Python科学计算基础

罗奇鸣 编著

2023年3月31日

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。

目录

第一章　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。

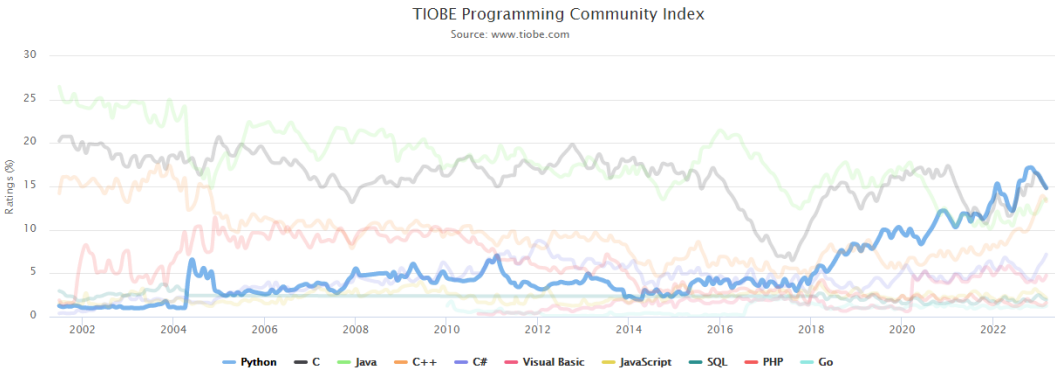

图 1.1: TIOBE指数2023年3月

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？

- 查阅书籍和技术文档；
- 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。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. 预约时间在前三次课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。

