i in range(2):
    for j in range(3):
        axes[i, j].set_xlabel(r"$x$", fontsize=18)
        axes[i, j].set_ylabel(r"$y$", fontsize=18)

fig.tight_layout(); plt.show()
```

11.1.3 数值积分

如果一个函数 $f(x)$ 不存在解析形式的积分原函数或函数的定义未知，只能通过数值方法求解其在区间 $[a,b]$ 上的定积分。即用区间内选取的 $n+1$ 个点 x_i $(i=0,1,...,n)$(称为积分节点)上的函数值的加权和近似计算：

$$\int_a^b f(x)\,dx \approx \sum_{i=0}^{n} w_i f(x_i) \tag{11.1}$$

其中 w_i 是函数值 $f(x_i)$ 的权重，称为积分系数。不同的数值计算公式的区别体现在积分节点和积分系数上。

插值求积公式 (interpolatory quadrature) 的原理是使用基于区间内一些节点进行插值得到的多项式函数作为原函数的近似。如果一个插值求积公式对于次数不超过 k 的多项式函数是精确的，但对次数为 $k+1$ 的多项式函数不精确，则称公式的精度 (degree of precision) 为 k。如果节点之间必须是等距的，得到的公式称为牛顿-柯特斯型求积公式 (Newton-Cotes quadrature)。如果节点选取为正交多项式 (例如勒让德 Legendre 多项式、切比雪夫 Chebyshev 多项式等) 的零点，得到的公式称为高斯型求积公式 (Gaussian quadrature)。当节点个数为 $n+1$ 时，一个插值求积公式的精度小于 $2n+2$，并且其精度等于 $2n+1$ 当且仅当它是高斯型求积公式 [WG2012]。为了提高插值求积公式的准确度，通常将积分区间划分为多个子区间，在每个子区间用多项式函数进行插值，由此得到的积分公式称为复合 (composite)

积分公式。表 5.1 中列出的复合梯形公式和复合辛普森公式属于牛顿-柯特斯型公式，复合高斯-勒让德公式属于高斯型公式。

梯形公式的节点有两个，精度为 1。辛普森法的节点有三个，精度为 3。两个节点的高斯-勒让德公式的精度为 3。程序 11.3 根据精度的定义，即

$$\int_a^b x^j\,dx = \sum_{i=0}^{n} w_i x_i^j \quad \text{for} \quad j = 0, ..., k \quad ,$$

利用 sympy 解方程组得到前两个公式的积分系数以及第三个公式的积分节点和积分系数**??**。

Listing 11.3: **数值积分公式**

```
import sympy as sym
a, b, x = sym.symbols("a, b, x"); f = sym.Function("f")

def Newton_Cotes(points):
    n = len(points) - 1;
    w = [sym.symbols("w_%d" % i) for i in range(n+1)]
    qu = sum([w[i] * f(points[i]) for i in range(n+1)])
    mo = [sym.Lambda(x, x**i) for i in range(n+1)]
    eqs = [qu.subs(f, mo[j]) - sym.integrate(mo[j](x), (x, a, b))
           for j in range(len(mo))]
    wv = sym.solve(eqs, w); print(wv)

def Gauss_Legendre():
    n = 1; n21 = n * 2 + 1
    w = [sym.symbols("w_%d" % i) for i in range(n+1)]
    p = [sym.symbols("p_%d" % i) for i in range(n+1)]
    qu = sum([w[i] * f(p[i]) for i in range(n+1)])
    mo = [sym.Lambda(x, x**i) for i in range(n21+1)]
    eqs = [qu.subs(f, mo[j]) - sym.integrate(mo[j](x), (x, a, b))
           for j in range(len(mo))]
    wp = sym.solve(eqs, (w+p))
    print(wp)

Newton_Cotes((a,b))                # Trapezoidal rule n=1
# {w_0: -a/2 + b/2, w_1: -a/2 + b/2}
Newton_Cotes((a, (a+b)/2, b))      # Simpson's rule n=2
# {w_0: -a/6 + b/6, w_1: -2*a/3 + 2*b/3, w_2: -a/6 + b/6}
Gauss_Legendre()                   # Gauss-Legendre n=1
```

```
'''
[(-(a - b)/2, -(a - b)/2,
 -sqrt(3)*a/6 + a/2 + sqrt(3)*b/6 + b/2,
 a/2 + b/2 + sqrt(3)*(a - b)/6),

 (-(a - b)/2, -(a - b)/2,
 sqrt(3)*a/6 + a/2 - sqrt(3)*b/6 + b/2,
 a/2 + b/2 - sqrt(3)*(a - b)/6)]
'''
```

scipy 库的 integrate 模块提供了多个数值积分函数。如果被积函数已给定，quad 函数使用高斯型公式计算其定积分，积分区间的上界和下界可以是正负无穷大。返回值为一个元组，由积分值和估计的绝对误差组成。如果被积函数包含多个变量，则 quad 只对第一个变量计算积分，可通过 args 关键字实参提供其他变量的值。如果要积分的变量不是函数的第一个变量 (例如 bessel 函数 jv(3, x))，可利用 lambda 函数将其交换到第一个变量。如果函数在求积公式的某个取样点处发散 (diverge)，可用 points 关键字实参指定需要规避的取样点。dblquad(f,a,b,c,d) 函数计算二重积分 $\int_a^b \int_{c(x)}^{d(x)} f(x,y)\,dx\,dy$，其中两个常数 a 和 b 是对 x 积分的区间的下界和上界，两个以 x 为自变量的 Python 函数 c 和 d 是对 y 积分的区间的的下界和上界。类似地，tplquad 函数计算三重积分 $\int_a^b \int_{c(x)}^{d(x)} \int_{g(x,y)}^{h(x,y)} f(x,y,z)\,dx\,dy\,dz$。nquad 函数计算 n 重积分，它的第一个参数是被积函数，第二个参数是一个列表，由对每个变量积分的区间的下界和上界组成的元组组成。

如果被积函数未知，只给定了函数的一些数据点 (例如通过实验和观测得到)，可使用牛顿-柯特斯型求积公式。simps 函数和 trapz 函数分别实现了辛普森公式和梯形公式。它们的参数为两个数组，分别存储了数据点的 y 坐标值和 x 坐标值。如果数据点的个数是 2^k+1(k 是一个正整数) 且数据点之间是等距的，可使用 romb 函数。romb 函数实现了 Romberg 公式，它的第一个参数是存储了数据点的 y 坐标值的数组，第二个参数是相邻数据点之间的间距。如果数据点之间不是等距的，也可以使用 simps 函数和 trapz 函数。

Listing 11.4: **quad 函数计算定积分**

```
In[1]: import numpy as np; from scipy import integrate as intg
In[2]: fx_x = lambda x: x**x; intg.quad(fx_x, 1.2, 3.4)
Out[2]: (30.599509184696394, 3.397227963907889e-13)
In[3]: f = lambda x: x**3 * np.exp(-x/2); intg.quad(f, 3, np.inf)
Out[3]: (89.69832437966879, 3.1502359529867805e-09)
In[4]: def f(x, m, s): return np.exp(-(np.log(x)-m)**2 / 2*s**2)
In[5]: intg.quad(f, 12, 345, args=(6, 7))
Out[5]: (15.74013093722831, 1.0590146746095474e-10)
In[6]: from scipy.special import jv; f = lambda x: jv(3, x)
In[7]: intg.quad(f, -2, 4)
```

```
Out[7]: (0.598452354532213, 8.20824966029651e-15)
In[8]: intg.quad(lambda x: 1/(x-2)**2, 1, 3)
Out[8]: ZeroDivisionError: float division by zero
In[9]: intg.quad(lambda x: 1/(x-2)**2, 1, 3, points=[2])
Out[9]: (-1.9999999878646795, 2.5911172985004782e-08)
In[10]: def f(x, y): return (x**2 + 3*y**2) * np.exp(-x**2 - y**2)
In[11]: intg.dblquad(f, 0.1, 2.3, lambda x:0.01,
                     lambda x:np.exp(x))
Out[11]: (1.4327976138140261, 1.339276139066072e-08)
In[12]: def f(x, y, z): return np.exp(-x-y**2-z**3)
In[13]: c = lambda x: x**2 + 1; d = lambda x: x**2 + 3*x + 2
In[14]: g = lambda x, y: 2; h = lambda x, y: x + y**2 + 3
In[15]: intg.tplquad(f, 0.1, 2.3, c, d, g, h)
Out[15]: (0.00673873020840579, 3.187940716866225e-11)
In[16]: intg.nquad(f, [(0, 1), (0, 1), (0, 1)])
Out[16]: (0.38121221110217796, 7.017914825696364e-15)
In[17]: x = np.linspace(1.2, 3.4, 129); y = fx_x(x)
In[18]: intg.simps(y, x), intg.trapz(y, x),
        intg.romb(y, x[1] - x[0])
Out[18]: 30.59950958243804 30.602983316264865 30.59950918469625
In[19]: x = np.sort(np.random.uniform(1.2, 3.4, 129))
In[20]: y = fx_x(x)
In[21]: intg.simps(y, x), intg.trapz(y, x)
Out[21]: 27.982982418725783 28.00358352039727
```

11.2 代数方程求解

11.2.1 线性方程组求解

线性方程组的一般形式是 $\mathbf{Ax} = \mathbf{b}$，其中 $\mathbf{A}$ 是一个 m 行 n 列的矩阵。

当 $m = n$ 并且 $\mathbf{A}$ 非奇异时：若 $\mathbf{A}$ 是稠密矩阵，可使用 6.5 节介绍的 scipy.linalg 模块中定义的 solve 函数求解；若 $\mathbf{A}$ 是稀疏矩阵，可使用 6.6 节介绍的 scipy.sparse.linalg 模块中定义的 spsolve、cg 和 bicg 等函数求解。

numpy.linalg 模块的 lstsq 函数通过最小化欧式范数 $\|\mathbf{Ax} - \mathbf{b}\|_2^2$ 求最小二乘解。$\mathbf{A}$ 的线性无关行的个数可以大于或小于或等于它的线性无关列的个数。

Listing 11.5: **使用 numpy.linalg.lstsq 函数求最小二乘解**

```
import numpy as np
X = np.array([[1, -5, 2, 7], [3, 4, -2, 8], [5, 1, -3, 9],
              [2, -4, 1, -6], [6, 2, 3, 7]])
y = np.array([11, 13, 17, 19, 23])
print(np.linalg.lstsq(X, y, rcond=None)[0])
# [ 4.76101434 -1.32173795 -0.00613474 -0.23511447]
```

10.5.1 节介绍的 SymPy 库的 linsolve 函数可以求解规模较小的线性方程组，并得到精确解。**A** 的元素可以使用符号变量。

11.2.2 非线性方程求解

10.5.1 节已介绍了使用 SymPy 库的 solveset 函数求解非线性方程。大多数非线性方程无法获得解析解，只能使用数值方法求近似解。数值方法求解非线性方程的基本方法有三种：二分法、割线法和牛顿法。这三种方法的共同点是通过多次迭代求解，区别在于收敛的速度和可靠性。后两种方法的基本思想都是使用线性函数作为非线性函数的近似。以下用 $f(x)=0$ 表示要求解的方程，α 表示未知解。定义第 k 次迭代时的解为 x_k，它和未知解的绝对误差为 $\delta_k=|x_k-\alpha|$。如果对于一个迭代算法可以证明 $\lim_{k\to+\infty}\frac{\delta_{k+1}}{{\delta_k}^q}$ 是一个常数，则它的收敛阶 (order of convergence) 为 q。

二分法依据的定理是：若函数 $f(x)$ 在 $[a,b]$ 上连续，且 $f(a)f(b)<0$，则在 (a,b) 上至少存在一个解。二分法的原理是通过每次迭代把包含解的区间的宽度缩小为原来的一半。令 $a_1=a$ 和 $b_1=b$。设第 k 次迭代时的区间是 $[a_k,b_k]$，定义 x_k 为区间的中点：$x_k=(a_k+b_k)/2$。若 $f(x_k)=0$，则已找到解。否则，两个不等式 $f(a_k)f(x_k)<0$ 和 $f(x_k)f(b_k)<0$ 中必有一个成立。若前者成立则设定 $a_{k+1}=a_k$ 和 $b_{k+1}=x_k$，否则设定 $a_{n+1}=x_k$ 和 $b_{k+1}=b_k$，然后进行下一次迭代。x_k 和未知解的绝对误差 $\delta_k=|\alpha-x_k|\le(b_k-a_k)/2=(b-a)/2^k$。为了使得 δ_k 不超过一个预先设定的值 ϵ，即 $(b-a)/2^n\le\epsilon$，需要迭代的次数为 $\lceil\log\frac{b-a}{\epsilon}/\log 2\rceil$。二分法的优点是可靠性：从满足条件的区间开始迭代可以保证获得解。二分法的缺点是收敛阶为 1，收敛速度比另外两种方法慢。

割线法从两个给定的起始点 x_0 和 x_1 开始迭代。其原理是在第 k 次迭代时，用通过当前解 x_k 和第 $k-1$ 次的解 x_{k-1} 的数据点 $(x_k,f(x_k))$ 和 $(x_{k-1},f(x_{k-1}))$ 的割线 $y=f(x_k)+\frac{f(x_k)-f(x_{k-1})}{x_k-x_{k-1}}(x-x_k)$ 作为函数 $f(x)$ 的近似，把割线和 x 轴的交点的横坐标作为 x_{k+1} 进行下一次迭代。求解线性方程 $0=f(x_k)+\frac{f(x_k)-f(x_{k-1})}{x_k-x_{k-1}}(x-x_k)$ 可得 $x_{k+1}=x_k-\frac{x_k-x_{k-1}}{f(x_k)-f(x_{k-1})}f(x_k)$。当两个起始点与未知解足够接近时割线法可以保证收敛 (称为局部收敛)，收敛阶 (至少) 是 1.61803[WG2012]。

牛顿法从一个给定的起始点 x_0 开始迭代。其原理是在第 k 次迭代时，用通过当前解 x_k 的数据点 $(x_k,f(x_k))$ 的切线 $y=f(x_k)+f'(x_k)(x-x_k)$ 作为函数 $f(x)$ 的近似，把切线和 x 轴的交

点的横坐标作为 x_{k+1} 进行下一次迭代。求解线性方程 $0 = f(x_k) + f'(x_k)(x_{k+1} - x_k)$ 可得 $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$。牛顿法也是局部收敛的。当 $f''(\alpha) \neq 0$ 时，牛顿法的收敛阶是 2。如果起始点和未知解相差较大，可能导致收敛速度很慢或不收敛，例如 $f(x) = x^{20} - 1$ 和 $x_0 = 1/2$[WG2012]。

Brent 方法是一种综合了二分法、割线法以及逆二次插值的复杂方法。它具有二分法的可靠性和接近割线法的收敛速度。如果函数的导数未知，则 Brent 方法是首选方法。

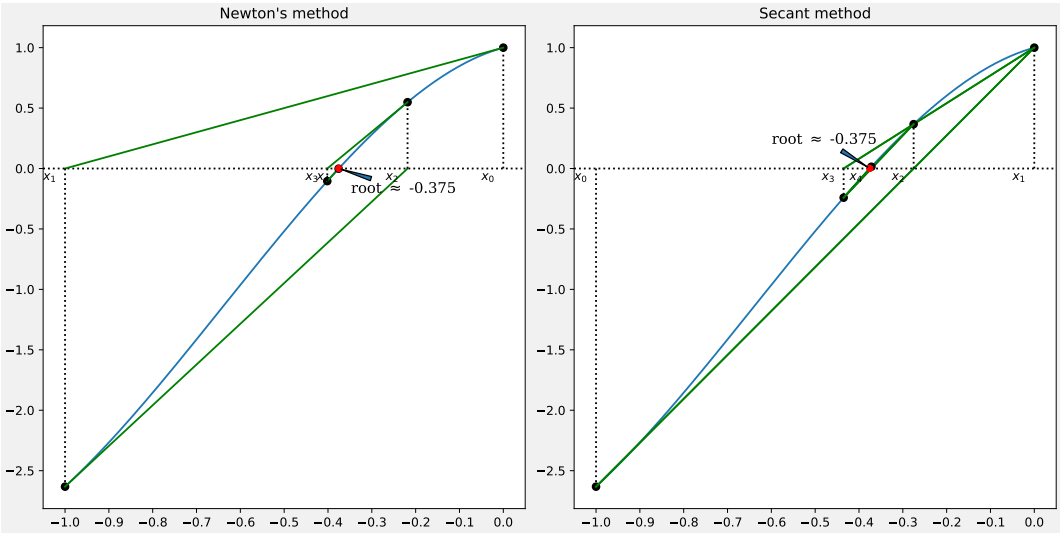


图 11.3: 牛顿法和割线法

scipy 库的 optimize 模块定义了多个求解非线性方程的函数，它们的第一个参数都是 $f(x)$。bisect 函数实现了二分法，它的第二个和第三个参数分别是区间的下界和上界。newton 函数实现了牛顿法和割线法，它的第二个参数是起始点。如果提供了 $f(x)$ 的导数作为 fprime 关键字实参，则使用牛顿法，否则使用割线法。brentq 函数实现了 Brent 方法，它的第二个和第三个参数分别是区间的下界和上界。程序 11.6 演示了使用这些函数求解方程 $e^x - 3x^3 - 6x^2 = 0$。程序 11.7 绘制的图 11.3 显示了牛顿法和割线法的前几次迭代过程。

Listing 11.6: **非线性方程求解**

```
In[1]: from scipy import optimize as optm
In[2]: def f(x): return np.exp(x) - 3*x**3 - 6*x**2
In[3]: optm.bisect(f, 0, 1), optm.bisect(f, -1, 0)
Out[3]: (0.4638232777815574, -0.37544418741708796)
In[4]: fp = lambda x: np.exp(x) - 9*x**2 - 12*x
In[5]: optm.newton(f, 1, fprime = fp),
       optm.newton(f, 0, fprime = fp)
```

```
Out[5]: (0.4638232777821511, -0.3754441874171241)
In[6]: optm.newton(f, 1), optm.newton(f, 0)
Out[6]: (0.4638232777821511, -0.3754441874171324)
In[7]: optm.brentq(f, 0, 1), optm.brentq(f, -1, 0)
Out[7]: (0.4638232777821511, -0.3754441874168658)
In[8]: optm.newton(lambda x: x**20-1, 0.5,
                   fprime = lambda x: 20*x**19)
Out[8]: RuntimeError: Failed to converge after 50 iterations, ...
```

Listing 11.7: 牛顿法和割线法

```
import numpy as np; import matplotlib.pyplot as plt
import sympy as sym

tol = 0.01; s_x = sym.symbols("x");
s_f = sym.exp(s_x) - 3*s_x**3 - 6*s_x**2
f = lambda x: sym.lambdify(s_x, s_f, 'numpy')(x)
fp = lambda x: sym.lambdify(s_x, sym.diff(s_f, s_x), 'numpy')(x)
lb = -1; ub = 0;  xk = ub; x = np.linspace(lb, ub, 1000);

fig, axes = plt.subplots(1, 2, figsize=(12, 6), dpi = 300,
                         constrained_layout=True)

def plot_x(i, x, n):
    axes[i].plot([x, x], [0, f(x)], color='k', ls=':')
    axes[i].plot(x, f(x), 'ko')
    axes[i].text(x-0.05, -.1, r'$x_%d$' % n, fontsize=10)

axes[0].plot(x, f(x)); axes[0].axhline(0, ls=':', color='k')
for n in range(5):
    xk_new = xk - f(xk) / fp(xk)
    plot_x(0, xk, n)
    axes[0].plot([xk, xk_new], [f(xk), 0], 'g-');
    xk = xk_new; n += 1
    print(n, xk_new)

axes[0].plot(xk, f(xk), 'ro', markersize=5)
axes[0].annotate(r"root␣$\approx$␣%.3f" % xk, fontsize=12,
                 family="serif", xy=(xk, f(xk)),
                 xytext=(10, -20), textcoords='offset␣points',
```

```
                    arrowprops=dict(arrowstyle='wedge'))
axes[0].set_title("Newton's method")
axes[0].set_xticks(np.linspace(lb, ub, 11))

axes[1].plot(x, f(x)); axes[1].axhline(0, ls=':', color='k')

xj = -1; xk = 0
plot_x(1, xj, 0)
for n in range(4):
    xk_new = xk - f(xk) * (xk - xj) / (f(xk) - f(xj))
    axes[1].plot([xk_new, xk], [0, f(xk)], 'g-');
    axes[1].plot([xk_new, xj], [0, f(xj)], 'g-');
    axes[1].plot([xk, xj], [f(xk), f(xj)], 'g-');
    n += 1; plot_x(1, xk, n); xj = xk; xk = xk_new;
    print(n, xk_new)

axes[1].plot(xk, f(xk), 'ro', markersize=5)
axes[1].annotate(r"root $\approx$ %.3f" % xk, fontsize=12,
                    family="serif", xy=(xk, f(xk)),
                    xytext=(-80, 20), textcoords='offset points',
                    arrowprops=dict(arrowstyle='wedge'))
axes[1].set_title("Secant method")
axes[1].set_xticks(np.linspace(lb, ub, 11))
```

11.2.3 非线性方程组求解

10.5.1 节已介绍了使用 SymPy 库的 nonlinsolve 函数求解非线性方程组。大多数非线性方程组无法获得精确解，只能使用数值方法求近似解。含有 n 个变量的非线性方程组的一般形式为

$$\begin{aligned} f_1(x_1, x_2, ..., x_n) &= 0 \\ f_2(x_1, x_2, ..., x_n) &= 0 \\ ... &= 0 \\ f_n(x_1, x_2, ..., x_n) &= 0 \end{aligned}$$

定义 $\mathbf{f} = (f_1, f_2, ..., f_n)$ 为一个以 n 维向量 $\mathbf{x} = (x_1, x_2, ..., x_n)$ 为自变量的函数，它的函数值也是 n 维向量。以上方程组表示为 $\mathbf{f}(\mathbf{x}) = \mathbf{0}$。对于一元函数使用的牛顿法可推广到 m 维向量函数。在第 k 次迭代时，使用线性函数 $\mathbf{f}(\mathbf{x_k}) + \mathbf{J_f}(\mathbf{x_k})(\mathbf{x} - \mathbf{x_k})$ 作为函数 $\mathbf{f}(\mathbf{x})$ 的近似。其中 $\mathbf{J_f}(\mathbf{x_k})$ 是函数 $\mathbf{f}(\mathbf{x})$ 在 $\mathbf{x_k}$ 处的 Jacobian 矩阵，$[\mathbf{J_f}(\mathbf{x})]_{ij} = \frac{\partial f_i(\mathbf{x})}{\partial x_j}$。求解线性方程

$\mathbf{0} = \mathbf{f}(\mathbf{x_k}) + \mathbf{J_f}(\mathbf{x_k})(\mathbf{x_{k+1}} - \mathbf{x_k})$ 可得 $\mathbf{x_{k+1}} = \mathbf{x_k} - \mathbf{J_f}(\mathbf{x_k})^{-1}\mathbf{f}(\mathbf{x_k})$。为了避免求矩阵的逆，可以定义 $_\mathbf{k} = \mathbf{x_{k+1}} - \mathbf{x_k}$，则每次迭代只需求解线性方程组 $\mathbf{J_f}(\mathbf{x_k})_\mathbf{k} = -\mathbf{f}(\mathbf{x_k})$，然后设定 $\mathbf{x_{k+1}} = \mathbf{x_k} + _\mathbf{k}$。

scipy 库的 optimize 模块的 root 函数可求解非线性方程组，它的第一个参数是要求解的方程组。关键字实参 method 可指定多种求解方法，例如‘hybr’(MINPACK 实现的 Powell 方法)、‘lm’(Levenberg-Marquardt 算法) 和‘broyden1’(近似计算 Jacobian 矩阵的 Broyden 方法) 等。对于前两种方法，可以通过关键字实参 jac 说明是否提供 Jacobian 矩阵。第三种方法无需提供 Jacobian 矩阵。
程序 11.8 演示了用以上方法求解非线性方程组：

$$\begin{aligned} 3x - \cos(yz) - 0.5 &= 0 \\ 4x^2 - 625y^2 + 2y - 1 &= 0 \\ e^{-xy} + 20z - 9\pi &= 0 \end{aligned}$$

第 5 行至第 7 行定义了一个列向量 f_mat，它的每个分量是组成方程组的一个方程，方程中的数学函数 (例如 cos) 由 SymPy 库定义。第 8 行使用 SymPy 计算了 Jacobian 矩阵 $\begin{bmatrix} 3 & z\sin(yz) & y\sin(yz) \\ 8x & 2-1250y & 0 \\ -ye^{-xy} & -xe^{-xy} & 20 \end{bmatrix}$。

第 10 行至第 13 行定义了一个函数 f，它返回一个列表，其中的每个元素是组成方程组的一个方程。方程中需要求解的各变量 (x,y,z) 用一个数组 x 的各元素 (x[0],x[1],x[2]) 分别表示，方程中的数学函数 (例如 cos) 由 NumPy 库定义。第 14 行至第 20 行定义了一个函数 f2，它返回一个元组。其中的第一个元素是 f(x)，第二个元素是存储了 Jacobian 矩阵的二维数组。关键字实参 jac 设定为 True 时 root 函数的第一个实参是 f2，否则是 f。

Listing 11.8: 非线性方程组求解

```
1  import numpy as np; import sympy as sym
2  from scipy import optimize as optm
3
4  x, y, z = sym.symbols("x, y, z")
5  f_mat = sym.Matrix([3*x - sym.cos(y*z) - 0.5,
6                      4*x**2 - 625*y**2 + 2*y - 1,
7                      sym.exp(-x*y) + 20*z - 9*sym.pi])
8  j = f_mat.jacobian(sym.Matrix([x, y, z]))
9
10 def f(x):
11     return [3*x[0] - np.cos(x[1]*x[2]) - 0.5,
12             4*x[0]**2 - 625*x[1]**2 + 2*x[1] - 1,
```

```
            np.exp(-x[0]*x[1]) + 20*x[2] - 9*np.pi]
def f2(x):
    f_j =  np.array([
            [3, x[2]*np.sin(x[1]*x[2]), x[1]*np.sin(x[1]*x[2])],
            [8*x[0], 2-1250*x[1], 0],
            [-x[1]*np.exp(-x[0]*x[1]),
             -x[0]*np.exp(-x[0]*x[1]), 20] ])
    return f(x), f_j

print(optm.root(f2, [1, 1, 1], jac=True, method='hybr'))
# x: array([0.49999684, 0.00319366, 1.36379647])
print(optm.root(f, [1, 1, 1], jac=False, method='hybr'))
# x: array([0.49999684, 0.00319366, 1.36379647])
print(optm.root(f2, [1, 1, 1], jac=True, method='lm'))
#  x: array([0.49999684, 0.00319366, 1.36379647])
print(optm.root(f, [1, 1, 1], jac=False, method='lm'))
# x: array([0.49999684, 0.00319366, 1.36379647])
print(optm.root(f, [1, 1, 1], method='broyden1'))
# x: array([0.49999738, 0.00319717, 1.36379671])
```

11.3 常微分方程求解

常微分方程指包含未知函数和其对于单个自变量的一阶或高阶导数的方程。常微分方程的阶数由方程中出现的最高阶导数确定。求解常微分方程的方法是对其进行解析积分或数值积分，一个 n 阶常微分方程在积分时会产生 n 个积分常数，需要指定 n 个边界条件才能确定所有积分常数。如果这些边界条件都位于自变量的同一个点上，称为初值问题。如果这些边界条件位于自变量的多个点上，称为边值问题。10.5.2 节已介绍了使用 SymPy 库的 dsolve 函数求解一些常微分方程。很多常微分方程无法获得解析解，只能使用数值方法求解。

11.3.1 常微分方程的初值问题

常微分方程的初值问题的标准形式 [WG2012] 是求解一个满足以下一阶常微分方程和初值条件的向量值函数 $\mathbf{y}(x) \in C^1[a,b]$：

$$\frac{d\mathbf{y}}{dx} = \mathbf{f}(x,\mathbf{y}), \quad a \le x \le b; \quad \mathbf{y}(a) = \mathbf{y}_0.$$

出现了高阶导数的常微分方程可以通过引入新的变量转换为标准形式。例如引入变量 $v = \frac{ds}{dt}$ 并定义 $\mathbf{y} = [s,v]^T$ 和 $f(t,\mathbf{y}) = [v, \frac{F}{m}]^T$，则牛顿第二运动定律 $F(s) = m\frac{d^2s}{dt^2}$ 可以转换为标准形式：$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t,\mathbf{y})$。

关于初值问题的解的基本定理是：设 $\mathbf{f}(x,\mathbf{y})$ 在 $[a,b]\times R^d$ 上连续，并且满足 Lipschitz 条件：

$$|\mathbf{f}(x,\mathbf{y})-\mathbf{f}(x,\mathbf{y}^*)|\le L|\mathbf{y}-\mathbf{y}^*|,\quad \forall x\in[a,b],\quad \forall\mathbf{y}\in R^d,\quad \forall\mathbf{y}^*\in R^d$$

则以上初值问题对于所有 $\mathbf{y}_0\in[a,b]$ 存在唯一解 $\mathbf{y}(x)$，并且 $\mathbf{y}(x)$ 在 $[a,b]$ 上连续可微。

初值问题的数值求解方法是在区间 $[a,b]$ 上的一些离散的点 x_n 上计算 $\mathbf{y}(x_n)$ 的近似值 $\mathbf{u}_n$。这些点 x_n 满足条件：$a=x_0<x_1<...x_N<x_{N+1}=b$，通常设定相邻点之间具有相同的间距 h，即 $x_n=x_0+nh$。求解方法可分为两类：单步 (one-step) 方法和多步 (multistep) 方法。

单步方法

单步方法根据 x_n、$\mathbf{u}_n$ 和步长 $h=x_{n+1}-x_n$ 计算 $\mathbf{u}_{n+1}$，一般形式为：

$$\mathbf{u}_{n+1}=\mathbf{u}_n+h(x_n,\mathbf{u}_n;h)$$

定义局部离散化误差 (local discretization error)$\tau(x,\mathbf{u};h)=(x,\mathbf{u};h)-\frac{\mathbf{y}(x+h)-\mathbf{y}(x)}{h}$。若对于一种单步方法，$\lim_{h\to0}\tau(x,\mathbf{u};h)=\mathbf{0}$，则称其为一致的 (consistent)。若 $\tau(x,\mathbf{u};h)=O(h^p)$ 当 $h\to0$，则称这种方法的阶 (order) 为 p[SB2002]。定义全局离散化误差 (global discretization error)$\mathbf{e}(x,h)=\mathbf{u}(x,h)-\mathbf{y}(x)$。若对于一种单步方法，$\lim_{h\to0}\mathbf{e}(x,\mathbf{u};h)=\mathbf{0}$，则称其为收敛的 (convergent)[SB2002]。
显式欧拉方法的公式是：$\mathbf{u}_{n+1}=\mathbf{u}_n+h\mathbf{f}(x_n,\mathbf{u}_n)$。隐式欧拉方法的公式是：
$\mathbf{u}_{n+1}=\mathbf{u}_n+h\mathbf{f}(x_n,\mathbf{u}_{n+1})$。隐式方法需要通过迭代求解非线性方程，计算量大于显式方法，但具有更好的稳定性。欧拉方法的阶是 1。欧拉法仅使用了点 $(x_n,\mathbf{u}_n)$，Runge-Kutta 方法使用多个点以提高计算结果的精确性。显式 Runge-Kutta 方法的定义是：

$$\phi(x_n,\mathbf{u}_n,h)=\sum_{s=1}^{r}\alpha_s\mathbf{k}_s,$$

$$\mathbf{k}_1(x_n,\mathbf{u}_n)=\mathbf{f}(x_n,\mathbf{u}_n),\quad \mathbf{k}_s(x_n,\mathbf{u}_n,h)=\mathbf{f}(x_n+\mu_sh,\mathbf{u}_n+h\sum_{j=1}^{r}\lambda_{sj}\mathbf{k}_j),\quad s=2,3,...,r.$$

公式中的系数的确定方式是：将局部误差的定义中的两项分别进行泰勒展开再合并 h 的各次幂的同类项，然后使得尽可能多的 h 的低次幂项的系数为 0，以最大化阶。
例如显式 4 阶 Runge-Kutta 方法的公式是

$$\phi(x_n,\mathbf{u}_n,h)=\frac{1}{6}(\mathbf{k}_1+2\mathbf{k}_2+2\mathbf{k}_3+\mathbf{k}_4),$$

$$\mathbf{k}_1(x_n,\mathbf{u}_n)=\mathbf{f}(x_n,\mathbf{u}_n),\ \mathbf{k}_2(x_n,\mathbf{u}_n,h)=\mathbf{f}(x_n+\frac{1}{2}h,\mathbf{u}_n+\frac{1}{2}h\mathbf{k}_1),$$

$$\mathbf{k}_3(x_n,\mathbf{u}_n,h)=\mathbf{f}(x_n+\frac{1}{2}h,\mathbf{u}_n+\frac{1}{2}h\mathbf{k}_2),\ \mathbf{k}_4(x_n,\mathbf{u}_n,h)=\mathbf{f}(x_n+h,\mathbf{u}_n+h\mathbf{k}_3)$$

隐式 Runge-Kutta 方法的定义是：

$$\phi(x_n,\mathbf{u}_n,h)=\sum_{s=1}^{r}\alpha_s\mathbf{k}_s(x_n,\mathbf{u}_n,h),$$

$$\mathbf{k}_s(x_n, \mathbf{u}_n, h) = \mathbf{f}(x_n + \mu_s h, \mathbf{u}_n + h\sum_{j=1}^{r}\lambda_{sj}\mathbf{k}_j), \quad s = 1, 3, ..., r.$$

半隐式 Runge-Kutta 方法的定义和隐式方法基本相同，唯一的区别是第二个公式中的求和项的下标范围改为从 $j = 1$ 到 $j = s$。隐式方法和半隐式方法都需要求解非线性方程组以确定 $\mathbf{k}_s$，计算量大于显式方法，但具有更好的稳定性。

单步方法的步长 h 如果太大，则局部离散化误差过大，导致计算结果不准确。步长 h 如果太小，则需要计算的步数太多，导致计算代价太大，并且舍入误差也会随着步数的增加不断积累。为了最优化步长，Runge-Kutta-Fehlberg 方法在每一步使用两种阶数 (例如 4 阶和 5 阶) 的 Runge-Kutta 方法，根据当前的计算结果动态调整下一步的步长 [SB2002]。

多步方法

线性多步方法的一般形式是：

$$\mathbf{u}_{n+k} + \alpha_{k-1}\mathbf{u}_{n+k-1} + ... + \alpha_0\mathbf{u}_n = h[\beta_k\mathbf{f}_{n+k} + \beta_{k-1}\mathbf{f}_{n+k-1} + ... + \beta_0\mathbf{f}_n], \quad n = 0, 1, 2, ..., N-k,$$

$$\mathbf{f}_r = \mathbf{f}(x_r, \mathbf{u}_r), \quad x_r = a + rh, \quad r = 0, 1, 2, ..., N$$

其中 $\alpha_i(0 \le i \le k-1)$ 和 $\beta_i(0 \le i \le k)$ 是给定的系数。若 α_0 和 β_0 不同时为 0，则以上公式称为 k 步方法。若 β_k 为 0，即需要计算的 $\mathbf{u}_{n+k}$ 只出现在公式的左边，则以上公式称为显式的 (explicit)，否则称为隐式的 (implicit)。在获得相同的计算精确度的前提下，隐式方法可以比显式方法使用更大的步长 h，但需要通过迭代求解非线性方程。使用线性多步方法时需要借助其他方法 (如单步法) 初始化 $\mathbf{u}_i(0 \le i \le k-1)$。

初值问题的积分形式是

$$\mathbf{y} = \mathbf{y_0} + \int_a^b \mathbf{f}(x, \mathbf{y})dx$$

将公式中的积分项用通过 k 个点 $x_{n+i}(0 \le i \le k-1)$ 的拉格朗日插值多项式的积分代替即可得到 Adams – Bashforth 方法。

$$\mathbf{u}_{n+k} = \mathbf{u}_{n+k-1} + \sum_{i=0}^{k-1}[\int_{x_{n+k-1}}^{x_{n+k}} L_i(x)dx]\mathbf{f}(x_{n+i}, \mathbf{u}_{n+i}) = h\sum_{i=0}^{k-1}\beta_{k,i}\mathbf{f}(x_{n+i}, \mathbf{u}_{n+i}),$$

$$\beta_{k,i} = \int_0^1 \prod_{j=0, j\ne i}^{k-1}(\frac{t+k-1-j}{i-j})dt, \quad i = 0, 1, ..., k-1$$

Adams – Bashforth 方法的隐式形式，即选取的 k 个点 $x_{n+i}(1 \le i \le k)$ 包含了 x_{n+k}，称为 Adams – Moulton 方法

$$\mathbf{u}_{n+k} = \mathbf{u}_{n+k-1} + h\sum_{i=1}^{k}\beta_{k,i}\mathbf{f}(x_{n+i}, \mathbf{u}_{n+i}),$$

$$\beta_{k,i} = \int_0^1 \prod_{j=0,j\neq i}^{k-1} (\frac{t+k-1-j}{i-j})dt, \quad i = 1, 2, ..., k$$

预测器修正器 (Predictor – Corrector) 方法是显式方法和隐式方法的组合：先使用显式方法得到 $\mathbf{u}_{n+k}$ 的近似，再以其作为隐式方法的初始值，隐式方法通过多次迭代对其进行修正，获得较为精确的计算结果。

刚性微分方程

以下右边给出了左边常微分方程的通解 ($x \geq 0$)，其中 C_1 和 C_2 是积分常数，参数 λ_1 和 λ_2 都是负数。

$$\begin{pmatrix} y_1'(x) \\ y_2'(x) \end{pmatrix} = \begin{pmatrix} \frac{\lambda_1+\lambda_2}{2} & \frac{\lambda_1-\lambda_2}{2} \\ \frac{\lambda_1-\lambda_2}{2} & \frac{\lambda_1+\lambda_2}{2} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \qquad \begin{cases} y_1(x) = C_1 e^{\lambda_1 x} + C_2 e^{\lambda_2 x} \\ y_2(x) = C_1 e^{\lambda_1 x} - C_2 e^{\lambda_2 x} \end{cases}$$

显式欧拉方法在第 n 步得到的结果是：
$\mathbf{u}_n = [C_1(1+h\lambda_1)^n + C_2(1+h\lambda_2)^n, C_1(1+h\lambda_1)^n - C_2(1+h\lambda_2)^n]^T$。当 $n \to +\infty$ 时，$(1+h\lambda_1)^n$ 和 $(1+h\lambda_2)^n$ 都收敛到 0 的条件是 $|1+h\lambda_1| < 1$ 和 $|1+h\lambda_2| < 1$。若 λ_1 和 λ_2 存在较大数量级差异，例如设定 $\lambda_1 = -1, \lambda_2 = -10000$，则在通解中含 $e^{\lambda_2 x}$ 的项和含 $e^{\lambda_1 x}$ 的项相比几乎可以忽略，但是数值求解的步长却必须限定为 $h < 2/|\lambda_2| = 0.0002$。这类常微分方程称为刚性 (stiff)，特点是 $d \times d$ 的矩阵 $\mathbf{f_y}(x, \mathbf{y})$ 存在多个实部为负数且存在较大数量级差异的特征值。

在初值问题 $\mathbf{y}' = \mathbf{A}\mathbf{y}, \quad \mathbf{y}(0) = \mathbf{y}_0$ 中，$\mathbf{A}$ 是一个常数矩阵。该问题的数值解满足条件 $\mathbf{u}_{n+1} = \mathbf{g}(h\mathbf{A})\mathbf{u}_n$，其中函数 $\mathbf{g}(\mathbf{z})$ 仅依赖于使用的方法。若假定 $\mathbf{A}$ 的所有特征值的实部为负数，则该问题的解析解在 $x \to +\infty$ 时收敛到 $\mathbf{0}$，而数值解在 $n \to +\infty$ 时收敛到 $\mathbf{0}$ 的条件是 $\mathbf{A}$ 的每个特征值 λ_j 都满足 $|\mathbf{g}(h\lambda_j)| < 1$。对于一种数值求解方法，定义其绝对稳定区域 (region of absolute stability) 为 $RAS = \{z \in C| \quad |\mathbf{g}(z)| < 1\}$，其中 C 表示复平面。RAS 和复平面的左半平面 $C_- = \{z \in C| \quad Re(z) < 0\}$ 的交集范围越大，则该方法越适用于刚性常微分方程。若一种方法的 RAS 包含 C_-，则称其为绝对稳定的 (absolutely stable)。对于显式欧拉方法，$\mathbf{g}(\mathbf{z}) = \mathbf{1} + \mathbf{z}$，因此 $RAS = \{z \in C| \quad |1+z| < 1\}$。对于隐式欧拉方法，$\mathbf{g}(\mathbf{z}) = \mathbf{1}/(\mathbf{1} - \mathbf{z})$，因此 $RAS = \{z \in C| \quad |1-z| > 1\} \supset C_-$。隐式欧拉方法是绝对稳定的，显式欧拉方法则不是。多步方法中不存在超过 2 阶的绝对稳定方法。BDF(backward differentiation formula) 通过求解方程 $P'(x_{n+k}) = f(x_{n+k}, P(x_{n+k}))$ 来确定 y_{n+k}，其中 P 是通过 $k+1$ 个点 $x_{n+i}(0 \leq i \leq k)$ 的插值多项式函数。不超过 6 阶的 BDF 方法具有较好的稳定性 [JC2021]。

11.3.2 初值问题的数值求解方法

scipy 库的 integrate 模块定义的 solve_ivp 函数使用数值方法求解初值问题，

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y}), \quad t_0 \leq x \leq t_f; \quad \mathbf{y}(t_0) = \mathbf{y}_0.$$

它的第一个参数 fun 是方程的右边表达式，第二个参数 t_span 是一个包含 t_0 和 t_f 的元组，第三个参数 y0 是 $\mathbf{y}_0$，第四个参数 method 指定求解方法 (11.1)，第五个参数 t_eval 指定一个取值范围在 t_span 内的数组表示返回的求解结果的自变量值。

方法名称	方法含义
'RK45'	显式 Runge-Kutta 方法，误差控制基于 4 阶方法，步长基于 5 阶方法。
'RK23'	显式 Runge-Kutta 方法，误差控制基于 2 阶方法，步长基于 3 阶方法。
'DOP853'	显式 8 阶 Runge-Kutta 方法，精确度高于前两种方法。
'Radau'	隐式 Radau IIA 5 阶 Runge-Kutta 方法。
'BDF'	隐式多步可变阶方法。

表 11.1: method 参数可指定的求解方法：前三种仅适用于非刚性微分方程，后两种适用于刚性微分方程。

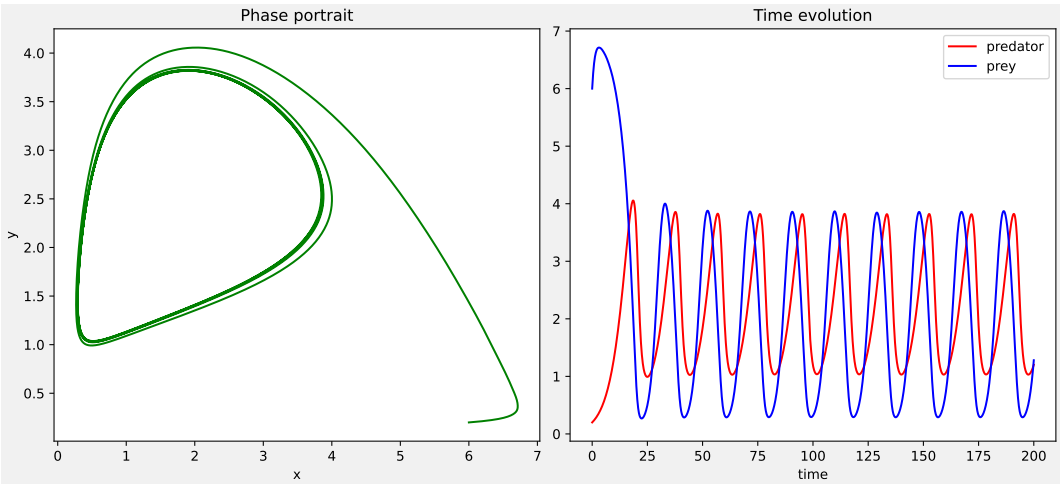


图 11.4: Holling-Tanner 模型

Holling-Tanner 模型 [SL2018] 是一种描述捕食者 (predator) 和食饵 (prey) 的数量变化情况的模型：

$$\begin{aligned}\frac{dx}{dt} &= x(1-\frac{x}{7}) - \frac{6xy}{7+7x} \\ \frac{dy}{dt} &= 0.2y(1-\frac{Ny}{x})\end{aligned}$$

方程中 $x(t) \neq 0$ 和 $y(t)$ 分别表示捕食者和食饵在时刻 t 的数量。项 $x(1-\frac{x}{7})$ 表示在无捕食者时食饵数量的增长速率，项 $\frac{6xy}{7+7x}$ 表示捕食者对食饵的捕食效果。项 $0.2y(1-\frac{Ny}{x})$ 表示当 x 个食饵可以最多支撑 x/N 个捕食者时捕食者数量的增长速率。当 $N=0.5$ 时，不论 x 和 y 的初值如何，捕食者和食饵的数量变化情况都会呈现周期性的增长和减少，如图 11.4 的右子图所示。图 11.4 的左子图显示了相平面 (phase plane) 上 $x(t)$ 和 $y(t)$ 的轨迹最终形成了一个稳定极限环 (stable limit cycle)。

程序 11.9 的第 4 行至第 7 行定义了表示 Holling-Tanner 模型的函数。第 9 行定义的变量 tmax 和 n 分别表示时间上界和点数，x0 和 y0 分别表示捕食者和食饵的初始数量。第 11 行调用 solve_ivp 函数进行求解，返回值 ht.y 是一个二维数组，它的第一行和第二行分别存储了 $x(t)$ 和 $y(t)$。

Listing 11.9:

```
1  import numpy as np; from scipy.integrate import solve_ivp
2  import matplotlib.pyplot as plt
3
4  def Holling_Tanner(t, X):
5      x, y = X
6      return np.array([ x*(1 - x/7) - 6*x*y/(7 + 7*x),
7                        0.2*y*(1 - 0.5*y/x) ])
8
9  tmax = 200; n = 10000; x0, y0 = 6, 0.2
10 t = np.linspace(0, tmax, n)
11 ht = solve_ivp(Holling_Tanner, [0, tmax], [x0, y0], t_eval = t)
12 x = ht.y[0]; y = ht.y[1]
13
14 fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(11,5),
15                                constrained_layout=True)
16
17 ax1.plot(x, y, color = 'g')
18 ax1.set_xlabel('x'); ax1.set_ylabel('y')
19 ax1.set_title('Phase␣portrait')
20
21 ax2.plot(t, y, 'r-', label = 'predator')
22 ax2.plot(t, x, 'b-', label = 'prey')
23 ax2.set_xlabel('time'); ax2.set_ylabel('')
24 ax2.legend(loc='best'); ax2.set_title("Time␣evolution")
25 plt.show()
```

洛伦兹方程 (Lorenz equation)[SL2018] 是描述对流流体运动的一个简化模型：

$$
\begin{aligned}
\frac{dx}{dt} &= \sigma(y-x) \\
\frac{dy}{dt} &= rx - y - xz \\
\frac{dz}{dt} &= xy - bz
\end{aligned}
$$

方程中 x 表示对流翻转的速率 (风速)，y 表示水平方向的温度变化，z 表示竖直方向的温度变化，σ 表示与流体粘度相关的 Prandtl 数，r 表示与温差相关的 Rayleigh 数，b 是一个尺度因

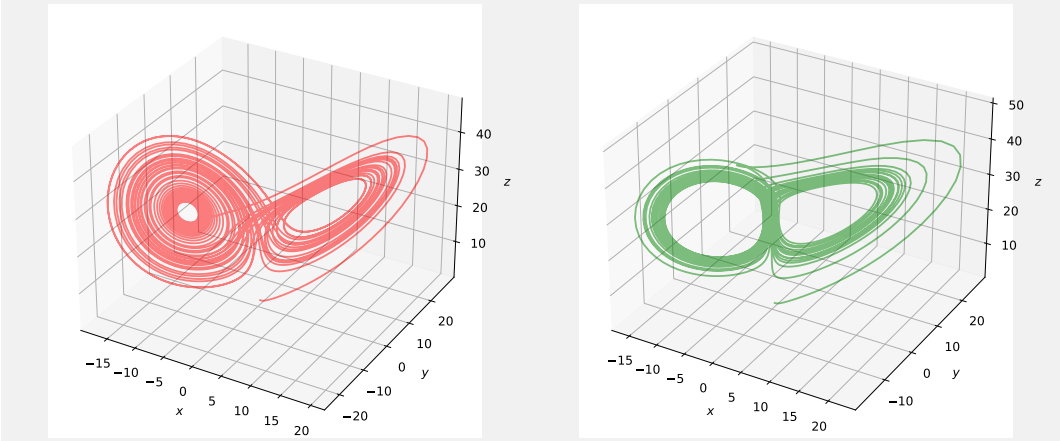


图 11.5: 洛伦兹吸引子

子。对于某些参数值，方程的解显示了混沌 (chaos) 现象。图 11.5 的两个子图展示了由两组参数值生成的三维相空间的两个轨迹，其特点是轨迹以不可预测的方式环绕两个吸引子 (attractor)，环绕的具体形式对初值条件敏感，但环绕的总体形式和初值条件无关。程序 11.10 调用 solve_ivp 函数对两组参数值分别进行求解，返回值 f1.y 和 f2.y 都是二维数组，它们的第一行、第二行和第三行分别存储了 $x(t)$、$y(t)$ 和 $z(t)$。

Listing 11.10: **洛伦兹吸引子**