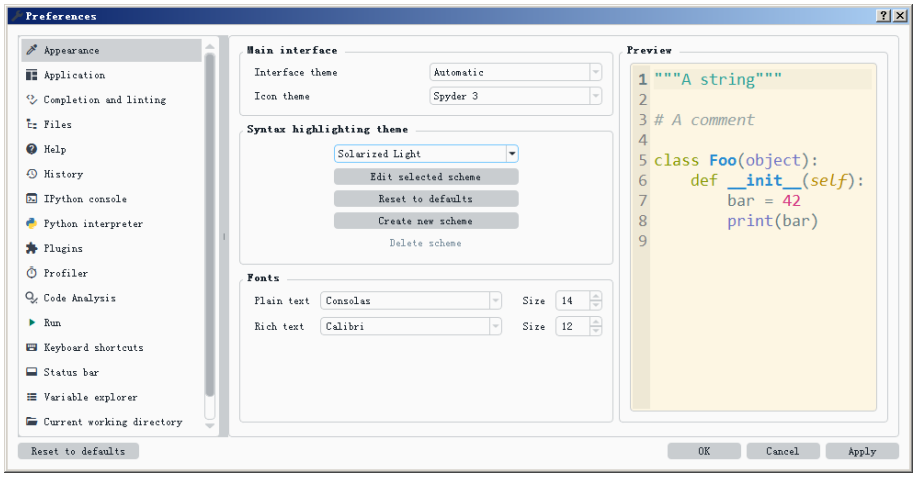

图 1.3: 设置spyder参数

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

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。

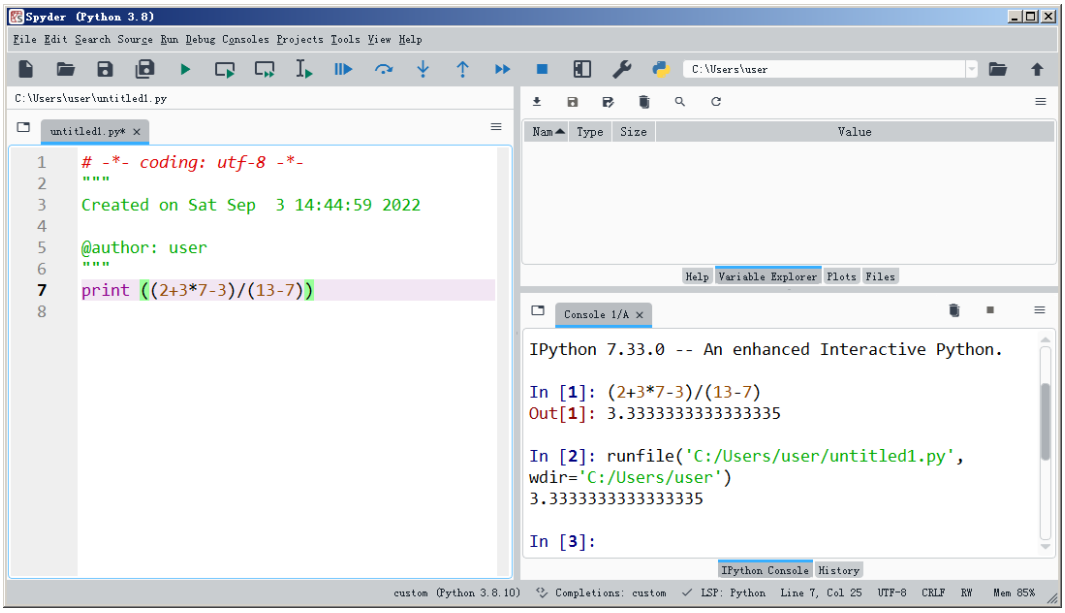

图 1.4: spyder界面

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
```

第二章　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=379516400906811930638014896080
In [5]: x**2
Out [5]: 144032698557259999607886110560755362973171476419973199366400
In [6]: y=120557357903313594474425387 67
In [7]: 991*y**2
Out [7]: 144032698557259999607886110560755362973171476419973199366399
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可以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。$e=2047, f=0, s=\pm 1$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。

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

2.7演示了求解一元二次方程$ax^2 + bx + c = 0$ $(a \neq 0)$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中存在等于x的元素，则返回False，否则返回True。 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，则返回元素x第一次出现的索引值，否则报错。 例: s.index(7)值为3
s.index(x, i)	在s中从索引值i开始向后查找，若存在元素x则返回其索引值，否则报错。 例: s.index(7,1)值为3
s.index(x, i, j)	在s中从索引值i开始向后查找到索引值j-1为止， 若存在元素x则返回其索引值，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以后s值为[2,3,5,8,11]
s[i:j] = t	将s中索引值从i到j-1的所有元素构成的序列修改为t。 例: 运行s[1:4] = t以后s值为[2, 13, 21, 17, 11]
del s[i:j]	删除s中索引值从i到j-1的所有元素构成的序列。 例: 运行del s[2:4]以后s值为[2, 7, 11]
s[i:j:k] = t	将s中索引值属于从i到j-1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以后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-1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]以后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。 例: 运行s.append(13)以后s值为[2, 7, 5, 3, 11, 13]
s.clear()	删除s中所有元素。例: 运行s.clear()以后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。 例: 运行s.extend(t)以后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以后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)以后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)以后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()以后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()以后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)以后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]
```

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：不返回字符串的运算和返回字符串的运算。由于str类型是不可变的，第二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中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[, start[, end]])	若s(或s[start:]或[start:end])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, 1, 4)的值为False。
s.find(t[, start[, end]])	若s(或s[start:]或[start:end])中存在子串t，则返回t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, 2)的值为7，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 2.5: str类型的不返回字符串的部分特有运算

表2.6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有子串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)的区别是仅对s的前k个子串t运行。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。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。例: t.join(['a', '123a', ''])的值为s。

表 2.6: str类型的返回字符串的部分特有运算

2.10演示了str类型的一些运算。

Listing 2.10: **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
```

```
In[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']
```

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进行格式化输出。%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。

2.11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 2.11: **格式化输出**

```
In[1]: "%-16.8f" % 345.678987654321012
Out[1]: '345.67898765␣␣␣␣'
In[2]: "%16.8g" % 3.456789876543210 12e34
Out[2]: '␣␣␣3.4567899e+34'
In[3]: "%16X␣%-16d" % (345678987654, 987654321012)
Out[3]: '␣␣␣␣␣␣507C12C186␣987654321012␣␣␣␣'
```

表2.7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 2.7: 常用的转换说明符

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2.8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 2.8: 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}
```

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)。 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质数，其映射到的值是键的平方。

表2.9列出了dict类型的常用运算。

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 2.9: 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]
```

```
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})
```

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。

第三章　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执行，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执行一个语句块，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执行。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时执行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时执行if后面的分支，值为False时执行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
4 else:
```

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

Listing 3.3: **if-else表达式计算绝对值**

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

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)，即可以在for循环中遍历其包含的所有元素，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**

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

```
lb = 100; ub = 120
for i in range(lb, ub+1, 2):
    print(i, end='␣')
# 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**

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

程序3.8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 3.8: **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3.9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 3.9: **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，提高程序的执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3.10。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 3.10: **for语句和break语句输出100到200之间的所有质数**

```
import math
lb = 100; ub = 200
if lb % 2 == 0: lb += 1
if ub % 2 == 0: ub -= 1
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
```

程序3.11实现了和3.10相同的功能。区别在于第11行如果确认isPrime的值为False，则使用continue语句跳过本次外循环的剩余语句并开始下一次外循环，否则在第12行输出i。

Listing 3.11: **for语句和continue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 not isPrime: continue
12     print(i, end='␣')
```

3.3 while语句

while语句包含一个条件表达式和一个语句块。while语句的执行过程如下：

1. 对条件表达式求值。
2. 若值为False，则while语句执行结束。
3. 若值为True，则执行语句块，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**

```
1  sum = 0; n = 10; i = 1
2  while i <= n:
3      sum += i
```

```
4     i += 1
5 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**

```
1 i = lb = 100; ub = 120
2 while i <= ub:
3     print(i, end='␣')
4     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**

```
1 a = 156; b = 732
2 str = 'The␣greatest␣common␣divisor␣of␣%d␣and␣%d␣is␣' % (a, b)
3 while a != b:
4     if a > b:
5         a -= b;
6     else:
7         b -= a;
8 print(str + ('%d' % a))
9 # 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}
```

```
print(square_of_odds)    # {133225, 3481, 625, 8281}

s = [25, 18, 91, 365, 12, 78, 59, 18, 91]
sr = {n:n%3 for n in set(s)}
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。

第四章　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1.1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;
8     return a
9
```

```
10 print(gcd(156, 732))  # 12
11 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1.2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1.3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1.4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

```
b = 10
c = 15
def f(a):
    global b
    b = 20
    return a + b + c
print(f(5))  # 40
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1.5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

```
import math
def get_primes(lb, ub=100):
    """ 求解给定取值范围[lb, ub]内的所有质数 """
    primes = []
    if lb % 2 == 0: lb += 1
    if ub % 2 == 0: ub -= 1
    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1.6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