```
import numpy as np; from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

def Lorenz(t, X, sigma, beta, rho):
    x, y, z = X
    dx_dt = -sigma * (x - y)
    dy_dt = rho * x - y - x * z
    dz_dt = -beta * z + x * y
    return dx_dt, dy_dt, dz_dt

tmax, n = 40, 4000
x0, y0, z0 = 0, 1, 1.05; sigma, beta, rho = 10, 2.667, 28
t = np.linspace(0, tmax, n)
f1 = solve_ivp(Lorenz, [0, tmax], [x0, y0, z0], t_eval = t,
               args=(sigma, beta, rho))
f2 = solve_ivp(Lorenz, [0, tmax], [x0, y0, z0], t_eval = t,
               args=(2*sigma, 0.5*beta, rho))

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12,5),
```

```
                                    constrained_layout=True,
                                    subplot_kw={'projection':'3d'})
for ax, xyz, c in [(ax1, f1.y, 'r'), (ax2, f2.y, 'g')]:
    x, y, z = xyz
    ax.plot(x, y, z, c, alpha=0.5)
    ax.set_xlabel('$x$'); ax.set_ylabel('$y$')
    ax.set_zlabel('$z$')
plt.show()
```

11.3.3 边值问题的数值求解方法

求解边值问题的基本方法有两种：打靶法 (shooting method) 和有限差分法 (finite difference method)[SB2002][KS2021]。打靶法将边值问题转换为多个初值问题进行求解。有限差分法用差商近似计算导数，将边值问题转换为线性方程组进行求解。

使用打靶法求解以下二阶常微分方程的边值问题。

$$y'' = f(x, y, y'), \quad y(a) = \alpha, \quad y(b) = \beta$$

设 $y'(a) = s$，则以条件 $y'(a) = s$ 和 $y(a) = \alpha$ 可以求解一个初值问题，用 $F(x, s)$ 表示它的解。原问题转换为一个代数方程，即求解满足方程 $F(b, s) = \beta$ 的 s，可以使用二分法和牛顿法等方法迭代求解，在求解过程中对每个 s 的迭代值都需要求解一个对应的初值问题。得到代数方程的解 $s = t$ 以后再以条件 $y'(a) = t$ 和 $y(a) = \alpha$ 求解初值问题，即可得到边值问题的解。

使用有限差分法求解以下二阶常微分方程的边值问题。

$$-y'' + q(x)y = g(x), \quad y(a) = \alpha, \quad y(b) = \beta$$

在区间 $[a, b]$ 上选取 $n + 2$ 个等间距的点：$a = x_0 < x_1 < ...x_n < x_{n+1} = b$，相邻两个点之间的间距 $h = (b - a)/(n + 1)$。在点 $x_j (j = 1, ..., n)$ 处使用中心差分公式近似计算二阶导数

$$y''(x_j) \approx \frac{y(x_{j+1}) - 2y(x_j) + y(x_{j-1})}{h^2}$$

可得到如下的线性方程组：

$$\frac{1}{h^2}\begin{pmatrix} 2 + g(x_1)h^2 & -1 & & 0 \\ -1 & 2 + g(x_2)h^2 & \ddots & \\ & \ddots & \ddots & -1 \\ 0 & & -1 & 2 + g(x_n)h^2 \end{pmatrix}\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} g(x_1) + \frac{\alpha}{h^2} \\ g(x_2) \\ \vdots \\ g(x_{n-1}) \\ g(x_n) + \frac{\beta}{h^2} \end{pmatrix}$$

求解该线性方程组得到的向量 $(u_1, u_2, ..., u_n)^T$ 即为 $y(x_1), y(x_2), ..., y(x_n)$ 的近似值。

图 11.6 显示了以下边值问题的求解结果：

$$-y'' + 400y = 800x - 400\cos^2 \pi x - 2\pi^2 \cos 2\pi x, \quad y(0) = 0, \quad y(1) = 2$$

它的解析解是

$$y(x) = 2x + \frac{e^{-20}}{1+e^{-20}} e^{20x} + \frac{1}{1+e^{-20}} e^{-20x} - \cos^2 \pi x$$

左子图中用红色圆点标示了打靶法的求解结果，右子图中用红色圆点标示了有限差分法的求解结果。两个子图中的绿色曲线绘制了解析解的函数图像。

程序 11.11 的第 16 行至第 28 行定义的函数 shooting 实现了打靶法。第 18 行至第 21 行定义的函数 F 以条件 $y'(0) = y1$ 和 $y(0) = 0$ 求解一个初值问题，其返回值是 $y(1)$ 和 2 的差。第 23 行调用 scipy 库的 optimize 模块的 fsolve 函数求解方程 F(x)=0，提供的参数分别是函数 F 和对于 x 的初始估计值 5。fsolve 函数的返回值是一个列表，它的第一个元素是方程 F(x)=0 的解。第 30 行至第 57 行定义的函数 finite_difference 实现了有限差分法。

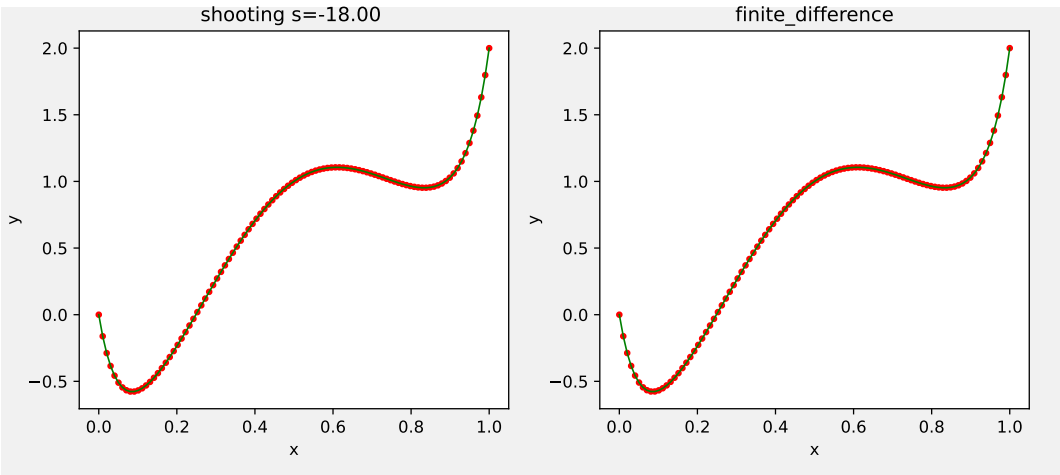


图 11.6: 打靶法和有限差分法求解边值问题

Listing 11.11: 打靶法和有限差分法求解边值问题

```
1  from scipy.optimize import fsolve
2  from scipy.integrate import solve_ivp
3  import matplotlib.pyplot as plt
4  import numpy as np
5
6  f = lambda x, y: \
7          np.array([y[1], -800 * x + 400 * y[0] +
8                          400 * np.cos(np.pi*x) ** 2 +
9                          2 * np.pi ** 2 * np.cos(2 * np.pi*x)])
10
```

```
def y(x):
    c = -20; ec = np.exp(c)
    return 2 * x + ec * np.exp(-c * x) / (1 + ec) + \
           np.exp(c * x) / (1 + ec) - np.cos(np.pi*x) ** 2

def shooting(n, ax):
    y0 = 0; ab = [0, 1]; x_eval = np.linspace(0, 1, n)
    def F(y1):
        sol = solve_ivp(f, ab, [y0, y1], t_eval = x_eval)
        y = sol.y[0]
        return y[-1] - 2

    s, = fsolve(F, 5); print(s)
    res = solve_ivp(f, ab, [y0, s], t_eval = x_eval)
    ax.set_title('shooting␣s=%.2f␣' % s)
    ax.plot(res.t, res.y[0], 'r.');
    ax.plot(x_eval, y(x_eval), 'g', lw=1)
    return res

def finite_difference(n, ax):
   def q(x): return 400
   def g(x): return 800 * x - 400 * np.cos(np.pi*x) ** 2 - \
                    2 * np.pi ** 2 * np.cos(2 * np.pi*x)

   a = 0; b = 1; h = (b - a) / (n + 1); h2 = h * h
   ya = 0; yb = 2
   x_eval = np.linspace(0, 1, n + 2)

   A = np.zeros((n, n))
   A[0, 1] = A[n-1, n-2] = -1;
   A[0, 0] = 2 + q(x_eval[1]) * h2
   A[n-1, n-1] = 2 + q(x_eval[n]) * h2
   for i in range(1, n - 1):
       A[i, i-1] = A[i, i+1] = -1
       A[i, i] = 2 + q(x_eval[i+1]) * h2
   A /= h2

   b = np.zeros(n)
   b[0] = g(x_eval[1]) + ya / h2
```

```
    b[n-1] = g(x_eval[n]) + yb / h2
    b[1:n-1] = g(x_eval[2:n])

    u = np.linalg.solve(A, b)
    v = np.hstack((np.hstack(([ya], u)), [yb]))
    ax.set_title('finite_difference'); ax.plot(x_eval, v, 'r.')
    ax.plot(x_eval, y(x_eval), 'g', lw=1)
    return v

fig, axes = plt.subplots(1, 2, figsize=(9, 4), dpi = 300)
for i in (0, 1):
     axes[i].set_xlabel('x'); axes[i].set_ylabel('y')
fig.tight_layout(); plt.show()

shooting(100, axes[0])
finite_difference(98, axes[1])
```

11.4 实验 11：SciPy 库简介

实验目的

本实验的目的是掌握使用 SciPy 库进行插值、代数方程求解和常微分方程求解。

提交方式

在 Blackboard 提交一个文本文件 (txt 后缀)，文件中记录每道题的源程序和运行结果。

实验内容

1. 使用一元三次样条函数对函数 $y = (5\sin(\frac{3x}{8}) + 3)\cos(\frac{-x^2}{9})$ 上横坐标由数组 np.linspace(0, 10, num=21, endpoint=True) 指定的 21 个数据点进行插值。绘制这些数据点和插值函数的曲线，并标注图例。

2. 使用 optimize 模块的 root 函数求解以下非线性方程组，选择 Levenberg-Marquardt 算法并设置关键字实参 jac 为 True。

$$y\sin(x) = 4 \tag{11.2}$$

$$xy - x = 5 \tag{11.3}$$

答案：
[1.6581955 4.015326]

3. 使用 solve_ivp 函数求解实验 10 第 3 题，t_eval 参数是 np.linspace(0, 10, 1000)。绘制数值解的数据点和解析解的曲线，并标注图例。

第 12 章　统计计算

本章简要介绍使用 scipy 库的 stats 模块和 statsmodels 库 [SS2010][StatsmodelsDoc] 实现以下功能：

- 计算概率分布的函数和特征；
- 生成服从指定分布的随机数；
- 对随机现象进行模拟；
- 通过抽样对关于总体分布的参数值的假设进行检验；
- 根据观测数据构建和拟合线性回归模型。

12.1　概率分布

scipy.stats 提供了统计计算使用的类和函数，包括概率分布和假设检验等。
表 12.1 和表 12.2 分别列出了描述离散概率分布和连续概率分布的一些常用函数和特征，包括 PMF(离散概率分布的概率质量函数)、PDF(连续概率分布的概率密度函数)、CDF(累积分布函数)、Survival Function(生存函数)、PPF(百分比点函数) 和 Moments(矩) 等。连续概率分布的参数 L 和 S 分别表示位置和尺度。对于正态分布，位置 (location) 是期望，尺度参数 (scale) 是标准差。
程序 12.1 获取 scipy.stats 提供的离散概率分布和连续概率分布的类的列表。每个类都有文档说明。对于每一种概率分布，rvs 函数生成指定个数的服从该分布的随机数。

Listing 12.1: 离散和连续概率分布

```
In[1]: from scipy import stats
In[2]: dist_discrete = [d for d in dir(stats)
           if isinstance(getattr(stats, d), stats.rv_discrete)]
In[3]: len(dist_discrete), dist_discrete
Out[3]: (16, ['bernoulli',...'binom', ..., 'poisson',
              'randint',...])
In[4]: dist_continuous = [d for d in dir(stats)
```

函数 (特征) 名称	定义
PMF	$p(x_k) = P[X = x_k]$
CDF	$F(x) = P[X \le x] = \sum_{x_k \le x} p(x_k)$ $F(x_k) - F(x_{k-1}) = p(x_k)$
Survival Function	$S(x) = 1 - F(x) = P[X > x]$
PPF	$G(q) = F^{-1}(q)$
Moments	非中心矩 $\mu'_m = E[X^m] = \sum_k x_k^m p(x_k)$ 中心矩 $\mu_m = E[(X-\mu)^m] = \sum_k (x_k - \mu)^m p(x_k)$ 期望 mean $\mu = \mu'_1 = E[X] = \sum_k x_k p(x_k)$ 方差 variance $\mu_2 = E[(X-\mu)^2]$ $= \sum_{x_k} x_k^2 p(x_k) - \mu^2$ 偏度 skewness $\gamma_1 = \mu_3/\mu_2^{3/2}$ 峰度 kurtosis $\gamma_2 = \mu_4/\mu_2^2 - 3$

表 12.1: 离散概率分布的函数和特征

函数 (特征) 名称	定义 (标准形式)	定义 ($X = L + SY$)
CDF	$F(x) = \int\limits_{-\infty}^{x} f(x)dx$	$F(x; L, S) = F(\frac{x-L}{S})$
PDF	$f(x) = F'(x)$	$f(x; L, S) = \frac{1}{S} f(\frac{x-L}{S})$
Survival Function	$S(x) = 1 - F(x)$	$S(x; L, S) = S(\frac{x-L}{S})$
PPF	$G(q) = F^{-1}(q)$	$G(x; L, S) = L + SG(q)$
Moments	非中心矩 $\mu'_n = E[Y^n]$ 中心矩 $\mu_n = E[(X-\mu)^n]$ 期望 mean μ 方差 variance μ_2 偏度 skewness $\gamma_1 = \mu_3/\mu_2^{3/2}$ 峰度 kurtosis $\gamma_2 = \mu_4/\mu_2^2 - 3$	$E[X^n]$ $E[(X-\mu_X)^n]$ $L + S\mu$ $S^2\mu_2$ γ_1 γ_2

表 12.2: 连续概率分布的函数和特征

```
            if isinstance(getattr(stats, d), stats.rv_continuous)]
In[5]: len(dist_continuous), dist_continuous
Out[5]: (101, ['chi2',...,'gamma',...,'norm',...,'t',...
                'uniform', ...])
In[6]: print(stats.norm.__doc__)
Out[6]: A normal continuous random variable.
        ......
```

程序 12.2 演示了用指定参数创建一个期望为 3 和标准差为 2 的正态分布并输出一些函数值。In[6] 行的 interval 函数返回一个区间，该区间在中位数 (median) 两侧的子区间内的 PDF 函数和 x 轴之间的面积相同，并且该区间内的 PDF 函数和 x 轴之间的面积为指定的实参值 0.95。In[7] 行的 ppf 函数对于实参列表中的每一个表示分位数的元素返回对应的 CDF 函数的自变量值。

Listing 12.2: **创建一个正态分布并输出一些函数值**

```
In[1]: from scipy import stats; x = stats.norm(3, 2)
In[2]: x.mean(), x.median(), x.var(), x.std(), x.stats()
Out[2]: (3.0, 3.0, 4.0, 2.0, (array(3.), array(4.)))
In[3]: [x.moment(n) for n in range(1, 5)]
Out[3]: [3.0, 13.0, 62.99999999999999, 344.99999999999994]
In[4]: x.pdf([2, 3, 4])
Out[4]: array([0.17603266, 0.19947114, 0.17603266])
In[5]: x.cdf([2, 3, 4])
Out[5]: array([0.30853754, 0.5       , 0.69146246])
In[6]: x.interval(0.95)
Out[6]: (-0.9199279690801081, 6.919927969080108)
In[7]: x.ppf([0.025, 0.975])
Out[7]: array([-0.91992797,  6.91992797])
In[8]: x.rvs(4)
Out[8]: array([3.56471374, 4.62667751, 4.02484388, 1.50417612])
```

程序 12.3 绘制了四个不同参数的正态分布的 PDF(图 12.1)。

Listing 12.3: **四个不同参数的正态分布的 PDF**

```
import numpy as np; from scipy import stats
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(6, 4), dpi = 300)

k = np.linspace(-5, 5, 1000)
```

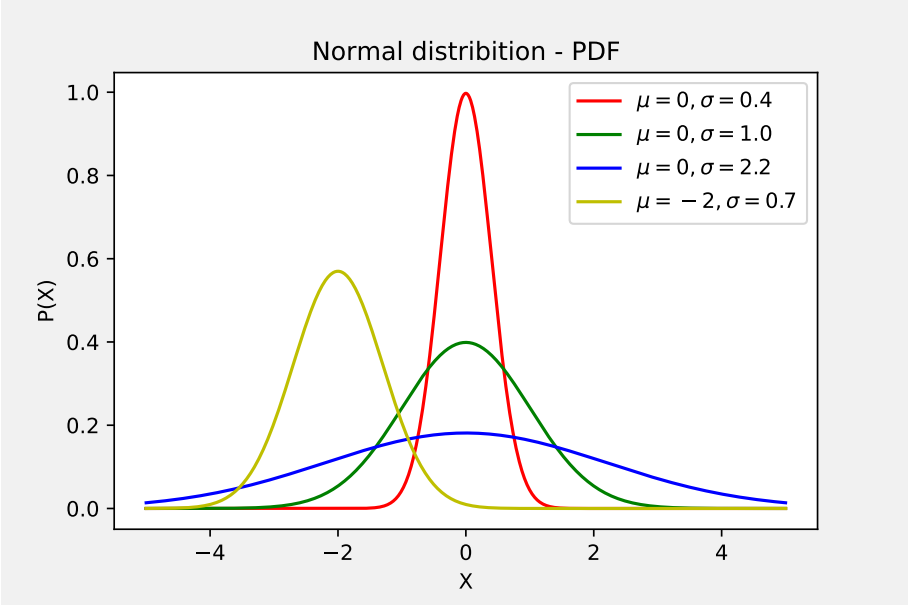


图 12.1: 四个不同参数的正态分布的 PDF

```
pd1 = stats.norm(loc=0, scale=0.4)
pd2 = stats.norm(loc=0, scale=1.0)
pd3 = stats.norm(loc=0, scale=2.2)
pd4 = stats.norm(loc=-2, scale=0.7)

ax.set_title('Normal distribition - PDF');
ax.set_xlabel('X'); ax.set_ylabel('P(X)');
ax.plot(k, pd1.pdf(k), 'r'); ax.plot(k, pd2.pdf(k), 'g');
ax.plot(k, pd3.pdf(k), 'b'); ax.plot(k, pd4.pdf(k), 'y');
ax.legend([r"$\mu=0, \sigma=0.4$", r"$\mu=0, \sigma=1.0$",\
           r"$\mu=0, \sigma=2.2$", r"$\mu=-2, \sigma=0.7$"])
plt.show()
```

程序 12.4 绘制了六个概率分布的 PDF(PMF)、CDF、SF 和 PPF(图 12.2)[RJ2019]。PDF(PMF) 和 x 轴之间的蓝色区域的面积是 0.95，左右边界的 x 值分别是 CDF 的 0.025 分位数和 0.975 分位数。

Listing 12.4: **六个概率分布的 PDF(PMF)、CDF、SF 和 PPF**

```
import numpy as np; import matplotlib.pyplot as plt
from scipy import stats

def plot_distribution(X, i, label):
    x_min_999, x_max_999 = X.interval(0.999)
```

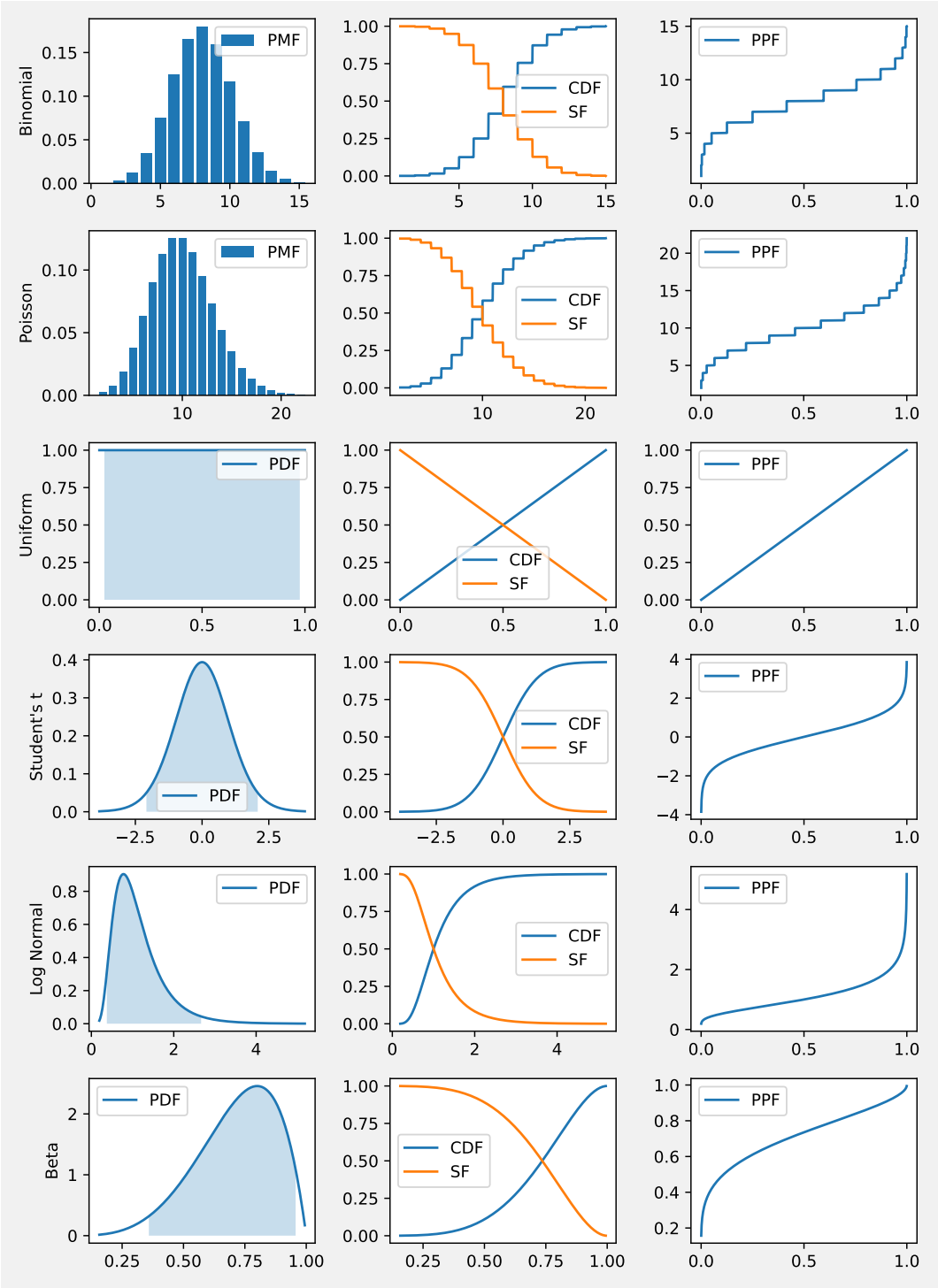


图 12.2: 六个概率分布的 PDF(PMF)、CDF、SF 和 PPF

```
    x999 = np.linspace(x_min_999, x_max_999, 1000)
    x_min_95, x_max_95 = X.interval(0.95)
    x95 = np.linspace(x_min_95, x_max_95, 1000)

    axes = axs[i, :]
    if hasattr(X.dist, "pdf"):   # 连续概率分布
        axes[0].plot(x999, X.pdf(x999), label="PDF")
        axes[0].fill_between(x95, X.pdf(x95), alpha=0.25)
    else:  # 离散概率分布
        x999_int = np.unique(x999.astype(int))
        axes[0].bar(x999_int, X.pmf(x999_int), label="PMF")
    axes[1].plot(x999, X.cdf(x999), label="CDF")
    axes[1].plot(x999, X.sf(x999), label="SF")
    axes[2].plot(X.cdf(x999), x999, label="PPF")

    for ax in axes: ax.legend()
    axes[0].set_ylabel(label)

fig, axs = plt.subplots(6, 3, figsize=(8, 11), dpi = 300)

X = stats.binom(20, 0.4); plot_distribution(X, 0, "Binomial")
X = stats.poisson(10); plot_distribution(X, 1, "Poisson")
X = stats.uniform(); plot_distribution(X, 2, "Uniform")
X = stats.t(20); plot_distribution(X, 3, "Student's␣t")
X = stats.lognorm(0.5); plot_distribution(X, 4, "Log␣Normal")
X = stats.beta(5, 2); plot_distribution(X, 5, "Beta")
fig.tight_layout(); plt.show()
```

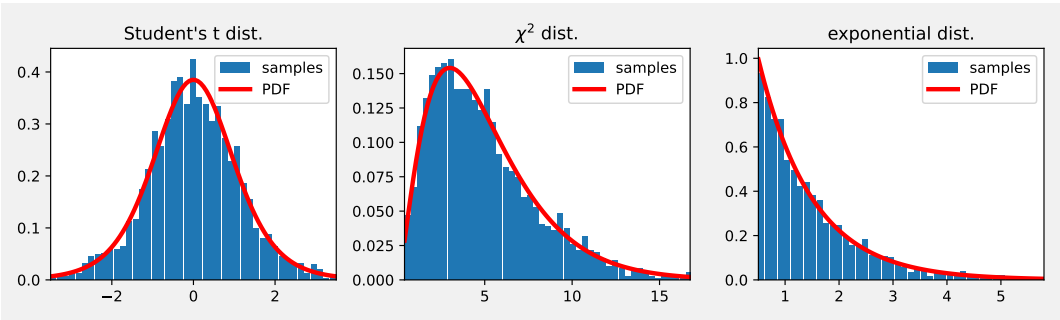


图 12.3: 三个概率分布的 PDF 以及从中抽样得到的随机数的直方图

程序 12.5 调用 plot_dist_samples 函数绘制了三个概率分布的 PDF 以及从中抽样得到的随机

数的直方图 (图 12.3)[RJ2019]。第 7 行调用 linspace 函数的实参 *x_lim 表示将元组 x_lim 分拆成为依次组成它的每个元素作为实参。第 8 行至第 9 行调用 hist 函数绘制数组 X_samples 的直方图，density=True 表示 通过归一化使得直方图与 x 轴之间的面积为 1 以便于和 PDF 进行比较，bins=75 表示绘制 75 个竖条，rwidth=0.85 表示每个竖条的相对宽度。

Listing 12.5: **三个概率分布的 PDF 以及从中抽样得到的随机数的直方图**

```
1  import numpy as np; import matplotlib.pyplot as plt
2  from scipy import stats
3
4  def plot_dist_samples(X, X_samples, title=None, ax=None):
5      x_lim = X.interval(.99)
6      x = np.linspace(*x_lim, num=100)
7      ax.hist(X_samples, label="samples", density=True,
8              bins=75, rwidth=0.85)
9      ax.plot(x, X.pdf(x), 'r', label="PDF", lw=3)
10     ax.set_xlim(*x_lim); ax.legend()
11     if title: ax.set_title(title)
12
13 fig, axes = plt.subplots(1, 3, figsize=(10, 3))
14 N = 2000
15 X = stats.t(7.0)
16 plot_dist_samples(X, X.rvs(N), "Student's␣t␣dist.", ax=axes[0])
17 X = stats.chi2(5.0)
18 plot_dist_samples(X, X.rvs(N), r"$\chi^2$␣dist.", ax=axes[1])
19 X = stats.expon(0.5)
20 plot_dist_samples(X, X.rvs(N), "exponential␣dist.", ax=axes[2])
21 fig.tight_layout(); plt.show()
```

12.2 随机数和模拟

程序 12.6 演示了生成随机数的方法。random 模块的 uniform(a,b) 函数可生成服从区间 [a,b) 内的均匀分布的随机数，这些随机数构成的序列由随机数种子确定。随机数种子的默认值为当前时间。如果需要程序在每次运行时产生相同序列的随机数，需要使用同一个数值设定随机数种子。In[1] 行调用 random 模块的 seed 函数设置随机数种子为 42。

numpy 库的 random 模块提供了多个可生成随机数的函数。In[3] 行调用 numpy 库的 random 模块的 seed 函数设置随机数种子为 26。rand 函数可生成服从区间 [0,1) 内的均匀分布的随机数。如果调用时不提供参数，则生成单个随机数。若提供一个或多个整数作为参数，则它们定义了一个新生成的数组的形状，该数组的每个元素都是由 random 函数生成的一个随机数。

randint 函数生成一个指定形状的数组，由服从某一区间内的均匀分布的随机整数组成，区间的范围 [low,high) 由参数 low 和 high 指定，数组的形状由参数 size 指定。randn 函数可生成服从服从标准正态分布的随机数，用法和 rand 函数类似。

Listing 12.6: **离散和连续概率分布**

```
In[1]: import random; random.seed(42)
In[2]: rands = [random.uniform(-1,1) for i in range(4)]; rands
Out[2]: [0.2788535969157675, -0.9499784895546661,
         -0.4499413632617615, -0.5535785237023545]
In[3]: import numpy as np; np.random.seed(26); np.random.rand()
Out[3]: 0.30793495262497084
In[4]: r = np.random.rand(4); r
Out[4]: array([0.51939148, 0.76829766, 0.78922074, 0.87056206])
In[5]:  r = np.random.rand(2, 2); r
Out[5]: array([[0.18792139, 0.26950525],
               [0.49619214, 0.73912175]])
In[6]:  r = np.random.randint(low=-4, high=5, size=5); r
Out[6]: array([0, 4, 1, 3, 2])
In[7]:  np.random.randint(low=10, high=20, size=(2, 4))
Out[7]: array([[18, 12, 15, 10],
               [12, 13, 13, 13]])
In[8]:  np.random.randn(2, 3)
Out[8]: array([[ 1.46032212,  0.79185191,  0.77784432],
               [-1.5919855 ,  0.73033189,  1.29484354]])
```

很多现象具有不确定性，例如微观粒子的运动、遗传与变异、股票价格的波动等。生成随机数的一个用途是用计算机程序对随机现象进行模拟。

赌徒破产 (Gambler’s Ruin)[NL2017] 是一个经典的随机过程模型。这里假定赌徒和资产为无穷大的庄家对赌。赌徒的初始资产为 x。当赌徒的资产达到目标值 N(N 满足 $0 \le x \le N$) 或者变为 0(即破产) 时赌博过程结束。在每一轮赌博中，赌注为 1 并且赌徒获胜的概率为 p。设 $p_x = P(\text{ruin}|x)$ 表示赌徒以资产 x 开始赌博并最终破产的条件概率。考察第一轮赌博的两种结果：获胜则资产变为 $x+1$，失败则资产变为 $x-1$。由此可得出方程：
$p_x = P(\text{ruin}|x) = P(\text{ruin}|x+1)p + P(\text{ruin}|x-1)(1-p)$。
由此可得递推公式：$p_x = p_{x+1}p + p_{x-1}(1-p)$。代入边界条件 $p_0 = 1$ 和 $p_N = 0$ 后经演算可得结果：

$$p_x = \begin{cases} 1 - \frac{1-((1-p)/p)^x}{1-((1-p)/p)^N} & \text{if } p \neq 0.5 \\ 1 - \frac{x}{N} & \text{if } p = 0.5 \end{cases}$$

程序 12.7 的 gamble 函数利用从二项分布生成的随机数模拟了 1000 次赌博过程来估算 $p = 0.49$ 时的破产概率，连续十次的输出结果依次为：0.608, 0.588, 0.602, 0.583, 0.596, 0.594,

0.608, 0.597, 0.595 和 0.605。它们的平均值 0.5976 和理论计算值 0.5987 很接近。

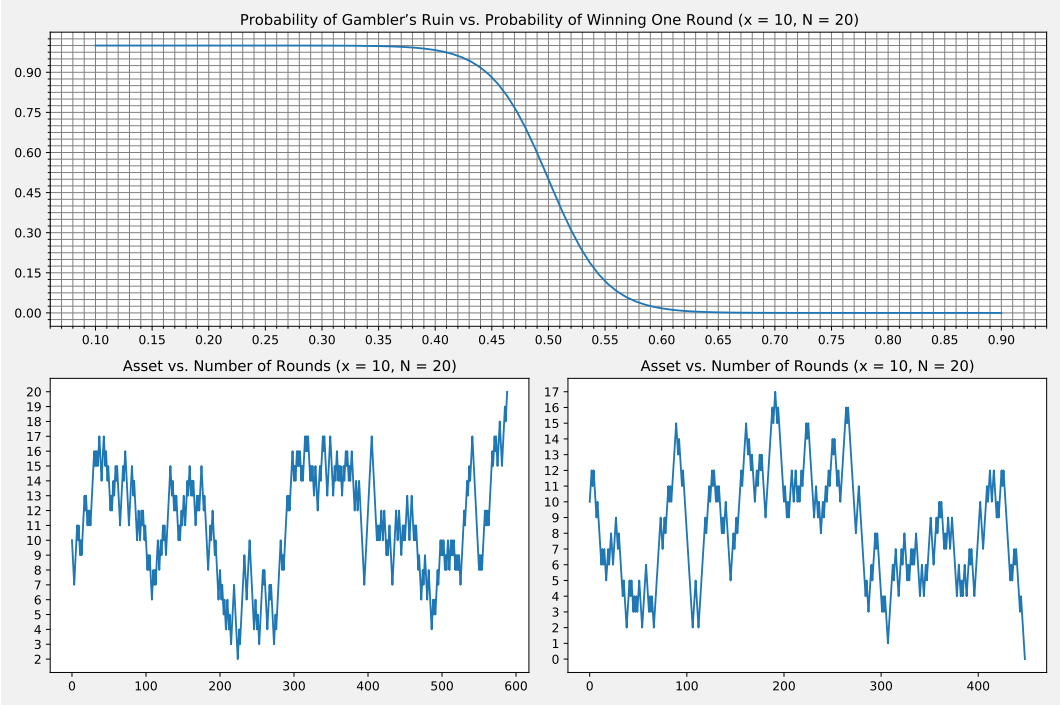


图 12.4: 赌徒破产

图 12.4 包含三个子图，这些子图绘制在由大小相同的方块构成的阵列上，每个子图的位置和大小通过其所占据的方块的行号和列号的范围指定。第一个子图绘制了 p_x 与 p 的关系，可见在 p 接近 0.5 时曲线非常陡峭，即 p 的微小变动导致 p_x 的大幅变动。当 p 从 0.5 逐渐减小时，破产的概率 p_x 急剧增加。其余两个子图分别绘制了两次赌博过程的资产变化情况，它们分别是从资产达到目标值和破产这两种情况的所有赌博过程中选取的轮次最多的例子。

Listing 12.7: **赌徒破产**

```
import numpy as np
import matplotlib as mpl; import matplotlib.pyplot as plt

def gamble(x, N, p):
    x0 = x; s_win = []; s_ruin = []; win = ruin = 0
    for k in range(0, 1000):
        w = [x]
        while x != N and x != 0:
            r = np.random.binomial(1, p, 1)
            x += 1 if r == 1 else -1
            w.append(x)
```

```
        if x == N:
            win += 1
            if len(w) > len(s_win): s_win = w.copy()
        else:
            ruin += 1
            if len(w) > len(s_ruin): s_ruin = w.copy()
        x = x0
    return (ruin / (ruin + win)), s_win, s_ruin

x = 10; N = 20
P, s_win, s_ruin = gamble(x, N, 0.49)
print(P)

p = np.linspace(0.1, 0.9, 100)
c = (1 - p) / p
px = 1 - (1 - c**x) / (1 - c**N)

fig = plt.figure(figsize=(12, 8), dpi = 300)
gs = mpl.gridspec.GridSpec(2, 2)

ax0 = fig.add_subplot(gs[0, :])
ax0.plot(p, px)
ax0.xaxis.set_major_locator(mpl.ticker.MaxNLocator(20))
ax0.xaxis.set_minor_locator(mpl.ticker.MaxNLocator(100))
ax0.yaxis.set_major_locator(mpl.ticker.MaxNLocator(10))
ax0.yaxis.set_minor_locator(mpl.ticker.MaxNLocator(50))
ax0.grid(color="grey", which="major", axis='x', linestyle='-',
         linewidth=0.5)
ax0.grid(color="grey", which="minor", axis='x', linestyle='-',
         linewidth=0.25)
ax0.grid(color="grey", which="major", axis='y', linestyle='-',
         linewidth=0.5)
ax0.grid(color="grey", which="minor", axis='y', linestyle='-',
         linewidth=0.25)
ax0.set_title("Probability of Gambler's Ruin vs. Probability" +
              " of Winning One Round (x = 10, N = 20)")

ax1 = fig.add_subplot(gs[1, 0])
ax1.plot(np.arange(len(s_win)), s_win)
```

```
ax1.yaxis.set_major_locator(mpl.ticker.MaxNLocator(21))
ax1.set_title("Asset vs. Number of Rounds (x = 10, N = 20)")

ax2 = fig.add_subplot(gs[1, 1])
ax2.plot(np.arange(len(s_ruin)), s_ruin)
ax2.yaxis.set_major_locator(mpl.ticker.MaxNLocator(21))
ax2.set_title("Asset vs. Number of Rounds (x = 10, N = 20)")
fig.tight_layout(); plt.show()
```

12.3 假设检验

假设检验 (Hypothesis Testing) 是一种统计推断方法 [JD2011]。统计学中的假设是对参数值或总体分布的判断。假设检验根据从总体中抽样得到的样本的计算结果对两个相互对立的假设做出决策。两个假设分别称为原假设 H_0(null hypothesis) 和备选假设 H_a(alternative hypothesis)。

检验统计量 (test statistic) 是样本数据的函数，服从特定的概率分布。在检验统计量的取值范围中确定一个非空子集称为拒绝域。如果根据样本数据计算的检验统计量的值落入了拒绝域则拒绝原假设并接受备选假设，否则不能拒绝原假设。不能拒绝原假设并不是肯定原假设成立，只是说明当前样本数据没有提供充分的拒绝原假设的证据。

由于抽样的随机性，做出以上决策时难免发生错误。第 I 类错误是在原假设为真的情况下拒绝原假设，第 II 类错误则是在原假设为假的情况下接受原假设。由于实践中犯第 I 类错误导致的后果更严重，需要确保犯第 I 类错误的概率不超过一个主观选定的值 α，称为显著性水平 (significance level)。拒绝域的选取方式是使得检验统计量落入其中的概率为 α。α 通常取值为 0.05，0.01 或更小的值。如果检验统计量的值落入了拒绝域，则认为发生了小概率事件，因此有充分理由拒绝原假设。置信区间是拒绝域的补集，即检验统计量落入置信区间的概率为 $1-\alpha$。

以 z 检验为例说明假设检验的计算过程。当总体服从标准差为 σ 的正态分布时，z 检验对总体的未知期望进行检验。原假设 H_0 为 $\mu=\mu_0$，这里 μ_0 是一个常数。和 H_0 对立的 H_a 可以根据需要选取三种不同的形式：$\mu \neq \mu_0$, $\mu < \mu_0$, $\mu > \mu_0$。这里选取 H_a 为 $\mu \neq \mu_0$。

用 X_1,...,X_n 表示一个从总体中抽样得到的样本，其中包含 n 个个体。假定 H_0 为真，则样本均值 $\overline{X}$ 服从正态分布 $N(\mu_0, \sigma/\sqrt{n})$。选取将 $\overline{X}$ 标准化后得到的统计量 z 作为检验统计量：$z=\frac{(\overline{X}-\mu_0)}{\sigma/\sqrt{n}}$。$z$ 服从标准正态分布。如果样本均值太大或太小，都有充分理由拒绝原假设。拒绝域的形式为 $(-\infty,-c]\cup[c,+\infty)$，其中 c 满足 $\alpha = P(z \geq c \quad \text{or} \quad z \leq -c)$。用 $\Phi(c)$ 表示 $P(z \leq c)$，则 $\alpha = \Phi(-c)+1-\Phi(c) = 2(1-\Phi(c))$。由此可得 $c = z_{\alpha/2}$，拒绝域为

$(-\infty, -z_{\alpha/2}] \cup [z_{\alpha/2}, +\infty)$，其中 $z_{\alpha/2}$ 表示标准正态分布的 $\alpha/2$ 分位数。如果将显著性水平 α 定为 0.05，则拒绝域为 $(-\infty, -1.96] \cup [1.96, +\infty)$。

显著性水平是一个主观选定的值。为了更客观地报告假设检验的结果，可以使用 P 值。P 值表示在假定 H_0 成立的前提下，检验统计量的取值比当前样本计算值更加否定 H_0 的概率。例如从一个样本计算的 z 值为 1.96，则根据本例中拒绝域的形式可计算 P 值为 $P(z \geq 1.96 \quad \text{or} \quad z \leq -1.96) = 2(1 - \Phi(1.96)) = 0.05$。P 值越小，说明当前样本已经提供了越充分的拒绝 H_0 的证据。得出 P 值以后可以无需再计算拒绝域。如果 P 值小于或等于显著性水平 α 则拒绝 H_0，否则不能拒绝 H_0。

这里列举几种常用的假设检验类型和使用方法 (表 12.3)。

1. z 检验：当总体服从标准差为 σ 的正态分布时，z 检验从总体中抽取一个样本对总体的未知期望进行检验。

2. 单样本 t 检验：当总体服从标准差未知的正态分布时，单样本 t 检验从总体中抽取一个样本对总体的未知期望进行检验，公式中的 s 是样本标准差。

3. 双样本 z 检验：当两个总体服从标准差分别为 σ_1 和 σ_2 的正态分布时，双样本 z 检验从它们中各抽取一个样本，对两个总体的未知期望的差进行检验。两个样本的大小分别为 m 和 n。

4. 双样本 t 检验：当两个总体服从标准差未知但相同的的正态分布时，双样本 t 检验从它们中各抽取一个样本，对两个总体的未知期望的差进行检验。两个样本的大小分别为 m 和 n。公式中的 s_p 满足 $s_p^2 = \frac{m-1}{m+n-2}s_1^2 + \frac{n-1}{m+n-2}s_2^2$，其中 s_1 和 s_2 分别是两个样本的样本标准差。

5. 配对 t 检验：配对 t 检验对同一组大小为 n 的个体在不同情况下的观测值所属分布的未知期望的差进行检验。设两组观测值分别为 X_i 和 $Y_i (1 \leq i \leq n)$，假设 $D_i = X_i - Y_i$ 服从正态分布，则可对样本 D_i 进行单样本 t 检验。

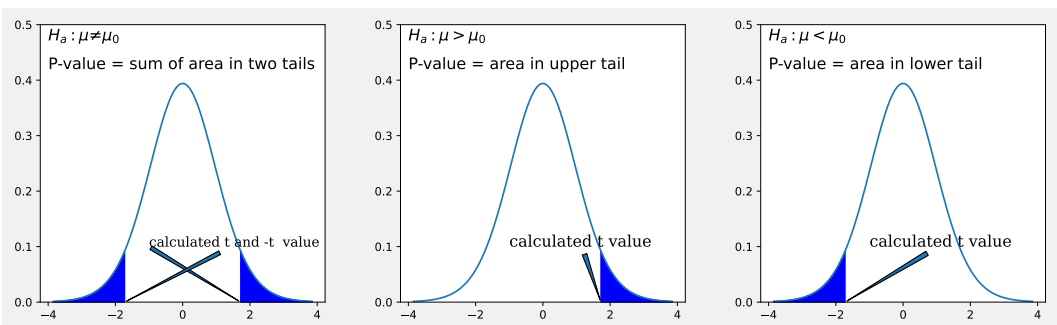


图 12.5: 单样本 t 检验中的 P 值

名称和统计量	原假设 H_0	备选假设 H_a	拒绝域
z 检验 $z=\frac{(\overline{x}-\mu_0)}{\sigma/\sqrt{n}}$	$\mu=\mu_0$	$\mu\neq\mu_0$ $\mu>\mu_0$ $\mu<\mu_0$	$(-\infty,-z_{\alpha/2}]\cup[z_{\alpha/2},+\infty)$ $[z_\alpha,+\infty)$ $(-\infty,-z_\alpha]$
单样本 t 检验 $t=\frac{(\overline{x}-\mu_0)}{s/\sqrt{n}}$	$\mu=\mu_0$	$\mu\neq\mu_0$ $\mu>\mu_0$ $\mu<\mu_0$	$(-\infty,-t_{\alpha/2,n-1}]\cup[t_{\alpha/2,n-1},+\infty)$ $[t_{\alpha,n-1},+\infty)$ $(-\infty,-t_{\alpha,n-1}]$
双样本 z 检验 $z=\frac{(\overline{x}-\overline{y}-\delta_0)}{\sqrt{\frac{\sigma_1^2}{m}+\frac{\sigma_2^2}{n}}}$	$\mu_1-\mu_2=\delta_0$	$\mu_1-\mu_2\neq\delta_0$ $\mu_1-\mu_2>\delta_0$ $\mu_1-\mu_2<\delta_0$	$(-\infty,-z_{\alpha/2}]\cup[z_{\alpha/2},+\infty)$ $[z_\alpha,+\infty)$ $(-\infty,-z_\alpha]$
双样本 t 检验 $t=\frac{(\overline{x}-\overline{y}-\delta_0)}{\sqrt{\frac{s_p^2}{m}+\frac{s_p^2}{n}}}$	$\mu_1-\mu_2=\delta_0$	$\mu_1-\mu_2\neq\delta_0$ $\mu_1-\mu_2>\delta_0$ $\mu_1-\mu_2<\delta_0$	$(-\infty,-t_{\alpha/2,m+n-2}]\cup[-t_{\alpha/2,m+n-2},+\infty)$ $[t_{\alpha,m+n-2},+\infty)$ $(-\infty,-t_{\alpha,m+n-2}]$
配对 t 检验 $t=\frac{(\overline{d}-\delta_0)}{s_D/\sqrt{n}}$	$\mu_D=\delta_0$	$\mu_D\neq\delta_0$ $\mu_D>\delta_0$ $\mu_D<\delta_0$	$(-\infty,-t_{\alpha/2,n-1}]\cup[t_{\alpha/2,n-1},+\infty)$ $[t_{\alpha,n-1},+\infty)$ $(-\infty,-t_{\alpha,n-1}]$

表 12.3: 几种常用的假设检验

scipy.stats 提供了进行多种假设检验的函数。ttest_1samp 函数进行单样本 t 检验。关键字参数 alternative 的默认值为 two-sided，即 H_0 为 $\mu=\mu_0$ 且 H_a 为 $\mu\neq\mu_0$。如果将其设为 greater，则 H_a 为 $\mu>\mu_0$。如果将其设为 less，则 H_a 为 $\mu<\mu_0$。图 12.6 绘制了以上三种情形下根据从样本计算得到的 t 统计量的值计算 P 值的方法。第一种情形下，P 值等于两个尾部区域的面积，这两个区域都位于 t 分布的 PDF 和 x 轴之间，x 值的取值范围分别为 $(-\infty,-|t|]$ 和 $[|t|,+\infty)$。第二种情形下，P 值等于 x 值的取值范围为 $[t,+\infty)$ 的尾部区域的面积。第三种情形下，P 值等于 x 值的取值范围为 $(-\infty,t]$ 的尾部区域的面积。

程序 12.8 演示了基于两组大小均为 40 的样本进行的单样本 t 检验。当检验值 4.0 恰好等于正态总体的均值时 P 值较大，而当检验值 1.0 与正态总体的均值相差较大时 P 值很小。

Listing 12.8: **单样本 t 检验**

```
In[1]: from scipy import stats; import numpy as np
In[2]: rng = np.random.default_rng()
In[3]: rvs = stats.norm.rvs(loc=4, scale=10, size=(40, 2),
                random_state=rng)
In[4]: stats.ttest_1samp(rvs, 4.0)
Out[4]: Ttest_1sampResult(statistic=array([0.52078101,
    0.45723463]), pvalue=array([0.60546376, 0.65003933]))
```

```
In[5]: stats.ttest_1samp(rvs, 1.0)
Out[5]: Ttest_1sampResult(statistic=array([2.33735525,
    2.3660094 ]), pvalue=array([0.0246469 , 0.02304528]))
In[6]: stats.ttest_1samp(rvs, 1.0, alternative='less')
Out[6]: Ttest_1sampResult(statistic=array([2.33735525,
    2.3660094 ]), pvalue=array([0.98767655, 0.98847736]))
In[7]: stats.ttest_1samp(rvs, 1.0, alternative='greater')
Out[7]: Ttest_1sampResult(statistic=array([2.33735525,
    2.3660094 ]), pvalue=array([0.01232345, 0.01152264]))
In[8]: stats.ttest_1samp(rvs, [4.0, 1.0])
Out[8]: Ttest_1sampResult(statistic=array([0.52078101,
    2.3660094 ]), pvalue=array([0.60546376, 0.02304528]))
```

程序 12.9 演示了使用 ttest_ind 函数进行双样本 t 检验。关键字参数 equal_var 的默认值为 True，即假定两个总体的方差相同。如果这一条件不成立，将其设定为 False，则执行 Welch 检验。关键字参数 alternative 的默认值为 two-sided，即 H_0 为 $\mu_1 = \mu_2$ 且 H_a 为 $\mu_1 \neq \mu_2$。如果将其设为 greater，则 H_a 为 $\mu_1 > \mu_2$。如果将其设为 less，则 H_a 为 $\mu_1 < \mu_2$。

Listing 12.9: **双样本 t 检验**

```
In[1]: rvs1 = stats.norm.rvs(loc=4, scale=10, size=400,
                             random_state=rng)
In[2]: rvs2 = stats.norm.rvs(loc=4, scale=10, size=200,
                             random_state=rng)
In[3]: rvs3 = stats.norm.rvs(loc=8, scale=20, size=200,
                             random_state=rng)
In[4]: stats.ttest_ind(rvs1, rvs2)
Out[4]: Ttest_indResult(statistic=0.47568430594388683,
                        pvalue=0.634473019722014)
In[5]: stats.ttest_ind(rvs1, rvs3, equal_var=False)
Out[5]: Ttest_indResult(statistic=-2.375285510987052,
                        pvalue=0.018247576389730322)
In[6]: stats.ttest_ind(rvs2, rvs3, equal_var=False)
Out[6]: Ttest_indResult(statistic=-2.5030356212394462,
                        pvalue=0.012812946996671242)
In[7]: stats.ttest_ind(rvs2, rvs3, equal_var=False,
                       alternative='greater')
Out[7]: Ttest_indResult(statistic=-2.5030356212394462,
                        pvalue=0.9935935265016644)
```

程序 12.10 演示了使用 ttest_rel 函数进行配对 t 检验。这个例子比较同一个班级在两次考试

中的平均成绩是否存在显著不同。

Listing 12.10: **配对 t 检验**