```
def fun(*args, **kwargs):
    print(type(args), type(kwargs))
    print('The positional arguments are', args)
    print('The keyword arguments are', kwargs)

fun(1, 2.3, 'a', True, u=6, x='Python', f=3.1415)
# <class 'tuple'> <class 'dict'>
# The positional arguments are (1, 2.3, 'a', True)
# 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1.7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**

```
In[1]: animals = ["elephant", "tiger", "rabbit", "goat", "dog",
                  "penguin"]
In[2]: sorted(animals)
Out[2]: ['dog', 'elephant', 'goat', 'penguin', 'rabbit', 'tiger']
In[3]: sorted(animals, key=len)
Out[3]: ['dog', 'goat', 'tiger', 'rabbit', 'penguin', 'elephant']
In[4]: sorted(animals, key=len, reverse=True)
Out[4]: ['elephant', 'penguin', 'rabbit', 'tiger', 'goat', 'dog']
In[5]: def m1(s): return ord(min(s))
In[6]: sorted(animals, key=m1)
Out[6]: ['elephant', 'rabbit', 'goat', 'dog', 'tiger', 'penguin']
In[7]: def m2(s): return ord(min(s)), len(s)
In[8]: sorted(animals, key=m2)
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1.8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**

```
def map_fs(f, s):
    for i in range(len(s)): s[i] = f(s[i])
    return s

a = [1, 3, 5, 7, 9]
print(map_fs(lambda x: x+1, a))  # [2, 4, 6, 8, 10]
print(map_fs(lambda x: x*x-1, a))  # [3, 15, 35, 63, 99]
```

程序1.8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1.9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1.7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**

```
In[1]: animals = ["elephant", "tiger", "rabbit", "goat", "dog",
                  "penguin"]
In[2]: def reverse(s): return s[::-1]
In[3]: list(map(reverse, animals))
Out[3]: ['tnahpele', 'regit', 'tibbar', 'taog', 'god', 'niugnep']
In[4]: def m2(s): return ord(min(s)), len(s)
In[5]: list(map(m2, animals))
Out[5]: [(97, 8), (101, 5), (97, 6), (97, 4), (100, 3), (101, 7)]
In[6]: def f(a, b, c): return a + b + c
In[7]: list(map(f, [1, 2, 3], [10, 20, 30], [100, 200, 300]))
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1.10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1.11。

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",
9             "penguin"]
10 for i in range(4):
11     print(sorted(animals, key=key_fun(i)))
```

Listing 4.11: 程序1.10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1.12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1.13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1.1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)

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1.14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1.15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

```
1 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1.16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_iteration1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_iteration2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**

```
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_iteration1(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_iteration2(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_iteration1(n), F_iteration2(n))
# 6765 6765 6765 6765
```

4.6.6 0-1背包问题

给定n件物品，每件物品都有其重量w_i和价值$v_i(0 \leq i \leq n-1)$。0-1背包问题从这n件物品中选择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。

用整数$0, 1, ..., n-1$给所有物品编号。用$\text{ks}(k, c)$表示当背包的当前容量为c并且选取范围是前k件物品时所能得到的最大总价值。对于前k件物品做出的选择可分为两部分：首先对第k件物品(编号为$k-1$)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，因此可以分阶段进行。以下列出了设计方案。

1. 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。在第二阶段的问题是当背包的当前容量为c时对前$k-1$件物品时进行选择并获得最大总价值，即求解子问题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，需要从两个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}，在第二阶段的问题是当背包的当前容量为$c-w_{k-1}$时对前$k-1$件物品时进行选择并获得最大总价值，即求解子问题ks$(k-1,c-w_{k-1})$。

      ii. 若不选取第k件物品，在第二阶段的问题是当背包的当前容量为c时对前$k-1$件物品时进行选择并获得最大总价值，即求解子问题ks$(k-1,c)$。

由此可总结出递归公式如下：

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

程序1.17的前14行定义的函数knapsack使用递归方法求解0-1背包问题，它的第一个形参weights和第二个形参values分别是存储了每件物品的重量和价值的列表，第三个形参c0是背包的容量。knapsack的函数体内定义了一个递归函数ks，它实现了上述递归公式并由第12行调用。函数ks中使用了在第3行初始化的嵌套列表selects来记录在子问题ks(k,c)中是否选取了第k件物品。第12行调用了第16行至第22行定义的函数get_items获取最优解所选取的所有物品的编号。

用二维表格ks记录ks(k,c)，从递归公式可以推导出以下迭代公式：

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

第24行至第41行定义的函数knapsack_iteration根据迭代公式使用双重循环求解0-1背包问题。第26行初始化的嵌套列表ks表示公式中的二维表格ks。第27行初始化的嵌套列表selects记录了在子问题ks(k,c)中是否选取了第k件物品。第40行使用pprint模块的pprint函数输出了ks的计算结果，显示在第47行至第51行。第44行和第45行分别使用第43行定义的实参调用knapsack和knapsack_iteration，它们的输出结果是：最优解选取了编号为0、1和3的物品，它们的总价值是37。

Listing 4.17: **求解0-1背包问题**

```
1 def knapsack(weights, values, c0):
2     n = len(weights)
3     selects = [[False]*(c0+1) for i in range(n+1)]
```

```
4       def ks(k, c):
5           if k == 0 or c == 0: return 0
6           exclude_k = ks(k-1, c)
7           if weights[k-1] > c: return exclude_k
8           select_k = values[k-1] + ks(k - 1, c - weights[k-1])
9           if exclude_k > select_k:
10              return exclude_k
11          else:
12              selects[k][c] = True
13              return select_k
14      return ks(n, c0), get_items(n, c0, weights, selects)
15
16  def get_items(k, c, weights, selects):
17      items = set()
18      while k > 0 and c > 0:
19          if selects[k][c]:
20              items.add(k-1); c -= weights[k-1]
21          k -= 1
22      return items
23
24  def knapsack_iteration(weights, values, c0):
25      n = len(weights)
26      ks = [[0]*(c0+1) for i in range(n+1)]
27      selects = [[False]*(c0+1) for i in range(n+1)]
28      for k in range(1, n+1):
29          for c in range(1, c0+1):
30              exclude_k = ks[k-1][c]
31              if weights[k-1] > c:
32                  ks[k][c] = ks[k-1][c]
33                  continue
34              select_k = values[k-1] + ks[k - 1][c - weights[k-1]]
35              if exclude_k > select_k:
36                  ks[k][c] = exclude_k
37              else:
38                  selects[k][c] = True
39                  ks[k][c] = select_k
40      import pprint; pprint.pprint(ks)
41      return ks[n][c0], get_items(n, c0, weights, selects)
42
```

```
weights = [2,1,3,2]; values = [12,10,20,15]; c0=5
print(knapsack(weights, values, c0))  # (37, {0, 1, 3})
print(knapsack_iteration(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Cost(i, j)是以下三部分之和：第一个子问题的计算代价Cost(i, k)；第二个子问题的计算代价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，Cost(i,j)也依赖于k的值。应选取最小化Cost(i,j)的k值。用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：

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

程序1.18定义的函数matrix_chain实现了矩阵链乘积问题的递归求解，它的形参r是存储了公式中的r_i值$(1\le i\le n+1)$的列表。由于矩阵的下标从1开始，第2行将矩阵的个数设置为r的长度减2。第3行创建的嵌套列表mcm_value存储公式中的mcm$(i,j)(1\le i\le j\le n)$，列表中所有元素的初始值是0。第4行创建的嵌套列表k_value存储公式中最小化Cost(i,j)的k值。matrix_chain的函数体内定义了一个递归函数mcm，它实现了上述递归公式，并由第19行调用。

变量min_cost记录Cost(i,j)的最小值，它的初始值在第8行设置为$k=j-1$时的Cost(i,j)，然后在第10行至第15行的循环中求得最小值。变量k_min_cost记录使得Cost(i,j)取最小值的k值，它的初始值在第8行设置为$k=j-1$，然后在第10行至第15行的循环中求得其最终值。

为了避免重复计算，第15行将函数mcm(i,j)的返回值存储在mcm_value中。第7行先判断mcm_value[i][j]是否大于0。若条件成立，说明mcm(i,j)已经被调用过，可以直接返回mcm_value[i][j]而不必重复计算。第21行至第26行定义的递归函数get_mcm_str根据k_value生成最小化Cost(i,j)的加了括号的矩阵计算序列。

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：

$$\text{mcm\_value}[i,j]=\begin{cases}0 & \text{if } i=j\\ \arg\min_{i\le k\le j-1}\text{mcm\_value}[i,k]+\text{mcm\_value}[k+1,j]+r_ir_{k+1}r_{j+1} & \text{if } i<j\end{cases}$$

第28行至第47行定义的函数matrix_chain_iteration根据迭代公式使用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。第34行开始的外层循环的循环变量l表示矩阵链长度，第35行开始的内层循环的循环变量i表示矩阵链的起始下标i。第36行根据i和l计算矩阵链的终止下标j。第37行至第46行使用迭代公式计算mcm_value和k_value。

第49行定义的列表rs存储了由6个矩阵组成的矩阵链的r_i值$(1\le i\le 7)$。第50行和第51行分别以rs为实参调用matrix_chain和matrix_chain_iteration，它们的输出结果相同：((A1(A2A3))((A4A5)A6))实现了最小计算代价15125。

Listing 4.18: **求解矩阵链乘积问题**

```
def matrix_chain(r):
    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)]
    def mcm(i, j):
        if i == j: return 0
        if mcm_value[i][j] > 0: return mcm_value[i][j]
        min_cost = mcm(i, j-1) + r[i] * r[j] * r[j+1]
        k_min_cost = j-1
        for k in range(i, j-1):
            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
        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):
    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_iteration(r):
    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):
        for i in range(1, n-l+2):
            j = i + l - 1
            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]
print(matrix_chain(rs))  # (15125, '((A1(A2A3))((A4A5)A6))')
print(matrix_chain_iteration(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1.19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.

For example, given year 2022 and month 9, the module prints
the monthly calendar of September 2022.

>>> run month_calendar.py --year 2022 --month 9
2022  9
----------------------------
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 # 若year是闰年返回True，否则返回False
23 def is_leap(year):
24     return (year % 4 == 0 and year % 100 != 0) or year % 400 == 0
25
26 # 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
27 def test____is_leap():
28 # 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。
29     d = {1900:False, 2000:True, 2020:True, 2022:False}
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 # 计算year年的元旦是星期几。返回0表示星期日，返回1至6分别表示星期一至星期六。
35 def get_0101_in_week(year):
36     return (year + math.floor((year - 1) / 4) -
37             math.floor((year - 1) / 100) +
38             math.floor((year - 1) / 400)) % 7
39
40 # 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。
41 def test____get_0101_in_week():
42 # 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。
43     d = {2008:2, 2014:3, 2021:5, 2022:6}
44     for y in d.keys():
45         if d[y] != get_0101_in_week(y):
46             print("test␣failed:␣get_0101_in_week(%d)␣!=␣%s"
47                   % (y, d[y]))
48
49 month_days = {1:31, 2:28, 3:31, 4:30, 5:31, 6:30,
50               7:31, 8:31, 9:30, 10:31, 11:30, 12:31}
```