```
In[1]: exam_1 = np.array([79, 100, 93, 75, 84, 107, 66, 86, 103,
    81, 83, 89, 105, 84, 86, 86, 112, 112, 100, 94])
In[2]: exam_2 = np.array([92, 100, 76, 97, 72, 79,  94, 71, 84,
    76, 82, 57, 67,  78, 94, 83, 85,  92,  76,  88])
In[3]: (t, pVal) = stats.ttest_rel (exam_1, exam_2); (t, pVal)
Out[3]: (2.3040209271929544, 0.032682085532223897)
In[4]: stats.ttest_rel (exam_1, exam_2, alternative='less')
Out[4]: Ttest_relResult(statistic=2.3040209271929544,
                        pvalue=0.9836589572338881)
```

12.4 线性回归

线性回归 (Linear Regression) 是一种统计模型，描述了因变量和一个或多个自变量之间的带有不确定性的线性依赖关系。线性回归的用途是可以根据构建的模型从给定的自变量值预测因变量值。

只有一个自变量的简单线性回归模型 [JD2011] 定义为 $Y = \beta_0 + \beta_1 x + \epsilon$。因变量 Y 是一个随机变量，它的每次观测值是自变量 x 的线性函数和一个随机偏差 ϵ 的叠加，后者服从正态分布 $N(0, \sigma^2)$，其中方差 σ^2 和 x 无关。给定观测到的 n 个数据对 $(x_1, y_1), ..., (x_n, y_n)$，最小二乘原理给出的估计为

$$\hat{\beta}_1 = \frac{\sum_{i=1}^n (x_i - \overline{x})(y_i - \overline{y})}{\sum_{i=1}^n (x_i - \overline{x})^2}, \quad \hat{\beta}_0 = \overline{y} - \hat{\beta}_1 \overline{x}, \quad \hat{\sigma}^2 = \frac{SSE}{n-2}$$

其中 $SSE = \sum_{i=1}^n (y_i - \hat{y}_i)^2 = \sum_{i=1}^n y_i^2 - \hat{\beta}_0 \sum_{i=1}^n y_i - \hat{\beta}_1 \sum_{i=1}^n x_i y_i$ 称为残差平方和 (residual sum of squares)。

决定系数 (coefficient of determination) 是一个衡量线性回归模型拟合数据的效果的指标，定义为 $r^2 = 1 - SSE/SST$，其中 $SST = \sum_{i=1}^n (y_i - \overline{y})^2 = \sum_{i=1}^n y_i^2 - (\sum_{i=1}^n y_i)^2/n$ 称为总平方和 (total sum of squares)。决定系数表示在解释因变量随自变量的变化时，回归模型能够说明的部分所占的比例。决定系数的取值范围是区间 [0,1]，它的值越高说明模型拟合数据的效果越好。

有 k 个自变量的多元回归模型定义为 $Y = \beta_0 + \beta_1 x_1 + ... + \beta_k x_k + \epsilon$。给定观测到的 n 个数据对 $([x_{11}, ..., x_{1k}], y_1), ..., ([x_{n1}, ..., x_{nk}], y_n)$，使用矩阵定义的线性回归方程为

$$\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon} = \begin{pmatrix} 1 & x_{11} & \dots & x_{1k} \\ \vdots & \vdots & & \vdots \\ 1 & x_{n1} & \dots & x_{nk} \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_k \end{pmatrix} + \begin{pmatrix} \epsilon_1 \\ \vdots \\ \epsilon_n \end{pmatrix}$$

$\mathbf{X}$ 称为设计矩阵 (design matrix)。最小二乘原理给出的估计为 $\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$。决定系数 $r^2 = 1 - SSE/SST$，其中 $SSE = (\mathbf{y} - \hat{\mathbf{y}})^T(\mathbf{y} - \hat{\mathbf{y}})$，$SST = (\mathbf{y} - \bar{\mathbf{y}})^T(\mathbf{y} - \bar{\mathbf{y}})$。

程序 12.11 演示了使用 numpy.linalg 模块的 lstsq 函数求解回归模型
$Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2 + \epsilon$。

Listing 12.11: **使用 numpy.linalg.lstsq 函数求解线性回归**

```
In[1]: import numpy as np
In[2]: y = np.array([11, 13, 17, 19, 23])
In[3]: x1 = np.array([1, 3, 5, 7, 9])
In[4]: x2 = np.array([2, 4, 6, 8, 10])
In[5]:X = np.vstack([np.ones(5), x1, x2, x1*x2]).T; X
Out[5]:
array([[ 1.,  1.,  2.,  2.],
       [ 1.,  3.,  4., 12.],
       [ 1.,  5.,  6., 30.],
       [ 1.,  7.,  8., 56.],
       [ 1.,  9., 10., 90.]])
In[6]: np.linalg.lstsq(X, y, rcond=None)[0]
Out[6]: array([ 6.10238095, -2.49761905,  3.6047619 ,
                0.03571429])
```

根据观测数据设计一个线性回归模型时，如果模型的决定系数较低就需要修改模型，即重新计算设计矩阵 $\mathbf{X}$。statsmodels 库实现了多种统计模型，包括线性回归、方差分析、多元统计分析和时间序列分析等。使用 Patsy 公式语言定义 statsmodels 库的线性回归模型可以自动计算设计矩阵，因此在修改模型时更加便捷。安装 statsmodels 库的命令是“conda install -c conda-forge statsmodels”。

程序 12.12 的 In[3] 行创建了一个字典，将变量名映射到对应的 Numpy 数组。In[4] 行创建了一个 pandas 模块的 DataFrame 对象，它存储了所有的输入数据。In[5] 行计算设计矩阵，并指定返回类型为 pandas 模块的 DataFrame。In[7] 行使用 OLS 函数创建了线性回归模型，然后使用 fit 函数对其进行拟合。In[8] 行将 DataFrame 格式的数据转换成 Numpy 数组。

Listing 12.12: **使用 statsmodels.OLS 函数求解线性回归**

```
In[1]: import patsy; import pandas as pd
```

```
In[2]: import statsmodels.api as sm
In[3]: data = {"y": y, "x1": x1, "x2": x2}
In[4]: df_data = pd.DataFrame(data)
In[5]: y, X = patsy.dmatrices("y␣~␣1␣+␣x1␣+␣x2␣+␣x1:x2", df_data,
                              return_type="dataframe")
In[6]: X
Out[6]:
Intercept   x1    x2   x1:x2
0         1.0  1.0   2.0    2.0
1         1.0  3.0   4.0   12.0
2         1.0  5.0   6.0   30.0
3         1.0  7.0   8.0   56.0
4         1.0  9.0  10.0   90.0
In[7]: model = sm.OLS(y, X); result = model.fit(); result.params
Out[7]:
Intercept    6.102381
x1          -2.497619
x2           3.604762
x1:x2        0.035714
dtype: float64
In[8]: np.array(X)
Out[8]:
array([[ 1.,  1.,  2.,  2.],
       [ 1.,  3.,  4., 12.],
       [ 1.,  5.,  6., 30.],
       [ 1.,  7.,  8., 56.],
       [ 1.,  9., 10., 90.]])
```

表 12.4 列出了 Patsy 公式语言 (https://patsy.readthedocs.io/en/latest/) 的常用语法 [RJ2019]。

程序 12.13 演示了一些使用 Patsy 公式语言定义线性回归模型的实例。

Listing 12.13: **使用 Patsy 公式语言定义线性回归模型**

```
In[1]: import patsy; from collections import defaultdict
In[2]: data = {k: np.array([]) for k in ["y", "u", "v", "w"]}
In[3]: patsy.dmatrices("y␣~␣u", data=data)[1].
                        design_info.term_names
Out[3]: ['Intercept', 'x']
In[4]: patsy.dmatrices("y␣~␣u␣+␣v", data=data)[1].
                        design_info.term_names
```

语法	含义和示例
lhs ~ rhs	~ 用来分隔模型方程的左边和右边。例: y ~ x 等价于 y ~ 1 + x，其中 1 表示常数项
var1 : var2	两个变量的交互项。例: u : v 等价于 uv
var1 + var2	+用来连接两个变量形成并集。例: y ~ u + v 等价于 y ~ 1 + u + v
var1 * var2	两个变量的所有低阶项和交互项。例: u * v 等价于 u + v + uv
var1 - var2	-用来连接两个变量形成差集。例: y ~ u*v - u:v 等价于 y ~ 1 + u + v
f(expr)	将 Python 函数 f 作用于参数 expr。例: np.log(u), np.cos(u+v)
I(expr)	将 expr 中的运算符解释为数学运算符。例: I(u*v) 解释为 u 和 v 的乘积

表 12.4: Patsy 公式语言的常用语法

```
8  Out[4]: ['Intercept', 'u', 'v']
9  In[5]: patsy.dmatrices("y␣~␣u␣*␣v", data=data)[1].
10                         design_info.term_names
11 Out[5]: ['Intercept', 'u', 'v', 'u:v']
12 In[6]: patsy.dmatrices("y␣~␣u␣*␣v␣*␣w", data=data)[1].
13                         design_info.term_names
14 Out[6]: ['Intercept', 'u', 'v', 'u:v', 'w', 'u:w', 'v:w', 'u:v:w']
15 In[7]: patsy.dmatrices("y␣~␣u␣*␣v␣*␣w␣␣-␣u:v:w", data=data)[1].
16                         design_info.term_names
17 Out[7]: ['Intercept', 'u', 'v', 'u:v', 'w', 'u:w', 'v:w']
18 In[8]: patsy.dmatrices("y␣~␣np.log(u)␣+␣v", data=data)[1].
19                         design_info.term_names
20 Out[8]: ['Intercept', 'np.log(u)', 'v']
21 In[9]: patsy.dmatrices(
22             "y␣~␣I(u**2)␣*␣I(v**2)␣-␣I(u**2)␣:␣I(v**2)␣+␣u*v",
23             data=data)[1].design_info.term_names
24 Out[9]: ['Intercept', 'I(u␣**␣2)', 'I(v␣**␣2)', 'u', 'v', 'u:v']
25 In[10]: z = lambda x, y: np.exp(x * np.cos(y))
26 In[11]: patsy.dmatrices("y␣~␣z(u,␣v)␣+␣w", data=data)[1].
27                          design_info.term_names
28 Out[11]: ['Intercept', 'z(u,␣v)', 'w']
```

程序 12.14 的第 5 行定义了一个基于自变量 x1 和 x2 的二次多项式函数作为真实模型。第 7 行根据随机生成的自变量值计算对应的因变量值，然后在第 9 行叠加服从正态分布的随机偏差以模拟观测值。接着用复杂度从低到高的四个模型依次进行拟合并输出决定系数。前三个模型的决定系数都较低。第四个模型的决定系数接近 1，说明拟合效果较好。

本例中真实模型是已知的，而实践中真实模型通常是未知的。在模型中增加自变量的数量会导致决定系数不断增加，但是过多的变量可能导致过度拟合 (overfitting)，即模型不仅反映了因变量和自变量之间的依赖关系，还反映了数据的随机噪声。过度拟合的模型在预测时的可靠性会降低。

Listing 12.14: **使用 statsmodels.OLS 函数求解线性回归**

```
import numpy as np; import pandas as pd;
import statsmodels.formula.api as smf
N = 100; x1 = np.random.randn(N); x2 = np.random.randn(N)
data = pd.DataFrame({"x1": x1, "x2": x2})
def y_true(x1, x2):
    return 1 - 2 * x1 + 3 * x2 - 4 * x1 * x2 + 5 * x1 * x1 - \
           6 * x2 * x2
data["y_true"] = y_true(x1, x2)
e = 0.5 * np.random.randn(N)
data["y"] = data["y_true"] + e
model = smf.ols("y ~ x1 + x2", data)
result = model.fit(); print(result.rsquared)
# 0.21203317236971864
model2 = smf.ols("y ~ x1 + x2 + x1*x2", data)
result2 = model2.fit(); print(result2.rsquared)
# 0.23144362039459077
model3 = smf.ols("y ~ x1 + x2 + x1*x2 + I(x1*x1)", data)
result3 = model3.fit(); print(result3.rsquared)
# 0.40808316159952807
model4 = smf.ols("y ~ x1 + x2 + x1*x2 + I(x1*x1) + I(x2*x2)", data)
result4 = model4.fit(); print(result4.rsquared)
# 0.9987167686446344
print(result4.summary())
```

程序 12.14 的输出结果 (图 12.6) 包含了模型中需要求解的参数值及其显著性水平为 0.05 的置信区间和多种假设检验的结果。如果拟合的效果较好，则决定系数 (R-squared) 应接近 1 且残差应服从正态分布。残差应服从正态分布的依据有：偏度 (Skew) 应接近 0，峰度 (kurtosis) 应接近 3，Omnibus 检验的 P 值不应太小，Jarque - Bera 检验的 P 值也不应太小。

在某些问题中因变量的取值不连续，例如因变量 y 取值为 1 和 0 分别表示某一事件发生和不发生。这种情形下线性回归模型不适用。设 x 为唯一自变量，需要预测给定 x 值时事件发生的概率：$p(x) = P(y = 1|x)$。由于 $p(x)$ 的取值范围是 [0,1]，所以使用 logit 函数 $f(t) = \log \frac{t}{1-t}$ 对其变换以后作为线性回归模型的因变量，这样得到的回归模型称为单个自变

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.999
Model:                            OLS   Adj. R-squared:                  0.999
Method:                 Least Squares   F-statistic:                 1.463e+04
Date:                Thu, 03 Nov 2022   Prob (F-statistic):          3.10e-134
Time:                        17:27:31   Log-Likelihood:                -49.530
No. Observations:                 100   AIC:                             111.1
Df Residuals:                      94   BIC:                             126.7
Df Model:                           5
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      1.0218      0.060     17.163      0.000       0.904       1.140
x1            -1.9804      0.042    -47.462      0.000      -2.063      -1.898
x2             3.0861      0.039     79.741      0.000       3.009       3.163
x1:x2         -4.1248      0.036   -115.129      0.000      -4.196      -4.054
I(x1 * x1)     4.9894      0.032    155.809      0.000       4.926       5.053
I(x2 * x2)    -6.0636      0.029   -208.003      0.000      -6.122      -6.006
==============================================================================
Omnibus:                        3.402   Durbin-Watson:                   1.984
Prob(Omnibus):                  0.182   Jarque-Bera (JB):                2.409
Skew:                          -0.211   Prob(JB):                        0.300
Kurtosis:                       2.367   Cond. No.                         3.68
==============================================================================
```

图 12.6: **程序 12.14 的输出结果**

量的 logistic 回归模型。

$$\log \frac{p(x)}{1-p(x)} = \beta_0 + \beta_1 x + \epsilon$$

若问题有 k 个自变量 $x_1, ..., x_k$，则 k 个自变量的 logistic 回归模型为

$$\log \frac{p(x_1, ..., x_k)}{1-p(x_1, ..., x_k)} = \beta_0 + \beta_1 x_1 + ... + \beta_k x_k + \epsilon$$

1986 年 1 月 28 日，美国挑战者号航天飞机在进行第 10 次太空任务时，因为右侧固态火箭推进器上面的一个 O 形环失效而导致的连锁反应，在升空后 73 秒时爆炸解体 [1]。程序 12.15 根据历次发射时的 O 形环失效事件和温度数据构建了一个 logistic 回归模型，它可以对于给定的温度 (自变量) 预测 O 形环失效的概率。第 4 行至第 5 行调用 genfromtxt 函数从一个 CSV 文件 (12.16) 中读取数据并保存在一个二维数组 data 中，参数 skip_header=1 表示忽略文件最开始的 1 行，参数 usecols=[1, 2] 表示只读取索引值为 1 和 2 的两列，参数 delimiter=',' 表示数据之间的分隔符为','。CSV 文件中从第 2 行开始的每行包括三个元素，即日期、温度和是否失效 (1 表示失效)。第 6 行先调用 isnan 函数返回一个布尔数组，该数组中对应于 data 中索引值为 1 的列中有'nan'(表示不是一个数) 出现的那些行的元素为 True，其他行的数据为 False；然后使用对这个数组进行非运算的结果对 data 进行索引，再将索引的结果覆盖 data。第 6 行的作用是去除了 data 中索引值为 1 的列中有'nan' 出现的那些行。第 9 行调用 statsmodels 库的 Logit 函数构建 logistic 回归模型。Logit 函数的第一个参数是一个 n 行 1 列的二维数组，存储了 n 个因变量数据 (即是否失效)。Logit 函数的第二个参数是设计矩阵，它的第一列全为 1，第二列包含了 n 个自变量数据 (即温度)。第 10 行对模型进行拟合并输出结果。

[1] https://byuistats.github.io/Statistics-Notebook/Analyses/Logistic%20Regression/Examples/challengerLogisticReg.html

图 12.7 显示了程序 12.15 的输出结果。对应于温度的系数 β_1 的假设检验 $H_0:\beta_1=0$ 和 $H_a:\beta_1\neq 0$ 的 P 值为 0.032。在 0.05 显著性水平上可以拒绝该假设，表明温度对于 O 形环失效发挥重要作用。事故当天的温度为华氏 31 度，代入公式 $P(y=1|x)\approx\frac{e^{15.043-0.232x}}{1+e^{15.043-0.232x}}$ 得到的计算结果为 0.99961，这说明当天 O 形环的失效几乎是必然的。

Listing 12.15: **logistic 回归模型预测 O 形环失效的概率**

```
import numpy as np
import statsmodels.api as sm
inFile = 'D:/Python/dat/challenger_data.csv'
data = np.genfromtxt(inFile, skip_header=1, usecols=[1, 2],
                     delimiter=',')
data = data[~np.isnan(data[:, 1])]
n = data.shape[0]
x = np.hstack((np.ones(n).reshape(n, 1),
               data[:,0].reshape(n, 1)))
logit_mod = sm.Logit(data[:,1], x)
logit_res = logit_mod.fit(); print(logit_res.summary())
```

```
Optimization terminated successfully.
         Current function value: 0.441635
         Iterations 7
                           Logit Regression Results
==============================================================================
Dep. Variable:                      y   No. Observations:                   23
Model:                          Logit   Df Residuals:                       21
Method:                           MLE   Df Model:                            1
Date:                Thu, 03 Nov 2022   Pseudo R-squ.:                  0.2813
Time:                        17:35:40   Log-Likelihood:                -10.158
converged:                       True   LL-Null:                       -14.134
Covariance Type:            nonrobust   LLR p-value:                 0.004804
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         15.0429      7.379      2.039      0.041       0.581      29.505
x1            -0.2322      0.108     -2.145      0.032      -0.444      -0.020
==============================================================================
```

图 12.7: **程序 12.15 的输出结果**

Listing 12.16: **challenger_data.csv**

```csv
Date,Temperature,Damage Incident
04/12/1981,66,0
11/12/1981,70,1
3/22/82,69,0
6/27/82,80,NA
01/11/1982,68,0
04/04/1983,67,0
6/18/83,72,0
8/30/83,73,0
11/28/83,70,0
02/03/1984,57,1
04/06/1984,63,1
8/30/84,70,1
10/05/1984,78,0
11/08/1984,67,0
1/24/85,53,1
04/12/1985,67,0
4/29/85,75,0
6/17/85,70,0
7/29/85,81,0
8/27/85,76,0
10/03/1985,79,0
10/30/85,75,1
11/26/85,76,0
01/12/1986,58,1
1/28/86,31,Challenger Accident
```

第 13 章　最优化方法

在科学研究、工程技术、经济管理等众多领域，很多问题都可以归结为一个数学模型：在一些条件限制下，通过合理决策寻求最优化 (最大化或最小化) 一个或多个目标。解决这些最优化问题的数学分支称为最优化理论和方法。最优化问题的三个基本要素是目标函数、决策变量和约束条件。目标函数是要实现的目标的数学表示。决策变量是决策的数学表示，也是目标函数的自变量。约束条件是对决策变量的取值范围的限制，包括两类：等式约束条件和不等式约束条件。满足所有约束条件的决策变量的取值范围称为可行域。基于给定的数学模型，最优化方法通过执行合适的算法在可行域内寻找可以最优化目标函数的决策变量值，称为最优解。

根据决策变量的取值是连续的还是离散的，最优化问题可分为两类：连续最优化问题和离散最优化问题。以下通过两个实例说明。

1. 纸箱问题 [RS2017]：给定一些硬纸板，需要制作一个纸箱。要求有两点：1) 在其表面积不超过 S 的条件下最大化其体积 V；2) 纸箱的底面和顶面需要使用三层硬纸板。在这个问题中，决策变量是纸箱的长度 (l)、宽度 (w) 和高度 (h)；约束条件是 $l > 0, \quad w > 0, \quad h > 0, \quad 2wh + 2lh + 6wl \leq S$；需要最大化的目标函数是 $V = lwh$。由于决策变量的取值是连续的，这是一个连续最优化问题。

2. 旅行推销员 (Traveling Salesman) 问题 [KT2006]：有一个推销员要到 N 个城市去推销商品，其中任意两个城市之间都有道路连接。对于推销员的旅程的要求有两点：1) 推销员从某个城市出发，最后必须回到那座城市；2) 除了出发城市，其他城市只能到达一次。在这个问题中，决策变量是旅程中从当前城市要前往的下一座城市。需要最小化的目标函数是整个旅程的的长度。由于决策变量的取值是离散的，这是一个离散最优化问题，也是一个著名的 NP 完全问题。

连续最优化问题可分为两类：线性最优化问题和非线性最优化问题。线性最优化问题的目标函数和约束条件均为决策变量的线性函数。非线性最优化问题的目标函数或约束条件是决策变量的非线性函数。本章讨论如下的连续最优化问题：对于一个连续可微目标函数 f，求解其在可行域 $\Omega \subseteq \mathbb{R}^n$ 内的局部极小值：

$$\min_{\mathbf{x}\in\Omega} f(\mathbf{x}), \Omega = \{\mathbf{x}|\mathbf{x} \in \mathbb{R}^n,\ c_i(\mathbf{x}) = 0\,(i = 1, 2, ..., n),\ c_i(\mathbf{x}) \geq 0\,(i = n+1, n+2, ..., n+m)\}$$

约束条件包括 n 个等式约束条件 $c_i(\mathbf{x}) = 0\,(i = 1, 2, ..., n)$ 和 m 个不等式约束条件 $c_i(\mathbf{x}) \geq 0\,(i = n+1, n+2, ..., n+m)$，这些约束条件函数 $c_i(\mathbf{x})$ $(i = 1, 2, ..., n+m)$ 连续可微。线性最优化问题都包含约束条件。非线性最优化问题可分为两类：无约束非线性最优化问题 $(m = n = 0$，即 $\Omega = \mathbb{R}^n)$ 和约束非线性最优化问题 $(m + n > 0)$。如果一个实际问题中需要对目标函数 $g(\mathbf{x})$ 求极大值，则设定 $f(\mathbf{x}) = -g(\mathbf{x})$ 即可转换为符合以上描述的问题。

$\mathbf{x}^* \in \Omega$ 称为 f 的一个局部极小点，如果 $\mathbf{x}^*$ 在 Ω 中的一个邻域内取得局部极小值，即满足条件 $\exists\epsilon > 0, \forall\mathbf{x} \in \Omega, |\mathbf{x} - \mathbf{x}^*| \leq \epsilon \Rightarrow f(\mathbf{x}) \geq f(\mathbf{x}^*)$，其中 $|...|$ 表示某种范数，例如欧式距离。$\mathbf{x}^* \in \Omega$ 称为 f 的的一个全局极小点，如果 $\mathbf{x}^*$ 在 Ω 内取得极小值，即满足条件 $\forall\mathbf{x} \in \Omega, f(\mathbf{x}) \geq f(\mathbf{x}^*)$。如果 f 在凸集 Ω 内是一个凸函数，则 f 的局部极小值也是全局极小值。一般情况下，f 在 Ω 内可能存在多个局部极小值，需要找到所有局部极小值并通过比较才能确定全局极小值。

13.1 非线性最优化问题的解析求解方法

13.1.1 无约束非线性最优化问题的解析求解方法

在无约束非线性最优化问题中，局部极小点满足以下的最优性条件 [JN2006]。

- 一阶必要条件：若 $\mathbf{x}^*$ 是 f 的一个局部极小点，则 f 在 $\mathbf{x}^*$ 处的梯度为零：$\nabla f(\mathbf{x}^*) = \mathbf{0}$。梯度为零的点称为驻点 (stationary point)，可能为局部极小点、局部极大点或鞍点。鞍点的特点是：从该点出发，沿着某个方向函数值增大而沿着另外一个方向函数值减小。

- 二阶必要条件：若 $\mathbf{x}^*$ 是 f 的一个局部极小点，且 f 在 $\mathbf{x}^*$ 的一个开邻域内具有连续的二阶导数，则 f 在 $\mathbf{x}^*$ 处的梯度 $\nabla f(\mathbf{x}^*) = \mathbf{0}$ 并且 f 在 $\mathbf{x}^*$ 处的 Hessian 矩阵 $\nabla^2 f(\mathbf{x}^*)$ 半正定。

- 二阶充分条件：若 f 在 $\mathbf{x}^*$ 的一个开邻域内具有连续的二阶导数，并且 $\nabla f(\mathbf{x}^*) = \mathbf{0}$ 和 $\nabla^2 f(\mathbf{x}^*)$ 正定，则 $\mathbf{x}^*$ 是一个严格局部极小点，严格是指不存在等于的情形：$\exists\epsilon > 0, \forall\mathbf{x} \in \Omega, |\mathbf{x} - \mathbf{x}^*| \leq \epsilon, \mathbf{x} \neq \mathbf{x}^* \Rightarrow f(\mathbf{x}) > f(\mathbf{x}^*)$。

- 如果 f 在凸集 Ω 内是一个凸函数，即在 Ω 内的每个点的 Hessian 矩阵都是半正定的，则任何局部极小点也是全局极小点。

根据最优性条件中的一阶必要条件，可通过求解梯度为零的方程 (组) 找到所有的驻点，然后使用其他条件判断其是否为局部或全局极小点。以下通过两个例子说明。

例 1：线性回归模型需要最小化 SSE：$g(\boldsymbol{\beta}) = |\mathbf{y} - \mathbf{X}\boldsymbol{\beta}|^2 = (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})$。当矩阵 $\mathbf{X}^T\mathbf{X}$ 可逆时，求解方程 $\frac{dg(\boldsymbol{\beta})}{d\boldsymbol{\beta}} = \frac{d}{d\boldsymbol{\beta}}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) = \frac{d}{d\boldsymbol{\beta}}(\mathbf{y}^T\mathbf{y} - \boldsymbol{\beta}^T\mathbf{X}^T\mathbf{y} - \mathbf{y}^T\mathbf{X}\boldsymbol{\beta} + \boldsymbol{\beta}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\beta}) = 0$。使用向量导数公式 $\frac{d(\mathbf{Ax})}{d\mathbf{x}} = \mathbf{A}^T \quad \frac{d(\mathbf{x}^T\mathbf{A})}{d\mathbf{x}} = \mathbf{A} \quad \frac{d(\mathbf{x}^T\mathbf{Ax})}{d\mathbf{x}} = \mathbf{Ax} + \mathbf{A^T x}$ 进行化简得到 $-\mathbf{X}^T\mathbf{y} - \mathbf{X}^T\mathbf{y} + \mathbf{X}^T\mathbf{X}\boldsymbol{\beta} + (\mathbf{X}^T\mathbf{X})^T\boldsymbol{\beta} = \mathbf{0}$，可得 $\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$。$g(\boldsymbol{\beta})$ 的 Hessian 矩阵 $\mathbf{X}^T\mathbf{X}$ 是一个常量且半正定，因此它是一个凸函数，驻点 $\hat{\boldsymbol{\beta}}$ 也是一个全局极小点。

例 2：程序 13.1 生成的图 13.1 的四个子图分别绘制了函数 $z = 2x^3 - 3x^2 + 12xy^2 - 12x - 10y^3 + 3y^2$ 在其四个驻点附近的曲面的三维图示。第 5 行在 SymPy 中定义了这个函数。第 19 行和第 20 行使用 SymPy 库的 diff 函数分别计算梯度向量和 Hessian 矩阵。第 21 行求解所有驻点。第 22 行至第 35 行的循环对于每个驻点输出其坐标 (第 23 行)，计算驻点处的 Hessian 矩阵 (第 25 行) 并输出其特征值 (第 26 行)，绘制四个驻点附近的曲面 (第 27 行至第 35 行)。输出结果显示在 13.3。第一个驻点 (-1, 0) 处的 Hessian 矩阵的特征值 (重数为 2) 是负数，即 Hessian 矩阵是负定的，因此它是局部极大点。第二个驻点 (-16/19, -9/19) 处的 Hessian 矩阵的两个特征值分别是负数和正数，即 Hessian 矩阵是不定的，因此它是鞍点。第三个驻点 (1, 1) 处的 Hessian 矩阵的两个特征值分别是正数和负数，即 Hessian 矩阵是不定的，因此它是鞍点。第四个驻点 (2, 0) 处的 Hessian 矩阵的两个特征值都是正数，即 Hessian 矩阵是是正定的，因此它是局部极小点。

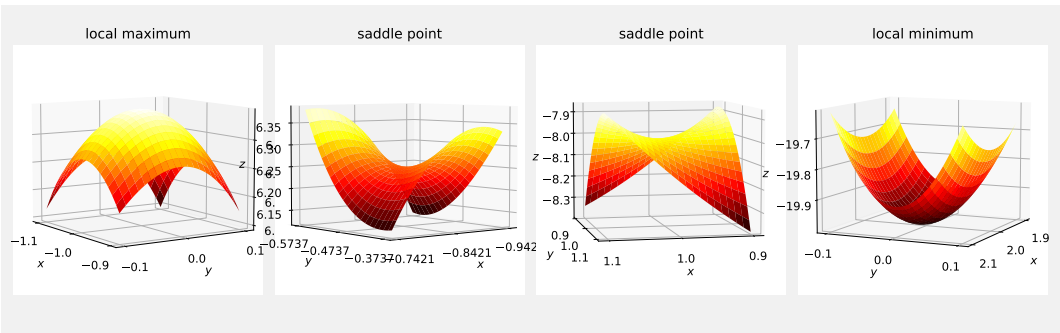


图 13.1: 不同类型的驻点

Listing 13.1: 不同类型的驻点

```
1  import numpy as np;
2  import matplotlib as mpl; import matplotlib.pyplot as plt
3  import sympy as sym
4  x, y = sym.symbols("x, y"); xy = (x, y)
5  z = 2*x**3 - 3*x**2 + 12*x*y**2 - 12*x - 10*y**3 + 3*y**2
6
7  w = 0.1; ea = [(8, -33),(4, 51),(6, 78),(8, 25)]
8  titles = ["local maximum", "saddle point",
9            "saddle point", "local minimum"]
10
11 def title_and_labels(ax, title, rx, ry):
12     ax.set_title(title); ax.set_xlabel("$x$")
13     ax.set_ylabel("$y$"); ax.set_zlabel("$z$")
14     ax.set_xticks(np.linspace(rx-w, rx+w, 3))
15     ax.set_yticks(np.linspace(ry-w, ry+w, 3))
16
```

```
fig, axes = plt.subplots(1, 4, figsize=(18, 4),
                         subplot_kw={'projection': '3d'})
eqns = [z.diff(u) for u in xy]
hess = [[z.diff(u, v) for u in xy] for v in xy]
sols = sym.nonlinsolve(eqns, x, y)
for j, sol in enumerate(sols):
    print(sol, end = '␣')
    rx, ry = float(sol[0]), float(sol[1])
    hess_v = sym.Matrix(hess).subs(([x, rx], [y, ry]))
    print(hess_v.eigenvals())
    xr = np.linspace(rx-w, rx+w, 100)
    yr = np.linspace(ry-w, ry+w, 100)
    X, Y = np.meshgrid(xr, yr)
    f =  sym.lambdify([x, y], z); Z = f(X, Y)
    norm = mpl.colors.Normalize(vmin = Z.min(), vmax = Z.max())
    p = axes[j].plot_surface(X, Y, Z, rcount=20, ccount=20,
                             norm=norm, cmap=mpl.cm.hot)
    title_and_labels(axes[j], titles[j], rx, ry)
    axes[j].view_init(elev=ea[j][0], azim=ea[j][1]);
fig.tight_layout(); plt.show()
```

Listing 13.2: **程序 13.1 的输出结果**

```
(-1, 0) {-18.0000000000000: 2}
(-16/19, -9/19) {-19.8947368421053: 1, 18.0000000000000: 1}
(1, 1) {18.0000000000000: 1, -42.0000000000000: 1}
(2, 0) {18.0000000000000: 1, 54.0000000000000: 1}
```

13.1.2 约束非线性最优化问题的解析求解方法

在约束非线性最优化问题中，设 $\mathbf{x}^*$ 是目标函数 f 的可行域内的一个点，即 $c_i(\mathbf{x}^*) = 0\,(i = 1, 2, ..., n)$，$c_i(\mathbf{x}^*) \geq 0\,(i = n+1, n+2, ..., n+m)$。
定义集合 $T = \{i | c_i(\mathbf{x}^*) = 0,\ n+1 \leq i \leq n+m\}$，集合 $S = \{i | 1 \leq i \leq n\} \cup T$。定义 $\mathbf{G}(\mathbf{x}^*)$ 为以集合 S 中所有元素为下标的约束条件函数在 $\mathbf{x}^*$ 处的梯度 $\{\nabla c_k(\mathbf{x}^*) | k \in S\}$ 的转置为行向量组成的矩阵。定义拉格朗日函数 $\mathbf{L}(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) - \sum_{i=1}^{n+m} \lambda_i c_i(\mathbf{x})$，其中 $\lambda_i\ (i = 1, 2, ..., n+m)$ 称为拉格朗日乘子。局部极小点满足以下的最优性条件 [JN2006]。

- 一阶必要条件：若 $\mathbf{x}^*$ 是一个局部极小点，并且矩阵 $\mathbf{G}(\mathbf{x}^*)$ 是行满秩的 (即所有行向量线性无关)，则存在满足以下三个条件的拉格朗日乘子 $\lambda_i^*\ (i = 1, 2, ..., n+m)$

  1. $\nabla_{\mathbf{x}} \mathbf{L}(\mathbf{x}^*, \lambda^*) = \nabla f(\mathbf{x}^*) - \sum_{i=1}^{n+m} \lambda_i^* \nabla c_i(\mathbf{x}^*) = \mathbf{0}$

2. $\lambda_i^* \geq 0, \quad i = n+1, n+2, ..., n+m$

3. $\lambda_i^* c_i(\mathbf{x}^*) = 0, \quad i = n+1, n+2, ..., n+m$

- 二阶必要条件：若 $\mathbf{x}^*$ 是一个局部极小点，并且矩阵 $\mathbf{G}(\mathbf{x}^*)$ 是行满秩的 (即所有行向量线性无关)，设 λ^* 为一阶必要条件定义的拉格朗日乘子向量，则对于 $\forall \mathbf{d} \in \{\mathbf{w}|\mathbf{G}(\mathbf{x}^*)\mathbf{w} = \mathbf{0}\}$ 不等式 $\mathbf{d}^\mathbf{T}\nabla^2_{\mathbf{xx}}\mathbf{L}(\mathbf{x}^*, \lambda^*)\mathbf{d} \geq 0$ 成立，符号 $\forall$ 表示对于所有的。

- 二阶充分条件：若对于点 $\mathbf{x}^*$ 存在拉格朗日乘子向量 λ^* 使得一阶必要条件列举的三个条件都满足，并且对于 $\forall \mathbf{d} \in \{\mathbf{w}|\mathbf{G}(\mathbf{x}^*)\mathbf{w} = \mathbf{0}, \mathbf{w} \neq \mathbf{0}\}$ 不等式 $\mathbf{d}^\mathbf{T}\nabla^2_{\mathbf{xx}}\mathbf{L}(\mathbf{x}^*, \lambda^*)\mathbf{d} > 0$ 成立，则 $\mathbf{x}^*$ 是 f 的一个严格局部极小点。

程序 13.3 演示了如何求解以下两个问题。
例 3：以 $x + y + z = 1$ 和 $x^2 + y^2 + z^2 \leq 1$ 为约束条件，计算函数 $f(x, y, z) = y + z$ 的局部极小点。
例 4：以 $l > 0, \quad w > 0, \quad h > 0, \quad 2wh + 2lh + 6wl \leq 450$ 为约束条件，计算函数 $f(l, w, h) = lwh$ 的局部极大点。

minz 函数使用 Sympy 求解一阶必要条件定义的非线性方程组。它有五个形参：f 表示目标函数，c_eqs 表示由等式约束条件函数组成的列表，c_ineqs 表示由不等式约束条件函数组成的列表，lambdas 表示由拉格朗日乘子组成的列表，X 表示由决策变量组成的列表，D 表示由二阶充分条件中的向量 $\mathbf{d}$ 的各分量组成的列表。第 8 行至第 14 行检查输入参数中的各列表的长度是否符合要求，若不符合则抛出异常。第 16 行至第 26 行创建一个列表 eqns，它保存了由一阶必要条件的第 1 个和第 3 个条件所定义的各方程。变量 lagrange 表示拉格朗日函数。第 28 行调用 SymPy 库的 nonlinsolve 函数求解 eqns 表示的非线性方程组，并将结果保存在 results 中。第 31 行计算拉格朗日函数的 Hessian 矩阵。第 33 行开始的循环检查 results 中的每个解是不是局部极小点。第 34 至第 35 行排除违反了一阶必要条件规定的拉格朗日乘子需要满足的第二个条件的解。subsX 是一个嵌套列表，它的每个元素是一个由决策变量和其在当前解中的对应值组成的列表。subsA 在 subsX 的基础上添加了每个拉格朗日乘子和其在当前解中的对应值。第 45 行至第 50 行计算 $\mathbf{G}(\mathbf{x}^*)$ 并保存在变量 G 中。第 29 行调用 SymPy 库的 linsolve 函数求解集合 $\{\mathbf{w}|\mathbf{G}(\mathbf{x}^*)\mathbf{w} = \mathbf{0}, \mathbf{w} \neq \mathbf{0}\}$ 并保存在 sols 中。第 56 行至第 58 行对于 sols 中的每个向量 $\mathbf{d}$ 计算 $\mathbf{d}^\mathbf{T}\nabla^2_{\mathbf{xx}}\mathbf{L}(\mathbf{x}^*, \lambda^*)\mathbf{d}$。

例 3 问题有一个等式约束条件和一个不等式约束条件。输出结果 13.4 显示例 3 问题的解有两个，即 (-1/3, 2/3, 2/3, 1/3, -1/2) 和 (1, 0, 0, 1, 1/2)。第一个解的拉格朗日乘子 $\lambda_2^* = -\frac{1}{2} < 0$ 违反了一阶必要条件规定的拉格朗日乘子需要满足的第二个条件，在第 35 行被排除。对于第二个解，计算二阶充分条件中的 $\mathbf{d}$=(0, -d3, d3)。$\mathbf{d} \neq \mathbf{0}$ 的条件要求 d3 $\neq 0$，因此 2*d3**2 $>$ 0，即二阶充分条件满足，(1, 0, 0, 1, 1/2) 是一个严格局部极小点。

在例 4 问题中，设满足所有约束条件的体积最大的纸箱的长度为 l、宽度为 w、高度为 h，则 $2wh + 2lh + 6wl = S$ 必然成立。用反证法证明，若 $2wh + 2lh + 6wl < S$，则存在 $\Delta w > 0$ 使

得 $2(w+\Delta w)h+2lh+6(w+\Delta w)l<S$，即保持长度和高度不变将宽度增加 Δw 使得表面积仍然不违反约束条件，这样得到的纸箱的体积为 $l(w+\Delta w)h>lwh$，导致矛盾 [RS2017]。因此原问题的不等式约束条件 $2wh+2lh+6wl\leq S$ 可转换为等式约束条件 $2wh+2lh+6wl=S$。输出结果 13.4 显示例 4 问题的解有两个，即 (-5, -5, -15, -5/4) 和 (5, 5, 15, 5/4)。第一个解的长度、宽度和高度都是负数，被排除。对于第二个解，计算二阶充分条件中的 **d**=(-d2 - d3/3, d2, d3)。$\mathbf{d}\neq\mathbf{0}$ 的条件要求 d2 $\neq 0$ 或者 d3 $\neq 0$，因此 5*(9*d2**2 + 3*d2*d3 + d3**2)/3 >0，即二阶充分条件满足，(5, 5, 15, 5/4) 是一个严格局部极小点。

Listing 13.3: 约束非线性最优化问题的解析求解方法

```
import sympy as sym

class ParameterMismatchError(Exception):
    def __init__(self, num1, num2):
        self.message = num1 + '␣does␣not␣match␣' + num2

def minz(f, c_eqs, c_ineqs, lambdas, X, D):
    if len(c_eqs) + len(c_ineqs) != len(lambdas):
        raise ParameterMismatchError(
            'number␣of␣lambdas', 'number␣of␣contraints')

    if len(X) != len(D):
        raise ParameterMismatchError(
            'dimension␣of␣X', 'dimension␣of␣D')

    lagrange = f; i = 0; eqns = []
    for c in c_eqs:
        lagrange -= lambdas[i] * c
        eqns.append(c); i += 1
    n = i
    for c in c_ineqs:
        lagrange -= lambdas[i] * c
        eqns.append(lambdas[i] * c); i += 1

    def jac(f): return [f.diff(u) for u in X]
    eqns += jac(lagrange)

    results = sym.nonlinsolve(eqns, *X, *lambdas)
    print(results)

```

```
    hess = [[lagrange.diff(u, v) for u in X] for v in X]

    for r in results:
        if n < len(r):
            if min(r[n:]) < 0: continue
        print(r)
        i = 0; subsX = []
        for u in X:
            subsX.append([u, r[i]]); i += 1
        subsA = subsX.copy(); i = len(X)
        for l in lambdas:
            subsA.append([l, r[i]]); i += 1
        print(subsX, subsA)

        c_jacs = []
        for c in c_eqs:
            c_jacs.append(jac(c))
        for c in c_ineqs:
            if (c.subs(subsX) == 0):
                c_jacs.append(jac(c))
        G = sym.Matrix(c_jacs).subs(subsX)

        sols = sym.linsolve(G * sym.Matrix(D), *D)
        hess_v = sym.Matrix(hess).subs(subsA)

        for s in sols:
            s_m = sym.Matrix(s)
            print((s, sym.factor((s_m.T * hess_v * s_m).det())))

try:
    x, y, z = sym.symbols("x, y, z"); X = (x, y, z)
    d1, d2, d3 = sym.symbols("d1, d2, d3");
    lambda1, lambda2 = sym.symbols("lambda1, lambda2")
    f = y + z; c1 = x + y + z - 1; c2 = 1 - x*x - y*y - z*z
    minz(f, [c1], [c2], [lambda1, lambda2], X, (d1, d2, d3))

    l, w, h = sym.symbols("l, w, h"); X = (l, w, h)
    f = -h*w*l
    c1 = 450 - 2*w*h - 2*l*h - 6*w*l
```

```
    minz(f, [c1], [], [lambda1], X, (d1, d2, d3))
except ParameterMismatchError as ex:
    print(ex.message)
```

Listing 13.4: **程序 13.3 的输出结果**

```
{(-1/3, 2/3, 2/3, 1/3, -1/2), (1, 0, 0, 1, 1/2)}
(1, 0, 0, 1, 1/2)
[[x, 1], [y, 0], [z, 0]] [[x, 1], [y, 0], [z, 0],
 [lambda1, 1], [lambda2, 1/2]]
((0, -d3, d3), 2*d3**2)
{(-5, -5, -15, -5/4), (5, 5, 15, 5/4)}
(-5, -5, -15, -5/4)
[[l, -5], [w, -5], [h, -15]] [[l, -5], [w, -5], [h, -15],
 [lambda1, -5/4]]
((-d2 - d3/3, d2, d3), -5*(9*d2**2 + 3*d2*d3 + d3**2)/3)
(5, 5, 15, 5/4)
[[l, 5], [w, 5], [h, 15]] [[l, 5], [w, 5], [h, 15],
 [lambda1, 5/4]]
((-d2 - d3/3, d2, d3), 5*(9*d2**2 + 3*d2*d3 + d3**2)/3)
```

13.2 非线性最优化问题的数值求解方法

很多非线性最优化问题无法使用解析方法求解，只能通过数值方法迭代求解。迭代的基本过程是从一个可行域的一个起始点出发，在每次迭代时从当前位置移动到一个新的位置，直到终止条件满足。终止条件可以是迭代次数达到了一个预定的上限，或者是梯度的范数小于一个预定的阈值等 [RS2017]。在第 k 次迭代时，从当前位置 $\mathbf{x}_k$ 移动到点 $\mathbf{x}_{k+1}$ 的基本策略有两种：线搜索 (line search) 和信赖域 (trust region)。

1. 线搜索策略首先选择移动的方向 $\mathbf{d}_k$，然后沿这个方向进行步长为 α_k 的移动，α_k 通过最小化一元函数 $f(\mathbf{x}_k + \alpha_k \mathbf{d}_k)$ 获得。

2. 信赖域策略利用收集到的信息在点 $\mathbf{x}_k$ 的附近构造一个函数 m_k(通常为二次函数) 作为 f 的近似，然后在 $\mathbf{x}_k$ 的 δ-邻域求解极小点 $\mathbf{x}^*$，即满足条件 $|\mathbf{x}^* - \mathbf{x}_k| \le \delta$ 和 $\forall \mathbf{x}, |\mathbf{x} - \mathbf{x}_k| \le \delta \Rightarrow f(\mathbf{x}) \ge f(\mathbf{x}^*)$，其中 δ 称为信赖域半径。如果 f 在 $\mathbf{x}^*$ 的函数值相对于点 $\mathbf{x}_k$ 的函数值没有发生符合预期的下降，则缩小 δ 并重新求解 $\mathbf{x}^*$。如果发生了符合预期的下降，则设定 $\mathbf{x}_{k+1}$ 为 $\mathbf{x}^*$，然后进行下一次迭代。如果不仅发生了符合预期的下降而且 $\mathbf{x}^*$ 位于信赖域的边界，则增加 δ。

无约束非线性最优化问题的基本迭代方法包括最速下降法、非线性共轭梯度法、牛顿法和拟牛顿法。这四种方法都可以使用线搜索策略。除了非线性共轭梯度法，其余三种方法都可以

使用信赖域策略 [JN2006]。

1. 最速下降法在第 k 次迭代时选择的移动方向为 $\mathbf{x}_k$ 处函数值下降最快的方向，即负梯度方向 $-\nabla f(\mathbf{x}_k)$。这种方法的优点是仅需计算梯度而不需计算 Hessian 矩阵。缺点是收敛速度较慢，在满足特定条件时收敛阶可达到 1(即线性收敛速度)。

2. 非线性共轭梯度法是用于最优化凸二次函数的线性共轭梯度法的推广，移动方向 $\mathbf{p}_k$ 的迭代公式为 $\mathbf{p}_k = -\nabla f(\mathbf{x}_k) + \beta_k \mathbf{p}_{k-1}$，其中 β_k 的取值保证 $\mathbf{p}_k$ 和 $\mathbf{p}_{k-1}$ 共轭。非线性共轭梯度法的优点是性能优于最速下降法并且无需计算和存储 Hessian 矩阵，因此适用于规模较大的问题。

3. 牛顿法在第 k 次迭代时使用 f 在 $\mathbf{x}_k$ 处进行二阶泰勒展开得到的二次函数 $f(\mathbf{x}_k) + (\mathbf{x}-\mathbf{x}_k)^T \nabla f(\mathbf{x}_k) + \frac{1}{2}(\mathbf{x}-\mathbf{x}_k)^T \nabla^2 f(\mathbf{x}_k)(\mathbf{x}-\mathbf{x}_k)$ 作为其近似。若 $\nabla^2 f(\mathbf{x}_k)$ 正定，设定 $\mathbf{x}_{k+1}$ 为此二次函数的极小点 $\mathbf{x}_k - (\nabla^2 f(\mathbf{x}_k))^{-1} \nabla f(\mathbf{x}_k)$，然后进行下一次迭代。为了避免求矩阵的逆，可以定义 $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$，则每次迭代只需求解线性方程组 $\nabla^2 f(\mathbf{x}_k)\mathbf{s}_k = -\nabla f(\mathbf{x}_k)$ 得到 $\mathbf{s}_k$，然后设定 $\mathbf{x_{k+1}} = \mathbf{x_k} + \mathbf{s_k}$。牛顿法的优点是收敛速度较快，在满足特定条件时收敛阶至少为 2。一个缺点是局部收敛，需要针对 Hessian 矩阵非正定的情形进行改进。另一个缺点是每次迭代都需要计算 Hessian 矩阵和求解线性方程组，当 n 较大时耗费较大的时间和空间开销。牛顿共轭梯度法是一种适用于大规模问题的非精确牛顿法，它利用共轭梯度法迭代求解线性方程组 $\nabla^2 f(\mathbf{x}_k)\mathbf{s}_k = -\nabla f(\mathbf{x}_k)$ 并且无需直接计算和存储 Hessian 矩阵。

4. 为减少计算代价，拟牛顿法对牛顿法进行改进，在每次迭代时根据梯度信息更新一个对称矩阵 $\mathbf{B}$ 作为 Hessian 矩阵的近似，通过求解线性方程组 $\mathbf{B}_k\mathbf{s}_k = -\nabla f(\mathbf{x}_k)$ 得到 $\mathbf{s}_k$。在第 k 次迭代时计算的 $\mathbf{B}_{k+1}$ 需要满足割线方程 $\mathbf{B}_{k+1}\mathbf{s}_k = \mathbf{y}_k$，其中 $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$，$\mathbf{y}_k = \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k)$。更新矩阵 $\mathbf{B}_k$ 的公式有多种，其中最为著名的是 SR1(symmetric-rank-one) 公式