```
51 # 返回year年month月所包含的天数，需要对闰年的二月单独处理。
52 def get_num_days_in_month(year, month):
53     n = month_days[month]
54     if month == 2 and is_leap(year):
55         return n + 1
56     return n
57
58 # 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。
59 def get_num_days_from_0101_to_m01(year, month):
60     n = 0
61     for i in range(1, month):
62         n += get_num_days_in_month(year, i)
63     return n
64
65 # 计算year年的month月的第一天是星期几。
66 def get_m01_in_week(year, month):
67     n1 = get_0101_in_week(year)
68     n2 = get_num_days_from_0101_to_m01(year, month)
69     n = (n1 + n2) % 7
70     return n
71
72 # 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。
73 def test____get_m01_in_week():
74 # 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，
75 # 对应的值表示该年该月的第一天是星期几。
76     d = {(2022, 6):3, (2019, 10):2, (2016, 5):0, (2011, 7):5}
77     for y in d.keys():
78         if d[y] != get_m01_in_week(y[0], y[1]):
79             print("test␣failed:␣get_m01_in_week(%s)␣!=␣%s"
80                   % (y, d[y]))
81
82 # 输出year年month月的日历的标题。
83 def print_header(year, month):
84     print("%d␣␣%d␣" % (year, month))
85     print("----------------------------")
86     print("Sun␣Mon␣Tue␣Wed␣Thu␣Fri␣Sat")
87
88 # 输出year年month月的日历的主体。
89 def print_body(year, month):
```

```
    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()

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

# 调用所有以“test___”为名称前缀的测试函数。
def test_all_functions():
    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，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:
        import argparse  # argparse模块获取用户在命令行输入的年份和月份。
        parser = argparse.ArgumentParser()
# 在每个参数(年份或月份)的输入值前面都要用"--参数名称"的格式指定对应的参数名称，
# 参数的顺序无关紧要。
        parser.add_argument('--year', type=int, default=2022)
        parser.add_argument('--month', type=int, default=1)
        args = parser.parse_args()
        year = args.year; month = args.month
        print_monthly_calendar(year, month) # 输出日历。
```

1.20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**

```
In[1]: cd D:\Python\src
Out[1]: D:\Python\src
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
```

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1.21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**

```
import month_calendar
y, m, d = 2022, 9, 18
n = (month_calendar.get_m01_in_week(y, m) + d - 1) % 7
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1.19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1.19中添加以下两个函数，然后将函数print_body的第一条语句(第90行)改为 n = get_m01_in_week_precomputed(year, month)。

Listing 4.23: **避免多次运行模块1.19时的重复计算**

```
1  def get_num_days_from_0101_to_m01_remainder():
2      n = 0; r = [0]
3      for i in range(1, 12):
4          n += get_num_days_in_month(2022, i)
5          r.append(n % 7)
6      print(r) # [0, 3, 3, 6, 1, 4, 6, 2, 5, 0, 3, 5]
7
8  def get_m01_in_week_precomputed(year, month):
9      r = [0, 3, 3, 6, 1, 4, 6, 2, 5, 0, 3, 5]
10     n1 = get_0101_in_week(year)
11     n2 = r[month-1];
12     if (is_leap(year)) and month > 2:
13         n2 += 1
14     n = (n1 + n2) % 7
15     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1.26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。 1.25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**

```
def is_sorted(s):  # 判断列表s是否已经排好序: to be implemented

def qsort(s):  # 对列表s排序
    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)

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1.26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。1.27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
```

```
44
45 def mul(x, y):  # 乘法: to be implemented
46
47 def div(x, y):  # 除法: to be implemented
48
49 def output(x):  # 按照示例的格式输出有理数: to be implemented
50
51 def get_rational(s):  # 从字符串s中得到列表[n, d]: to be implemented
52
53 if __name__ == '__main__':
54     if len(sys.argv) == 1:
55         print(__doc__)
56     elif len(sys.argv) == 2 and sys.argv[1] == '-h':
57         print(__doc__)
58     elif len(sys.argv) == 2 and sys.argv[1] == 'test':
59         test_all_functions()
60     else:
61         import argparse
62         parser = argparse.ArgumentParser()
63         parser.add_argument('--op', type=str)
64         parser.add_argument('--x', type=str)
65         parser.add_argument('--y', type=str)
66         args = parser.parse_args()
67         op = args.op
68         x = get_rational(args.x); y = get_rational(args.y)
69         f = {'add':add, 'sub':sub, 'mul':mul, 'div':div}
70         output(f[op](x, y))
```

第五章　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26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5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:
3 # __init__称为构造方法，用来初始化新创建的对象的所有属性。
```

```
4       def __init__(self, x, y, name=''):
5           self.x = x  # 用形参x初始化表示点的x坐标的self.x属性。
6           self.y = y  # 用形参x初始化表示点的y坐标的self.y属性。
7           self.name = name  # 用形参name初始化表示点的名称的name属性。
8
9   # 计算self沿x轴平移一段距离delta_x以后的x坐标。
10      def move_x(self, delta_x): self.x += delta_x
11
12  # 计算self沿y轴平移一段距离delta_y以后的x坐标。
13      def move_y(self, delta_y): self.y += delta_y
14
15  # 计算self以另一个点p为轴旋转角度t以后的坐标。
16      def rotate(self, p, t):
17          xr = self.x - p.x; yr = self.y - p.y
18          x1 = p.x + xr * math.cos(t) - yr * math.sin(t)
19          y1 = p.y + xr * math.sin(t) + yr * math.cos(t)
20          self.x = x1; self.y = y1;
21
22  # 计算self与另一个点p之间的距离。
23      def distance(self, p):
24          xr = self.x - p.x; yr = self.y - p.y
25          return math.sqrt(xr * xr + yr * yr)
26
27  # 返回self的字符串表示。
28      def __str__(self):
29          if len(self.name) < 1:
30              return '(%g, %g)' % (self.x, self.y)
31          else:
32              return '%s: (%g, %g)' % (self.name, self.x, self.y)
33
34  # 创建了一个点a，然后输出其字符串表示。
35  a = Point2D(-5, 2, 'a'); print(a)  # a: (-5, 2)
36  # 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。
37  a.move_x(-1); print(a)             # a: (-6, 2)
38  # 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。
39  a.move_y(2); print(a)              # a: (-6, 4)
40  # 创建了一个点b，然后输出其字符串表示。
41  b = Point2D(3, 4, 'b');  print(b)  # b: (3, 4)
42  # 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。
```

```
print('The distance between a and b is %f' % a.distance(b))
# The distance between a and b is 9.000000
# 调用点a的rotate方法将点b以点a为中心旋转90度。
b.rotate(a, math.pi/2)
print(a); print(b)      # a: (-6, 4)   b: (-6, 13)
# 调用点a的rotate方法将点a以点b为中心旋转180度。
a.rotate(b, math.pi)
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。

Listing 5.2: **Complex类**

```
import math

class Complex:
    def __init__(self, re=0, im=0):
# 判断形参re和im的类型是不是float或int，若不是则以抛出异常的方式报错(第七章)。
        if not isinstance(re, (float,int)) or \
           not isinstance(im, (float,int)):
            raise TypeError('Error: float or int expected')
# 将属性名're'和'im'分别映射到对应的值re和im。
        self.__dict__['re'] = re; self.__dict__['im'] = im

# 当self的任何属性的值被试图修改时，该方法会被自动调用。
# 方法以抛出异常的方式禁止任何属性的值被修改。
    def __setattr__(self, name, value):
        raise TypeError('Error: Complex objects are immutable')

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

# 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。
    def __repr__(self): return 'Complex' + str(self)

```

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

```
23 # 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)
24 # 被转换为与其等价的方法调用c.__abs__()。
25     def __abs__(self): return math.hypot(self.re, self.im)
26
27 # 方法abs是方法__abs__的别名。例如方法调用self.abs()
28 # 被转换为与其等价的方法调用self.__abs__()。
29     abs = __abs__
30
31 # 返回self的幅角。
32     def angle(self): return math.atan2(self.im, self.re)
33
34 # 对于一个Complex类的对象c，计算c与other的和的简化语法c+other
35 # 被转换为与其等价的方法调用c.__add__(other)。
36     def __add__(self, other):
37         other = to_Complex(other)
38         return Complex(self.re + other.re, self.im + other.im)
39
40 # 对于一个Complex类的对象c，计算other与c的和的简化语法other+c
41 # 被转换为与其等价的方法调用c.__radd__(other)。
42 # 加法运算是可交换的，c.__radd__(other)被转换为c.__add__(other)。
43     __radd__ = __add__
44
45 # 对于一个Complex类的对象c，计算c与other的差的简化语法c-other
46 # 被转换为与其等价的方法调用c.__sub__(other)。
47     def __sub__(self, other):
48         other = to_Complex(other)
49         return Complex(self.re - other.re, self.im - other.im)
50
51 # 对于一个Complex类的对象c，计算other与c的差的简化语法other-c
52 # 被转换为与其等价的方法调用c.__rsub__(other)。
53     def __rsub__(self, other):
54         other = to_Complex(other)  # 将对象other转换成Complex类的对象
55         return other - self  # 被转换为方法调用other.__sub__(self)。
56
57 # 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
58 # 被转换为与其等价的方法调用c.__mul__(other)。
59     def __mul__(self, other):
60         other = to_Complex(other)
61         return Complex(self.re*other.re - self.im*other.im,
```

```
62                          self.re*other.im + self.im*other.re)
63
64  # 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
65  # 被转换为与其等价的方法调用c.__rmul__(other)。
66  # 乘法运算是可交换的，c.__rmul__(other)被转换为c.__mul__(other)。
67      __rmul__ = __mul__
68
69  # 对于一个Complex类的对象c，计算c与other的商的简化语法c/other
70  # 被转换为与其等价的方法调用c.__truediv__(other)。
71      def __truediv__(self, other):
72          other = to_Complex(other)
73          d = float(other.re*other.re + other.im*other.im)
74          if is_zero(d): # 若除数的模为0，则以抛出异常的方式报错。
75              raise ZeroDivisionError('Error: division by 0')
76          return Complex((self.re*other.re + self.im*other.im) / d,
77                         (self.im*other.re - self.re*other.im) / d)
78
79  # 对于一个Complex类的对象c，计算other与c的商的简化语法other/c
80  # 被转换为与其等价的方法调用c.__rtruediv__(other)。
81      def __rtruediv__(self, other):
82          other = to_Complex(other)  # 将对象other转换成Complex类的对象
83          return other / self  # 被转换为方法调用other.__truediv__(self)。
84
85  # 计算以self为底数和n为指数的乘方运算。
86      def __pow__(self, n):
87          if is_Complex(n):  # 判断n的类型是不是Complex
88              if not is_zero(n.im):  # 判断n的虚部是不是0
89                  if is_zero(self.im):  # 判断self的虚部是不是0
90  # 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)。
91                      z = n * math.log(self.re)
92                      r = math.exp(z.re)
93                      return Complex(r * math.cos(z.im),
94                                     r * math.sin(z.im))
95                  else: # 抛出异常提示尚未实现底数和指数都是复数的乘方运算。
96                      raise NotImplementedError('(a+bi)^(c+di)')
97              n = n.re  # 已确定n的虚部是0，提取其实部
98  # 对于复数t和实数n，令t = re^{iθ}，则t^n = r^n cos nθ + ir^n sin nθ。
99          r = self.abs() ** n
100         phi = n * self.angle()
```