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \frac{(\mathbf{y}_k - \mathbf{B}_k\mathbf{s}_k)(\mathbf{y}_k - \mathbf{B}_k\mathbf{s}_k)^T}{(\mathbf{y}_k - \mathbf{B}_k\mathbf{s}_k)^T\mathbf{s}_k}$$

和 BFGS(Broyden, Fletcher, Goldfarb, and Shanno) 公式

$$\mathbf{B}_{k+1} = \mathbf{B}_k - \frac{\mathbf{B}_k\mathbf{s}_k\mathbf{s}_k^T\mathbf{B}_k}{\mathbf{s}_k^T\mathbf{B}_k\mathbf{s}_k} + \frac{\mathbf{y}_k\mathbf{y}_k^T}{\mathbf{y}_k^T\mathbf{s}_k}$$

在实现 BFGS 公式时也可以更新一个对称矩阵 $\mathbf{H}$ 作为 Hessian 矩阵的逆的近似，从而直接计算 $\mathbf{s}_k = -\mathbf{H}_k \nabla f(\mathbf{x}_k)$

$$\mathbf{H}_{k+1} = (\mathbf{I} - \rho_k\mathbf{s}_k\mathbf{y}_k^T)\mathbf{H}_k(\mathbf{I} - \rho_k\mathbf{y}_k\mathbf{s}_k^T) + \rho_k\mathbf{s}_k\mathbf{s}_k^T, \quad \rho_k = \frac{1}{\mathbf{y}_k^T\mathbf{s}_k}$$

这样可以避免求解线性方程组。

适合大规模约束非线性最优化问题的数值求解方法包括序列二次规划 (sequential quadratic programming, SQP) 和内点法 (interior-point methods)。序列二次规划在在第 k 次迭代时求解一个子问题以获得搜索方向 $\mathbf{p}_k$：

$$\min_{\mathbf{p}\in\Omega_k} \frac{1}{2}\mathbf{p}^{\mathbf{T}}\nabla^2_{\mathbf{xx}}\mathbf{L}(\mathbf{x}^{\mathbf{k}},\lambda^{\mathbf{k}})\mathbf{p} + \nabla f(\mathbf{x}^{\mathbf{k}})^T\mathbf{p}$$

$$\Omega_k = \{\mathbf{p}|\nabla c_i(\mathbf{x_k})^T\mathbf{p}+c_i(\mathbf{x_k}) = 0\,(i=1,2,...,n),\ \nabla c_i(\mathbf{x_k})^T\mathbf{p}+c_i(\mathbf{x_k}) \geq 0\,(i=n+1,n+2,...,n+m)\}$$

内点法在每次迭代时求解一个子问题

$$\min_{\mathbf{x}\in,\mathbf{s}} f(\mathbf{x}) - \mu\sum_{i=1}^{n}\log s_i$$

$$\Omega = \{\mathbf{x}|c_i(\mathbf{x}) = 0\,(i=1,2,...,n),\ c_i(\mathbf{x}) - s_{i-n} = 0\,(i=n+1,n+2,...,n+m)\}$$

其中 $s_i > 0$ 称为松弛变量。引入松弛变量可将不等式约束条件转换为等式约束条件。当 s_i 接近 0 时，目标函数的第二项 $-\mu\sum_{i=1}^{n}\log s_i$ 的不断增长产生一种障碍，从而保证不等式约束条件成立。随着迭代次数的增加，μ 逐渐向 0 接近，子问题的解最终收敛到最优解。

scipy 库的 optimize 模块的 minimize 函数提供了一个通用的接口进行无约束和约束最优化计算。它的第一个参数是要求解极小点的函数，第二个参数是起始点。关键字实参 method 指定使用的最优化算法。关键字实参 jac、hess 和 hessp 分别指定梯度、Hessian 矩阵和一个计算 Hessian 矩阵与任意向量的乘积的函数。minimize 函数的运行结果包括多项信息，其中“fun”表示收敛到的极小值，“nit”表示迭代次数，数组“x”表示收敛到的极小点。

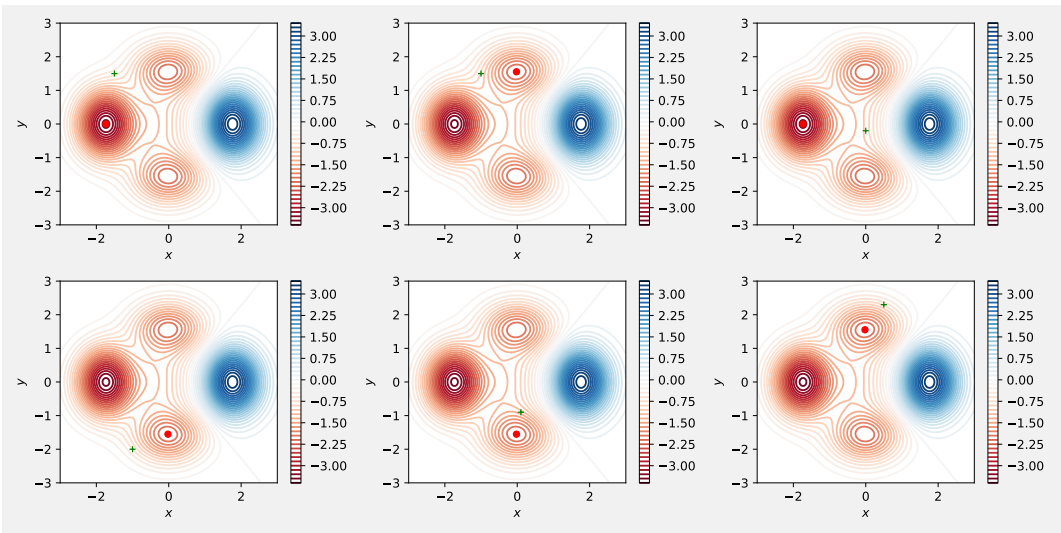


图 13.2: 无约束非线性最优化问题的数值求解方法

程序 13.5 演示了使用无约束非线性最优化问题的数值求解方法计算函数 $z = (x^7 - y^6 + x^5 - y^4 + x^3 - y^2 + x - 1)e^{-x^2-y^2}$ 的极小值。第 7 行和第 8 行利用 SymPy 分别计算了梯度和 Hessian 矩阵，然后将它们的自变量通过 take_array_as_arg 函数转换为一个

数组。第 29 行至第 30 行定义的 starts 列表存储了六个起始点的坐标。第 33 行至第 34 行的循环调用 plot_minimizer 函数，从每个起始点出发使用 BFGS 拟牛顿法计算极小值并绘制等高线图。图 13.2 中绿色加号表示起始点，红色圆点表示最终收敛到的局部极小点。可见起始点的位置对于收敛到的极小点的位置有很大影响。

第 42 行至第 43 行使用 BFGS 拟牛顿法从起始点 (0.1, 0.3) 出发计算极小点，gtol 选项用来指定迭代终止时梯度范数必须小于的数值。结果显示极小值为-3.4991690399798148，迭代次数为 11，极小点坐标为 (-1.73450121e+00, 2.61727718e-12)。牛顿共轭梯度法可通过两种策略实现：线搜索和信赖域。线搜索牛顿共轭梯度法、信赖域牛顿共轭梯度法和信赖域 krylov 子空间方法都适用于规模较大的最优化问题。在使用这些方法时，既可提供 Hessian 矩阵，也可提供一个计算 Hessian 矩阵与任意向量的乘积的函数。对于规模较大的问题，存储和计算 Hessian 矩阵耗费较大的时间和空间开销。这些算法仅需要 Hessian 矩阵和任意向量的乘积，当 Hessian 矩阵较为稀疏时 (例如 Rosenbrock 函数)，可以定义一个函数返回预先计算的乘积，从而减少时间和空间开销。第 43 行至第 47 行使用线搜索牛顿共轭梯度法进行最优化，xtol 选项用来指定迭代终止时解的平均相对误差必须小于的数值。第 45 行通过参数 hess 提供了 Hessian 矩阵，而第 47 行通过参数 hessp 提供了一个计算 Hessian 矩阵与任意向量的乘积的函数 get_hessp。第 48 行至第 49 行使用信赖域牛顿共轭梯度法计算极小点。第 50 行至第 51 行使用信赖域 krylov 子空间方法进行最优化，这种方法相对于信赖域牛顿共轭梯度法的优点是在 Hessian 矩阵非正定的情形时可减少迭代次数。

Listing 13.5: **无约束非线性最优化问题的数值求解方法**

```
1  import numpy as np; from scipy import optimize as optm;
2  import sympy as sym; import matplotlib.pyplot as plt
3
4  x, y = sym.symbols("x, y"); xy = (x, y)
5  f_sym = (x**7 - y**6 + x**5 - y**4 + x**3 - y**2 + x - 1) * \
6          sym.exp(-x**2 -y**2)
7  jac_sym = [f_sym.diff(u) for u in xy]
8  hess_sym = [[f_sym.diff(u, v) for u in xy] for v in xy]
9
10 def take_array_as_arg(f):
11     return lambda x: np.array(f(x[0], x[1]))
12
13 f_ = sym.lambdify(xy, f_sym, 'numpy')
14 f_a = take_array_as_arg(f_)
15 jac_ = sym.lambdify(xy, jac_sym, 'numpy')
16 jac_a = take_array_as_arg(jac_)
17 hess_ = sym.lambdify(xy, hess_sym, 'numpy')
18 hess_a = take_array_as_arg(hess_)
```

```python

def plot_minimizer(start, ax):
    x_opt = optm.minimize(f_a, start, method='BFGS', jac = jac_a)
    c = ax.contour(X, Y, f_(X, Y), 50, cmap=plt.cm.RdBu)
    ax.plot(*start, 'g+', markersize=5)
    ax.plot(x_opt.x[0], x_opt.x[1], 'ro', markersize=5)
    ax.set_xlabel(r"$x$"); ax.set_ylabel(r"$y$")
    plt.colorbar(c, ax=ax)
    return ax

starts = [(-1.5, 1.5), (-1, 1.5), (0, -0.2), (-1, -2),
          (0.1, -0.9), (0.5, 2.3)]
x_ = y_ = np.linspace(-3, 3, 256); X, Y = np.meshgrid(x_, y_)
fig, axes = plt.subplots(2, 3, figsize=(12, 6))
for i in range(len(starts)):
    plot_minimizer(starts[i], axes[i//3, i%3])

fig.tight_layout()

def get_hessp(x, p):
    H = np.array(hess_(x[0], x[1]))
    return H.dot(p)

optm.minimize(f_a, (0.1, 0.3), method='BFGS', jac = jac_a,
              options={'gtol': 1e-8})
optm.minimize(f_a, (0.1, 0.3), method='Newton-CG', jac = jac_a,
              hess = hess_a, options={'xtol': 1e-8})
optm.minimize(f_a, (0.1, 0.3), method='Newton-CG', jac = jac_a,
              hessp = get_hessp, options={'xtol': 1e-8})
optm.minimize(f_a, (0.1, 0.3), method='trust-ncg', jac = jac_a,
              hess = hess_a, options={'gtol': 1e-8})
optm.minimize(f_a, (0.1, 0.3), method='trust-krylov', jac = jac_a,
              hess = hess_a, options={'gtol': 1e-8})
plt.show()
```

程序 13.6 演示了使用信赖域内点法求解约束非线性最优化问题。图 13.3 的左子图显示了从五个起始点出发对以下问题的计算结果，绿色圆点表示起始点，红色加号表示最终收敛到的局部极小点。

$$\min_{(x,y)\in\Omega}(x^7-y^6+x^5-y^4+x^3-y^2+x-1)e^{-x^2-y^2}$$

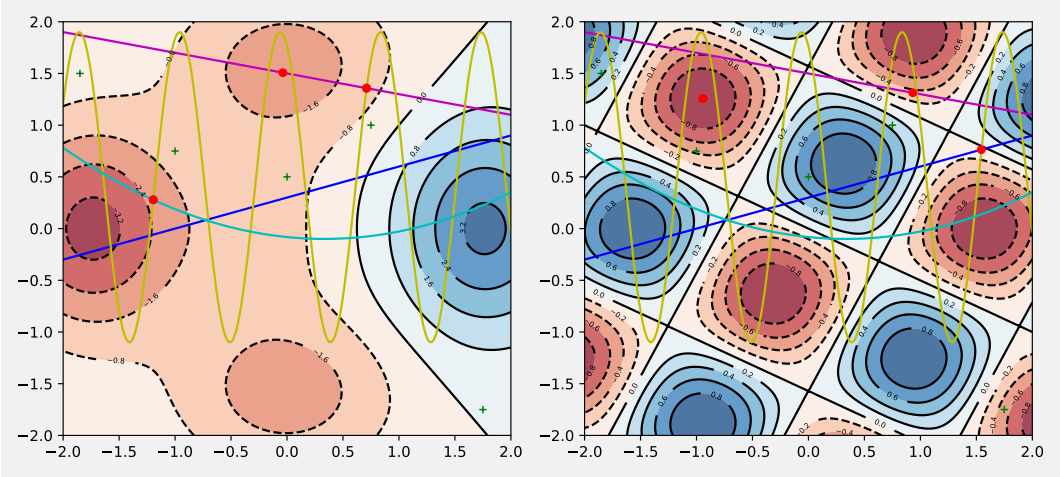


图 13.3: 约束非线性最优化问题的数值求解方法

$$
\begin{aligned}
\Omega = \{(x,y)| \quad & -2 \le x \le 2, \ \ -2 \le y \le 2, \\
& y \le -0.2x + 1.5, \ \ y \ge 0.3x + 0.3, \\
& y \le 1.5\sin(7x+2) + 0.4, \ \ y \ge 0.2(0.9x - 0.3)^2 - 0.1\}
\end{aligned}
$$

从程序 13.6 的输出结果 13.7 可发现以下现象：即使起始点不在可行域内，也可以收敛到可行域内的局部极小点，例如从 (1.75 -1.75) 出发 收敛到 (0.711 1.358)。

图 13.3 的右子图显示了从五个起始点出发对以下问题的计算结果，绿色圆点表示起始点，红色加号表示最终收敛到的局部极小点。

$$
\min_{(x,y)\in\Omega} \sin(x+2y)\cos(2x-y)
$$

Ω 的定义和前一个问题相同。从输出结果 13.7 观察到的现象和前一个问题类似。

程序 13.6 的第 64 行至第 67 行调用 minimize 函数：method='trust-constr' 表示使用信赖域内点法；jac 和 hess 分别指定梯度和 Hessian 矩阵；constraints 指定一个列表，由线性约束条件和非线性约束条件组成；bounds 表示可行域的下界和上界。本例中的线性约束条件可以写成以下标准形式：

$$
\begin{pmatrix} -\infty \\ -\infty \end{pmatrix} \le \begin{pmatrix} 0.2 & 1 \\ 0.3 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \le \begin{pmatrix} 1.5 \\ -0.3 \end{pmatrix}
$$

第 44 行至第 45 行基于这一形式创建了一个 LinearConstraint 对象。本例中的非线性约束条件可以写成以下标准形式：

$$
\begin{pmatrix} -\infty \\ -\infty \end{pmatrix} \le \begin{pmatrix} -1.5\sin(7x+2) + y \\ 0.2(0.9x-0.3)^2 - y \end{pmatrix} \le \begin{pmatrix} 0.4 \\ 0.1 \end{pmatrix}
$$

第 47 行至第 57 行基于这一形式创建了一个 NonlinearConstraint 对象，cons_f 函数定义了约束条件，cons_J 函数定义了约束条件的梯度，cons_H 函数定义了约束条件的 Hessian 矩阵的线性组合。

Listing 13.6: 约束非线性最优化问题的数值求解方法

```
import numpy as np; from scipy import optimize as optm
import sympy as sym; import matplotlib.pyplot as plt

def take_array_as_arg(f):
    return lambda x: np.array(f(x[0], x[1]))
def f1(x,y):
    return (x**7 - y**6 + x**5 - y**4 + x**3 - y**2 + x - 1) * \
           np.exp(-x**2 - y**2)
def f2(x,y):
    return np.sin(x + 2 * y) * np.cos(2 * x - y)

x, y = sym.symbols("x, y"); xy = (x, y)
fig, ax = plt.subplots(1, 2, figsize=(10, 4.5))
for k in range(2):
    if k == 0:
        f_sym = (x**7 - y**6 + x**5 - y**4 + x**3 - y**2 + \
                 x - 1) * sym.exp(-x**2 - y**2)
        f = f1
    else:
        f_sym = sym.sin(x + 2 * y) * sym.cos(2 * x - y)
        f = f2

    f_J_sym = [f_sym.diff(u) for u in xy]
    f_H_sym = [[f_sym.diff(u, v) for u in xy] for v in xy]

    f_a = take_array_as_arg(sym.lambdify(xy, f_sym, 'numpy'))
    f_J_a = take_array_as_arg(sym.lambdify(xy, f_J_sym, 'numpy'))
    f_H_a = take_array_as_arg(sym.lambdify(xy, f_H_sym, 'numpy'))

    x1 = np.linspace(-2, 2, 256); y1 = np.linspace(-2, 2, 256)
    X,Y = np.meshgrid(x1, y1)

    C = ax[k].contour(X, Y, f(X, Y), 10, colors='k')
    ax[k].clabel(C, inline=1, fontsize=5)
    ax[k].contourf(X, Y, f(X, Y), 10, alpha=.75,
                   cmap=plt.cm.RdBu)
    ax[k].plot(x1, -0.2*x1 + 1.5, color='m')
    ax[k].plot(x1, 0.3*x1 + 0.3, color='b')
```

```
    ax[k].plot(x1, 1.5 * np.sin(7*x1+2) + 0.4, color='y')
    ax[k].plot(x1, 0.2 * (0.9*x1-0.3)**2 - 0.1, color='c')

    bounds = optm.Bounds([-2, -2], [2.0, 2.0])

    linear = optm.LinearConstraint([[0.2, 1], [0.3, -1]],
        [-np.inf, -np.inf], [1.5, -0.3])

    def cons_f(x):
        return [-1.5*np.sin(7*x[0]+2) + x[1],
                0.2 * (0.9*x[0]-0.3)**2 - x[1]]
    def cons_J(x):
        return [[-10.5*np.cos(7*x[0]+2), 1],
                [0.36 * (0.9*x[0]-0.3), -1]]
    def cons_H(x, v):
        return v[0]*np.array([[73.5*np.sin(7*x[0]+2), 0], \
               [0, 0]]) + v[1]*np.array([[0.324, 0], [0, 0]])
    nonlinear = optm.NonlinearConstraint(cons_f,
        [-np.inf, -np.inf], [0.4, 0.1], jac=cons_J, hess=cons_H)

    starts = [(-1.85, 1.5), (-1, 0.75), (0, 0.5), (0.75, 1.0),
              (1.75, -1.75)]

    for i in range(len(starts)):
        ax[k].plot(*starts[i], 'g+', markersize=5)
        res = optm.minimize(f_a, starts[i],
                  method='trust-constr', jac=f_J_a, hess=f_H_a,
                  constraints=[linear, nonlinear],
                  options={'verbose': 0}, bounds=bounds)
        p = res.x;
        print('(%4.2f␣%4.2f)␣->␣(%6.3f␣%6.3f)␣␣%6.3f' %
              (*starts[i], *p, f_a(p)))
        ax[k].plot(p[0], p[1], 'ro', markersize=5)

plt.tight_layout(); plt.show()
```

Listing 13.7: 程序 **13.6** 的输出结果

```
(-1.85 1.50) -> (-1.195  0.278)  -2.197
(-1.00 0.75) -> (-1.195  0.278)  -2.197
```

```
(0.00 0.50) -> (-0.037  1.507)  -2.080
(0.75 1.00) -> ( 0.711  1.358)  -1.066
(1.75 -1.75) -> ( 0.711  1.358)  -1.066
(-1.85 1.50) -> (-0.942  1.257)  -1.000
(-1.00 0.75) -> (-0.942  1.257)  -1.000
(0.00 0.50) -> (-0.942  1.257)  -1.000
(0.75 1.00) -> ( 0.932  1.314)  -0.346
(1.75 -1.75) -> ( 1.545  0.763)  -0.048
```

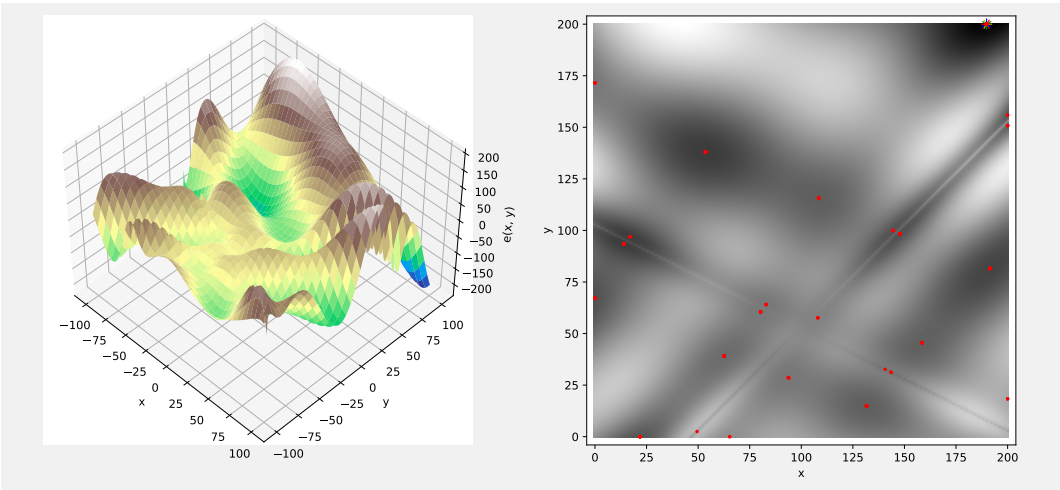


图 13.4: 全局优化

借助求解局部极小点的方法，全局优化在由给定的各自变量的上下界所定义的区域内计算目标函数的全局极小点。程序 13.8 演示了使用四种全局优化方法在正方形区域 $[-100, 100] \times [-100, 100]$ 内求解 eggholder 函数

$$e(x, y) = (-y + 47)\sin(\sqrt{|y + \frac{x}{2} + 47|}) - x\sin(\sqrt{|x - (y + 47)|})$$

的全局极小点。这四种方法是差分进化 (differential_evolution)、双重退火 (dual_annealing)、单纯同调 (shgo, simplicial homology global optimization) 和 sobol 单纯同调 (shgo_sobol)。第四种方法返回找到的所有局部极小点，从中可以确定全局极小点。运行结果表明四种方法在这个问题中都找到了全局极小点 (89.7988,100) 和对应的极小值-227.1231。图 13.5 的左子图显示了函数的三维表面图。右子图显示了函数的二维灰度图，每个像素点的明暗程序反映了该点处的函数值大小，图中的红色圆点显示第四种方法找到的局部极小点，用不同颜色的三叉符号标示的四种方法找到的全局极小点重合在一起。

Listing 13.8: **全局优化**

```
import numpy as np; from scipy import optimize as optm;
import matplotlib.pyplot as plt
```

```python

N = 100; r = 29
def e(x):
        return (-(x[1] + 47) *
            np.sin(np.sqrt(abs(x[0]/2 + (x[1] + 47)))) -
                x[0] * np.sin(np.sqrt(abs(x[0] - (x[1] + 47)))))

bounds = [(-N, N), (-N, N)]
x = np.arange(-N, N+1); y = np.arange(-N, N+1)
xgrid, ygrid = np.meshgrid(x, y)
xy = np.stack([xgrid, ygrid])
fig = plt.figure(figsize=(13, 6))
gs = fig.add_gridspec(1, 2)
ax1 = fig.add_subplot(gs[0, 0], projection='3d')
ax1.view_init(45, -45)
ax1.plot_surface(xgrid, ygrid, e(xy), cmap='terrain')
ax1.set_xlabel('x'); ax1.set_ylabel('y')
ax1.set_zlabel('e(x, y)')

results = dict()
results['DE'] = optm.differential_evolution(e, bounds)
results['DA'] = optm.dual_annealing(e, bounds)
results['shgo'] = optm.shgo(e, bounds)
results['shgo_sobol'] = optm.shgo(e, bounds, n=200, iters=5,
                                  sampling_method='sobol')

ax2 = fig.add_subplot(gs[0, 1])
im = ax2.imshow(e(xy), interpolation='bilinear',
                origin='lower', cmap='gray')
ax2.set_xlabel('x'); ax2.set_ylabel('y')
ax2.set_xlim([-4, (N+2)*2]); ax2.set_ylim([-4, (N+2)*2])

def plot_point(res, marker='+', color=None):
    ax2.plot(N+res.x[0], N+res.x[1], marker=marker, color=color,
             ms=10)
plot_point(results['DE'], color='g', marker='1')
plot_point(results['DA'], color='b', marker='2')
plot_point(results['shgo'], color='r', marker='3')
plot_point(results['shgo_sobol'], color='y', marker='4')
```

```python

min = 1e10; p = [0,0]
for i in range(results['shgo_sobol'].xl.shape[0]):
    ax2.plot(N + results['shgo_sobol'].xl[i, 0],
             N + results['shgo_sobol'].xl[i, 1], 'ro', ms=2)
    v = e([results['shgo_sobol'].xl[i, 0],
           results['shgo_sobol'].xl[i, 1]])
    if v < min:
        min = v; p = results['shgo_sobol'].xl[i]
print(p, min) # [ 89.79881536 100.        ] -227.1231059205501
plt.show()

for method in ['DE', 'DA', 'shgo']:
    p = results[method].x
    print(method, p, e(p))
# DE [ 89.79881536 100.        ] -227.1231059205501
# DA [ 89.79881536 100.        ] -227.1231059205501
# shgo [ 89.79881536 100.        ] -227.1231059205501
```