```
101         return Complex(r * math.cos(phi), r * math.sin(phi))
102
103 # 计算以base为底数和self为指数的乘方运算。
104     def __rpow__(self, base):
105         base = to_Complex(base)  # 将对象base转换成Complex类的对象
106         return pow(base, self)
107
108 # 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。
109 def is_zero(x, tol = 1e-15): return abs(x) < tol
110
111 # 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。
112 def is_Complex(obj):
113     return hasattr(obj, 're') and hasattr(obj, 'im')
114
115 # 将一个对象obj转换成Complex类的对象。
116 def to_Complex(obj):
117     if is_Complex(obj):
118         return obj  # 对象obj已经是Complex类的对象，直接返回即可。
119     elif isinstance(obj, tuple):  # 判断obj的类型是不是元组。
120         if len(obj) <= 2:  # 判断obj元组包含的元素的个数是不是不超过2。
121             return Complex(*obj)  # 语法*obj提取obj包含的所有元素。
122         else:
123             raise TypeError('Error: <= 2 numbers expected')
124     else:
125         return Complex(obj)
126
127 # 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。
128 def polar_to_Complex(r = 0, phi = 0):
129     return Complex(r * math.cos(phi), r * math.sin(phi))
130
131 # 在确认形参obj是Complex类的对象后返回其re属性，否则返回obj。
132 def Re(obj):
133     return obj.re if is_Complex(obj) else obj
134
135 # 在确认形参obj是Complex类的对象后返回其im属性，否则返回0。
136 def Im(obj):
137     return obj.im if is_Complex(obj) else 0
138
139 # 将a和b的值代入expr中求值，若求值结果与正确答案value不一致则报错。
```

```
140 def check(expr, a, b, value, verbose = False, rel_tol = 1e-6):
141     if verbose: print('␣␣␣␣␣␣␣␣', a, 'and', b, end = '')
142     try:
143 # 内建函数eval对表达式求值，求值环境包括当前可访问的所有全局和局部变量。
144 # 例如第一次运行时expr的值是'a+b'，形参a和b的值分别是Complex(0, 3)和2，
145 # eval将a和b的值代入expr中求值。若求值过程中出错，则运行except语句块。
146         result = eval(expr)
147     except: # 求值过程可能会抛出异常。
148         print('Error␣in␣evaluating␣' + expr); return
149 
150     if verbose: print('␣->␣', result)
151     rel_err = abs(result - value) / abs(value)
152 # 判断相对误差是否超过一个预先确定的阈值rel_tol(默认值为10^-6)。
153     if rel_err > rel_tol:
154         print('%s␣for␣a=%s␣and␣b=%s␣=␣%s' % (expr, a, b, result))
155         print('␣␣␣␣Correct␣value␣=␣%s␣␣Relative␣error␣=␣%f' %
156               (value, rel_err))
157 
158 # 测试Complex类实现的复数运算。
159 def test(verbose):
160 # 字典testsuite包含了所有的测试数据。它的关键字是要测试的表达式，
161 # 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，
162 # 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。
163     testsuite = {
164             'a+b': [ (Complex(0, 3), 2, Complex(2, 3)),
165                      (2, Complex(0, 3), Complex(2, 3)),
166                      (Complex(2,-3), Complex(-4,5),
167                       Complex(-2,2)) ],
168             'a-b': [ (Complex(0, 3), 2, Complex(-2, 3)),
169                      (2, Complex(0, 3), Complex(2, -3)),
170                      (Complex(2,-3), Complex(-4,5),
171                       Complex(6,-8)) ],
172             'a*b': [ (Complex(0, 3), 2, Complex(0, 6)),
173                      (2, Complex(0, 3), Complex(0, 6)),
174                      (Complex(2,-3), Complex(-4,5),
175                       Complex(7,22)) ],
176             'a/b': [ (Complex(0, 3), 2, Complex(0, 1.5)),
177                      (2, Complex(0, 3), Complex(0, -0.6666667)),
178                      (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]

```

```
9  # 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)。
10     def __call__(self, x):
11         value = 0.0
12         for power in self.poly:
13             value += self.poly[power]*x**power
14         return value
15
16 # 计算self+other
17     def __add__(self, other):
18 # 从self.poly复制一个副本sum
19         sum = self.poly.copy()    #  print(id(sum), id(self.poly))
20 # 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
21 # 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。
22         for power in other.poly:
23             if power in sum:
24                 sum[power] += other.poly[power]
25             else:
26                 sum[power] = other.poly[power]
27         return Polynomial(sum) # 调用构造方法创建一个新的多项式。
28
29 # 根据指数相加和系数相乘的规则计算self*other。
30     def __mul__(self, other):
31         sum = {}
32         for self_power in self.poly:
33             for other_power in other.poly:
34                 power = self_power + other_power
35                 m = self.poly[self_power] * \
36                     other.poly[other_power]
37                 if power in sum:
38                     sum[power] += m
39                 else:
40                     sum[power] = m
41         return Polynomial(sum)  # 调用构造方法创建一个新的多项式。
42
43 # 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。
44     def __str__(self):
45         s = ''
46         for power in sorted(self.poly):
47             s += ' + %g*x^%d' % (self.poly[power], power)
```