13.3 线性最优化问题的求解方法

线性最优化问题 ((也称为线性规划问题，linear program) 的目标函数和约束条件均为决策变量的线性函数。线性最优化问题有可能无解，例如可行域为空集，或者由于决策变量的取值无界而导致目标函数在需要最优化的方向上的取值也无界。在非空集的前提下，线性最优化问题的可行域是一个多胞体 (polytope)，即由若干 $\mathbb{R}^n$ 中的超平面 (二维空间中为直线，三维空间中为平面) 围成的凸流通集，可能有界也可能无界。如果线性最优化问题存在最优解，则至少有一个最优解是多胞体的一个顶点 [JN2006]。

图 13.5 的左子图绘制了以下问题的可行域和目标函数的等高线图:

$$\max_{(x,y)\in\Omega} 4x+3y, \Omega=\{(x,y)|0\le x\le 16, 0\le y\le 15, 3x+y\ge 9, x+2y\le 40\}$$

可行域是以六个红色圆点为顶点的多边形。目标函数的等高线是多条互相平行的直线，其中直线 $4x+3y=100$ 和可行域的交集是多边形的一个顶点 (16,12)。易知点 (16,12) 是该问题的最优解。图 13.5 的右子图绘制了以下问题的可行域和目标函数的等高线图:

$$\max_{(x,y)\in\Omega} x+2y, \Omega=\{(x,y)|0\le x\le 16, 0\le y\le 15, 3x+y\ge 9, x+2y\le 40\}$$

这个问题的可行域与前一个问题相同，区别在于目标函数。目标函数的等高线也是多条互相平行的直线，其中直线 $x+2y=40$ 和可行域的交集是多边形上的一条边，即两个顶点

(10,15) 和 (16,12) 之间的线段。易知这条线段上的所有点都是该问题的最优解。

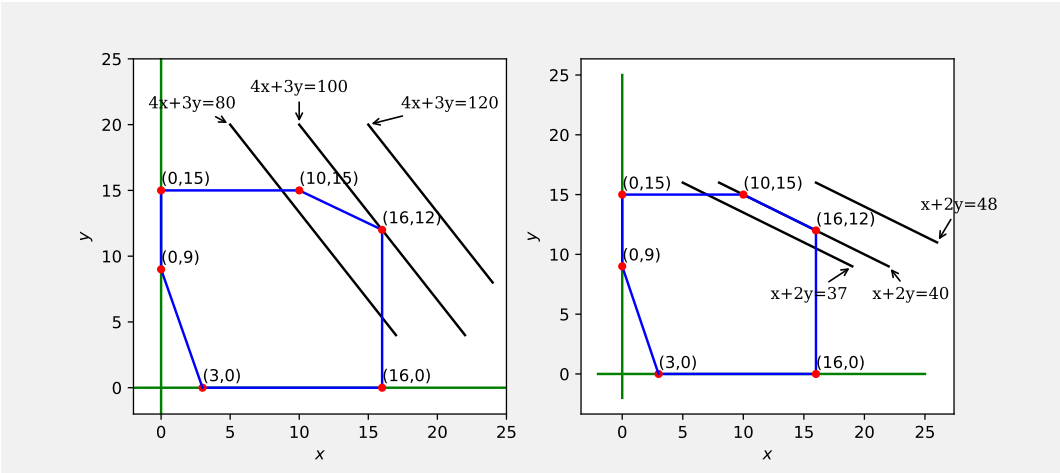


图 13.5: 线性最优化问题

求解线性最优化问题的方法称为线性规划 (linear programming)。适于求解大规模问题的线性规划方法包括单纯形法 (simplex method) 和内点法。单纯形法的求解过程是从可行域的某个顶点出发开始迭代，每次迭代到达一个相邻的并且使得目标函数值改善的顶点，直至找不到这样的顶点为止。内点法的求解过程是从可行域内部的某个点出发，每次迭代时求解一个子问题

$$\min_{\mathbf{x}\in,\mathbf{s}} f(\mathbf{x}) - \mu \sum_{i=1}^{n} \log s_i$$

$$\Omega = \{\mathbf{x}|c_i(\mathbf{x}) = 0\,(i = 1, 2, ..., n),\ c_i(\mathbf{x}) - s_{i-n} = 0\,(i = n+1, n+2, ..., n+m)\}$$

其中 $s_i > 0$ 称为松弛变量。引入松弛变量可将不等式约束条件转换为等式约束条件。当 s_i 接近 0 时，目标函数的第二项 $-\mu\sum_{i=1}^{n}\log s_i$ 的不断增大长产生一种障碍，从而保证不等式约束条件成立。随着迭代次数的增加，μ 逐渐向 0 接近，子问题的解最终收敛到最优解。以 N 和 M 分别表示决策变量的个数和约束条件的个数，单纯形法在最坏情形下的时间复杂度是 N 的指数函数，实验测量得出的结论是 $T \approx 0.18\min\{M, N\}^{1.42}$[RV2014]。内点法在最坏情形下的时间复杂度是 N 的多项式函数。

scipy 库的 optimize 模块的 linprog 函数提供了一个通用的接口求解线性最优化问题，默认方法是‘interior-point’(内点法)。可使用关键字参数 method 指定的其他方法有‘highs’(运行速度最快)、‘revised simplex’(改进的单纯形法，比内点法更精确) 和‘simplex’(单纯形法)。该函数的要求是问题需要转换为以下的标准形式：

$$\min_{\mathbf{x}\in} \mathbf{c}^{\mathbf{T}}\mathbf{x},\ \Omega = \{\mathbf{x}|\mathbf{l} \le \mathbf{x} \le \mathbf{u},\ \mathbf{A}_{ub}\mathbf{x} \le \mathbf{b}_{ub},\ \mathbf{A}_{eq}\mathbf{x} = \mathbf{b}_{eq}\}$$

程序 13.9 演示了使用 linprog 函数求解以下问题：

$$\max_{(x,y,z)\in\Omega} x - 3z$$

$\Omega = \{(x,y,z)|x \geq -3, 4 \leq y \leq 10, z \leq 10, 3x-2y-z \geq 10, 5x+7y-6z \leq 30, 4x+8y-5z = -16\}$

原问题转换成为符合 linprog 函数的要求的标准形式：

$$\min_{(x,y,z)} \begin{pmatrix} -1 & 0 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \text{ subject to } \begin{pmatrix} -3 \\ 4 \\ -\infty \end{pmatrix} \leq \begin{pmatrix} x \\ y \\ z \end{pmatrix} \leq \begin{pmatrix} +\infty \\ 10 \\ 10 \end{pmatrix},$$

$$\begin{pmatrix} -3 & 2 & -1 \\ 5 & 7 & -6 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \leq \begin{pmatrix} -10 \\ 30 \end{pmatrix}, \begin{pmatrix} 4 & 8 & -9 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -16 \end{pmatrix}$$

Listing 13.9: **使用 linprog 函数求解线性最优化问题**

```python
import numpy as np
from scipy.optimize import linprog
c = np.array([-1, 0, 3])
A_ub = np.array([[-3, 2, 1], [5, 7, -6]])
b_ub = np.array([-10, 30])
A_eq = np.array([[4, 8, -9]]); b_eq = np.array([-16])
x_bounds = (-3, np.inf); y_bounds = (4, 10)
z_bounds = (-np.inf, 10)
bounds = [x_bounds, y_bounds, z_bounds]
result = linprog(c, A_ub=A_ub, b_ub=b_ub, A_eq=A_eq,
                 b_eq=b_eq, bounds=bounds, method='highs')
print(result.message)
# Optimization terminated successfully.
print(result.x, c @ result.x)
# [9.13043478 4.         9.39130435] 19.043478260869563
```

13.4 混合整数线性最优化问题的求解方法

混合整数线性最优化问题 (也称为混合整数线性规划，mixed integer linear program) 是一类特殊的线性最优化问题，特殊性体现在某些决策变量必须取整数值。混合整数线性规划问题的一般形式是：

$$\min\{z = cx + hy : (x,y) \in S\}, \quad S = \{(x,y) \in \mathbb{Z}_+^n \times \mathbb{R}_+^p : Ax + Gy \leq b\}$$

其中 c 和 h 分别是包含 n 个分量和 p 个分量的行向量，需要求解的 x 和 y 分别是包含 n 个分量和 p 个分量的列向量，A 和 G 分别是 $m \times n$ 和 $m \times p$ 的矩阵。这里约定：问题存在有限最小值；满足所有约束条件的 (x,y) 称为可行解，简称为解；使得目标函数值 (以下简称为 z 值) 达到有限最小值的解称为最优解。以上问题的一个特例是 $p = 0$，此时的问题称为纯整数线性

规划问题 (pure integer linear program)。

混合整数线性规划问题 (以下简称为整数问题) 的自然线性松弛 (natural linear relaxation，以下简称为松弛) 是：

$$\min\{z = cx + hy : (x, y) \in P\},\ \ P = \{(x, y) \in \mathbb{R}_+^n \times \mathbb{R}_+^p : Ax + Gy \le b\}$$

它和整数问题的区别在于：需要求解的向量 x 的各分量的取值范围在整数问题中是非负整数值，在松弛中是非负实数值。松弛的某个解若满足 $x \in \mathbb{Z}_+^n$，则称为松弛的整数解。由于整数问题的可行域是松弛的可行域的子集，以下性质成立：

性质 1：设整数问题在 (x^*, y^*) 取得最小值 z^* 并且松弛在 (x^0, y^0) 取得最小值 z^0，则 $z^0 \le z^*$。

性质 2：若松弛无解 (infeasible)，即松弛的可行域为空集，则整数问题也无解。

性质 3：若松弛的最优解同时是整数解，则这个解也是整数问题的最优解。

此外，向一个线性最优化问题 (以下简称为原问题) 添加新的约束得到的新问题的可行域是原问题的可行域的子集，因此以下性质成立：

性质 4：设原问题在 (x^1, y^1) 取得最小值 z^1 并且新问题在 (x^2, y^2) 取得最小值 z^2，则 $z^1 \le z^2$。

若一个混合整数线性规划问题的可行域是一个由有限个元素组成的集合，理论上可以通过穷举所有元素获得最优解。对于大多数实际问题，混合整数线性规划问题的基本求解方法是分支定界法 (Branch-and-Bound) 和切割平面法 (Cutting Plane)[CC2014]。它们的共同特点是在求解过程中不断添加约束条件，这些约束条件通过改变松弛的的可行域用以去除松弛的非整数最优解，最终得到的松弛的最优解就是整数问题的最优解。

13.4.1 分支定界法

若一个线性最优化问题 (以下简称为现有问题) 存在最优解 (x^0, y^0)，并且向量 x^0 的某个分量 $x_j^0 (1 \le j \le n)$ 取非整数值，则可以向现有问题分别添加两个约束条件 $x_j \le \lfloor x_j^0 \rfloor$ 和 $x_j \ge \lceil x_j^0 \rceil$，得到两个新问题，这个过程称为分支。分支去除了现有问题的部分可行域，即 x_j 满足 $\lfloor x_j^0 \rfloor < x_j < \lceil x_j^0 \rceil$ 的部分，这部分不包含现有问题的整数解。因此，现有问题的整数解的集合等于两个新问题的整数解的并集。若现有问题存在最优整数解 (x^*, y^*)，则它必定是两个新问题之一的最优整数解。分支定界法在求解过程中不断向现有问题添加上述约束条件生成两个新问题，新问题又可以作为现有问题生成两个新问题，这些问题形成了一种树状结构。分支定界法在求解过程中还记录和更新一个变量 z^b，它表示当前已经找到的原始整数问题的一个解的 z 值。用 z^c 表示现有问题的最优解的 z 值，现有问题的任何整数解的 z 值必定大于或等于 z^c，z^c 可看成现有问题的整数解的下界，这就是定界的含义。若 $z^c < z^b$，则现有问题的可行域有可能包含 z 值小于 z^b 的整数解，需要从现有问题分支以便进一步求解。当某些条件满足时，无需从现有问题分支，分为以下几种情形：

1. 现有问题的最优解是整数解，因此也是原始整数问题的一个可行解。用 z^c 表示其 z 值，若 $z^c < z^b$，则将 z^b 更改为 z^c。

2. 现有问题无解。

3. 现有问题的最优解不是整数解，并且其 z 值 (用 z^c 表示) 满足 $z^c > z^b$。因为分支不改变现有问题的整数解的集合，通过分支不可能找到 z 值小于 z^c(更不可能小于 z^b) 的整数解。

以下描述使用分支定界法求解最大化问题

$$\max\{z = x + y : (x, y) \in S\}$$
$$S = \{(x, y)|17x + 32y \leq 136, 32x + 15y \leq 120, x \in \mathbb{Z}_+, y \in \mathbb{Z}_+\}$$

的过程 (图 13.6)。

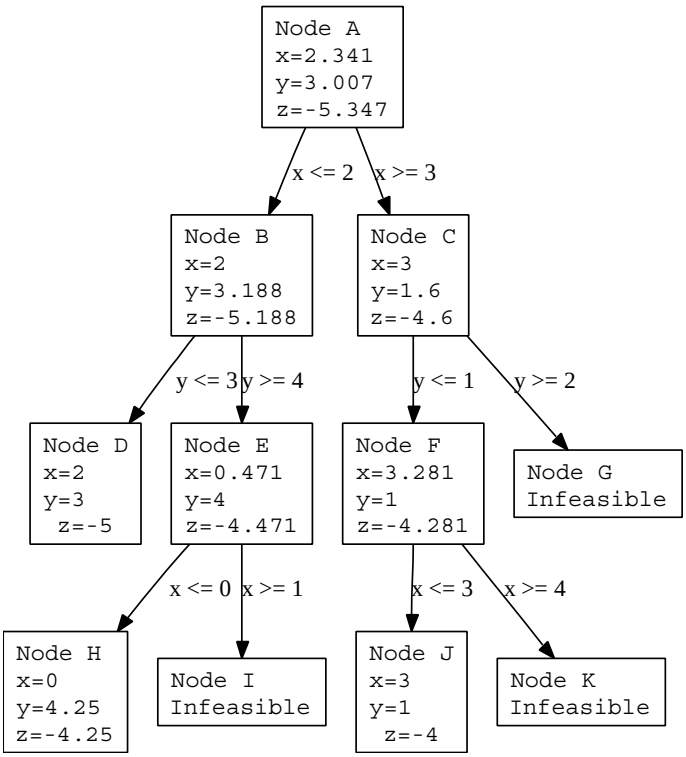


图 13.6: 分支定界法

首先将问题转换为最小化问题

$$\min\{z = -x - y : (x, y) \in S\}$$
$$S = \{(x, y)|17x + 32y \leq 136, 32x + 15y \leq 120, x \in \mathbb{Z}_+, y \in \mathbb{Z}_+\}$$

然后定义问题的松弛

$$\min\{z = -x - y : (x, y) \in S\}$$
$$S = \{(x, y)|17x + 32y \leq 136, 32x + 15y \leq 120, x \in \mathbb{R}_+, y \in \mathbb{R}_+\}$$

节点 A 显示了松弛的最优解。节点 B 和节点 C 都是节点 A 的子节点，从节点 A 到节点 B 的有向边上标注了 $x \leq 2$，表示节点 B 显示的是向节点 B 对应的问题 (即松弛) 添加约束条件 $x \leq 2$ 得到的新问题

$$\min\{z = -x - y : (x, y) \in S\}$$
$$S = \{(x, y) | 17x + 32y \leq 136, 32x + 15y \leq 120, x \in \mathbb{R}_+, y \in \mathbb{R}_+, x \leq 2\}$$

的最优解。
从节点 B 到节点 D 的有向边上标注了 $y \leq 3$，表示节点 D 显示的是向节点 B 对应的问题添加约束条件 $y \leq 3$ 得到的新问题

$$\min\{z = -x - y : (x, y) \in S\}$$
$$S = \{(x, y) | 17x + 32y \leq 136, 32x + 15y \leq 120, x \in \mathbb{R}_+, y \in \mathbb{R}_+, x \leq 2, y \leq 3\}$$

的最优解。其他节点的含义依此类推。

分支定界法的求解过程就是生成和遍历这些节点的过程。这一过程并非唯一确定的，因为在中间步骤可以有多种选择，例如选择哪个变量进行分支，从哪个变量的哪个分支生成子节点等。图 13.6 显示的是从从节点 A 选择变量 x 进行分支的情形，如果选择变量 y 进行分支会得到其他过程。不同的求解过程的计算量存在差别，目前并没有适用于所有混合整数线性规划问题的最佳选择策略 [RS2017]。这里列举两种过程。

1. 从节点 A 生成节点 B，再从节点 B 生成节点 D。由于节点 D 表示整数解，不再分支，给 z^b 赋初值为-5。从节点 D 上溯到其父节点 (节点 B)，再从节点 B 生成节点 E。由于节点 E 的 z 值-4.471 大于 $z^b = -5$，无需分支。从节点 E 上溯到其父节点 (节点 B)，再从节点 B 上溯到其父节点 (节点 A)，再从节点 A 生成节点 C。由于节点 C 的 z 值-4.6 大于 $z^b = -5$，无需分支。求解过程到此结束，最优解是节点 D。需要求解五个线性最优化问题。

2. 从节点 A 生成节点 C。访问节点 C，从节点 C 生成节点 F。访问节点 F，从节点 F 生成节点 J。由于节点 J 表示整数解，不再分支，给 z^b 赋初值为-4。从节点 J 上溯到其父节点 (节点 F)，从节点 F 生成节点 K。节点 K 无解，无需分支。从节点 K 上溯到其父节点 (节点 F)，再从节点 F 上溯到其父节点 (节点 C)。从节点 C 生成节点 G。节点 G 无解，无需分支。从节点 G 上溯到其父节点 (节点 C)，再从节点 C 上溯到其父节点 (节点 A)。从节点 A 生成节点 B，再从节点 B 生成节点 D。由于节点 D 表示整数解，不再分支。由于节点 D 的 z 值-5 小于当前 $z^b = -4$，将 z^b 更改为-5。从节点 D 上溯到其父节点 (节点 B)，再从节点 B 生成节点 E。由于节点 E 的 z 值-4.471 大于 $z^b = -5$，无需分支。求解过程到此结束，最优解是节点 D。需要求解九个线性最优化问题。

13.4.2 Pyomo

Pyomo(Python Optimization Modeling Objects, http://www.pyomo.org/) 是一个 Python 开源软件包，支持以面向对象的建模语言 (例如 AMPL) 表示多种类型的最优化问题，包括线性

规划、非线性规划和整数规划等。Pyomo 借助于最优化求解器求解这些问题。glpk 是一个开源的单线程线性规划和整数规划求解器，适用于中小规模的问题。COIN-OR CBC 是一个多线程混合整数线性规划求解器，适用于中等以上规模的问题。COIN-OR Ipopt 是一个开源的基于内点法的无整数约束的非线性规划求解器，适用于大规模的问题。COIN-OR Bonmin 是一个开源的非线性整数规划求解器。安装 Pyomo 和 glpk 的命令如下

Listing 13.10: **pyomo**

```
pip install pyomo
conda install -c conda-forge glpk
```

程序 13.11 演示了使用 Pyomo 求解以下问题

$$\max\{z = x + y : (x, y) \in S\}$$
$$S = \{(x, y)|17x + 32y \leq 136, 32x + 15y \leq 120, x \in \mathbb{Z}_+, y \in \mathbb{Z}_+\}$$

如果将第 4 行和第 5 行的“NonNegativeIntegers”改为 NonNegativeReals，则得到该问题的松弛

$$\max\{z = x + y : (x, y) \in S\}$$
$$S = \{(x, y)|17x + 32y \leq 136, 32x + 15y \leq 120, x \in \mathbb{Z}_+, y \in \mathbb{R}_+\}$$

的解“Objective = -9.10 (x, y) = (6.35, 2.75)”。上一节的整数问题的最优解 $(2, 3)$ 可以从松弛的最优解 $(2.341, 3.007)$ 取整得到。但是这个整数问题的最优解 $(5, 3)$ 不等于松弛的最优解 $(6.35, 2.75)$ 的取整结果。

Listing 13.11: **pyomo**

```
1  from pyomo.environ import *
2
3  model = ConcreteModel()
4  model.x = Var(domain=NonNegativeIntegers)
5  model.y = Var(domain=NonNegativeIntegers)
6  model.objective = Objective(expr = -model.x - model.y,
7                              sense=minimize)
8  model.constraint1 = Constraint(
9      expr = 17 * model.x + 32 * model.y <= 196
10 )
11 model.constraint2 = Constraint(
12     expr = 5 * model.x + 3 * model.y <= 40
13 )
14 SolverFactory('glpk').solve(model).write()
15 model.pprint()
16 print('\nObjective␣=␣%8.2f␣(x,␣y)␣=␣(%6.2f,␣%6.2f)' %\
```

```
        (model.objective(), model.x(), model.y()))
# Objective =     -8.00 (x, y) = (   5.00,    3.00)
```

参考文献

[CC2014] Michele Conforti, Gérard Cornuéjols and Giacomo Zambelli, Integer programming, Springer, 2014.

[CG2016] T. Chen and C. Guestrin.“XGBoost: A Scalable Tree Boosting System”. In: KDD. ACM, 2016, pp. 785 – 794.

[CL2009] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, Introduction to Algorithms, MIT Press, 2009.

[CythonDoc] Cython Documentation, https://cython.readthedocs.io/en/stable/index.html

[HL2020] 汉斯. 佩特. 兰坦根 (挪) 著，张春元、刘万伟、毛晓光、陈立前、周会平、李暾译，科学计算基础编程——Python 版, 清华大学出版社, 2020.

[IJ2002] I. T. Jolliffe, Pricipal Component Analysis, Springer, 2002.

[JC2021] Jean-Pierre Corriou, Numerical Methods and Optimization, Springer, 2012.

[JD2011] Jay L. Devore and Kenneth N. Berk, Modern Mathematical Statistics with Applications, Springer, 2011.

[JN2006] Jorge Nocedal and Stephen J. Wright, Numerical Optimization, Springer, 2006.

[KM2022] Kevin P. Murphy, Probabilistic machine learning : an introduction, The MIT Press, 2022.

[KS2021] Qingkai Kong, Timmy Siauw and Alexandre M. Bayen, Python Programming And Numerical Methods: A Guide For Engineers And Scientists, Elsevier, 2021.

[MG2017] Micha Gorelick and Ian Ozsvald(美) 著 ，胡世杰、徐旭彬译，Python 高性能编程, 人民邮电出版社, 2017.

[KT2006] Jon Kleinberg and Eva Tardos, Algorithm Design, Pearson Education, Inc., 2006

[MH2010] Magnus Lie Hetland, Python Algorithms: Mastering Basic Algorithms in the Python Language, Springer, 2010.

[MH2017] Magnus Lie Hetland, Beginning Python, Springer, 2017.

[NL2017] Nicolas Lanchier, Stochastic Modeling, Springer, 2017.

[NumPyDoc] NumPy Reference, https://numpy.org/doc/stable

[PE2011] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

[PythonDoc] Python Documentation, https://docs.python.org/

[RJ2019] Robert Johansson, Numerical Python, Springer, 2019.

[RS2017] Ramteen Sioshansi and Antonio J. Conejo, Optimization in Engineering: Models and Algorithms, Springer, 2017.

[RV2014] Robert J. Vanderbei, Linear Programming, Springer, 2014.

[SB2002] J. Stoer and R. Bulirsch, Introduction to Numerical Analysis, Springer, 2002.

[Scikit-learnDoc] Scikit-learn Documentation, https://scikit-learn.org/stable/user_guide.html

[SciPyDoc] SciPy Reference Guide, https://scipy.github.io/devdocs/index.html

[SL2018] Stephen Lynch, Dynamical Systems with Applications using Python, Springer, 2018.

[SS2010] Seabold, Skipper, and Josef Perktold, statsmodels: Econometric and statistical modeling with python, Proceedings of the 9th Python in Science Conference, 2010.

[StatsmodelsDoc] statsmodels Documentation, https://www.statsmodels.org/stable/index.html

[SW2011] Robert Sedgewick and Kevin Wayne, Algorithms, Pearson Education, Inc., 2011.

[SymPyDoc] SymPy Documentation, https://docs.sympy.org/latest/index.html

[TL2019] Tom Lyche, Numerical Linear Algebra and Matrix Factorizations, Springer, 2019.

[TM1997] T. Mitchell. Machine Learning. McGraw Hill, 1997.

[WG2012] Walter Gautschi, Numerical Analysis, Springer, 2012.