```
48          s = s.replace('+␣-', '-␣')
49          s = s.replace('x^0', '1')
50          s = s.replace('␣1*', '␣')
51          s = s.replace('x^1␣', 'x␣')
52          # s = s.replace('x^1', 'x') replaces x^100 by x^00
53          if s[0:3] == '␣+␣':  # remove initial +
54              s = s[3:]
55          if s[0:3] == '␣-␣':  # fix spaces for initial -
56              s = '-' + s[3:]
57          return s
58
59  # 创建了一个多项式对象p1并输出其字符串表示。
60  p1 = Polynomial({0: -1, 2: 1, 7: 3}); print(p1)
61  #  -1 + x^2 + 3*x^7
62  # 创建了一个多项式对象p2并输出其字符串表示。
63  p2 = Polynomial({0: 1, 2: -1, 5: -2, 3: 4}); print(p2)
64  #  1 - x^2 + 4*x^3 - 2*x^5
65  p3 = p1 + p2; print(p3)
66  #  4*x^3 - 2*x^5 + 3*x^7
67  p4 = p1 * p2; print(p4)
68  #  -1 + 2*x^2 - 4*x^3 - x^4 + 6*x^5 + x^7 - 3*x^9 + 12*x^10
69  #   - 6*x^12
70  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:

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

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Diff，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64行至第65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):  # 计算数值结果和精确结果之间的相对误差
8          if self.exact is not None:
9              df_numerical = self(x)
10             df_exact = self.exact(x)
11             return abs( (df_exact - df_numerical) / df_exact )
12
13 class Forward1(Differentiation):  # 一阶向前差分
14     def __call__(self, x):
15         f, h = self.f, self.h
```

```
16          return (f(x+h) - f(x))/h
17
18  class Backward1(Differentiation):  # 一阶向后差分
19      def __call__(self, x):
20          f, h = self.f, self.h
21          return (f(x) - f(x-h))/h
22
23  class Central2(Differentiation):  # 二阶中心差分
24      def __call__(self, x):
25          f, h = self.f, self.h
26          return (f(x+h) - f(x-h))/(2*h)
27
28  class Central4(Differentiation):  # 四阶中心差分
29      def __call__(self, x):
30          f, h = self.f, self.h
31          return (4./3)*(f(x+h) - f(x-h))  /(2*h) - \
32                 (1./3)*(f(x+2*h) - f(x-2*h))/(4*h)
33
34  # 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
35  # 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
36  # 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。
37  def table(f, x, h_values, methods, dfdx=None):
38      print('%-10s' % 'h', end='␣')
39      for h in h_values: print('%-8.2e' % h, end='␣')
40      print()
41      for method in methods:
42          print('%-10s' % method.__name__, end='␣')
43          for h in h_values:
44              if dfdx is not None:
45                  d = method(f, h, dfdx)
46                  output = d.get_error(x)
47              else:
48                  d = method(f, h)
49                  output = d(x)
50              print('%-8.6f' % output, end='␣')
51          print()
52
53  import math
54  def g(x): return math.exp(x*math.sin(x))
```

```
55
56 # 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
57 # g'(x) = (x cos x + sin x)e^{x sin x}并保存在dgdx中。
58 import sympy as sym
59 sym_x = sym.Symbol('x')  # sym_x是一个sympy变量
60 sym_gx = sym.exp(sym_x*sym.sin(sym_x)) # sym_gx是一个sympy函数
61 sym_dgdx = sym.diff(sym_gx, sym_x) # sym_dgdx是sym_gx的一阶导函数
62 dgdx = sym.lambdify([sym_x], sym_dgdx) # 把sym_dgdx转换成一个Python函数
63
64 table(f=g, x=-0.65, h_values=[10**(-k) for k in range(1, 7)],
65       methods=[Forward1, Central2, Central4], dfdx=dgdx)
```

Listing 5.5: **程序5.4的输出结果**

```
1 h           1.00e-01  1.00e-02  1.00e-03  1.00e-04  1.00e-05  1.00e-06
2 Forward1    0.104974  0.010906  0.001095  0.000110  0.000011  0.000001
3 Central2    0.004611  0.000046  0.000000  0.000000  0.000000  0.000000
4 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方法已实现的功能的补充。 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**

```
1 class Parent:
2     def __init__(self):
```

```
3         self.c = 1
4     def m(self):
5         print('Calling␣m␣in␣class␣Parent')
6
7 class Child1(Parent):
8     def __init__(self):  # 覆盖了父类的构造方法
9         super().__init__()  # 通过super函数调用其父类的构造方法
10        self.d = 2
11    def m(self):  # 覆盖了父类的m方法
12        print('Calling␣m␣in␣class␣Child1')
13
14 class Child2(Parent):
15    def m(self):  # 覆盖了父类的m方法
16        super().m()  # 通过super函数调用其父类的m方法
17        print('Calling␣m␣in␣class␣Child2')
18
19 class Child3(Parent):
20    def __init__(self):  # 覆盖了父类的构造方法
21        self.f = 3
22    def m2(self):
23        print('Calling␣m2␣in␣class␣Child3')
24
25 c1 = Child1()             # c1是Child1类的对象
26 c2 = Child2()             # c2是Child2类的对象
27 c3 = Child3()             # c3是Child3类的对象
28 p = Parent()              # p是Parent类的对象
29 c1.m()                    # 调用Child1类的m方法
30 c2.m()                    # 调用Child2类的m方法
31 c3.m()                    # 调用Parent类的m方法
32 p.m()                     # 调用Parent类的m方法
33 print(c1.__dict__)        #  {'c': 1, 'd': 2}
34 print(c2.__dict__)        #  {'c': 1}
35 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):  # 生成下一个Fibonacci数
6          self.a, self.b = self.b, self.a + self.b
7          if self.a > self.ub:  # 如果下一个数超过了指定上界，则停止。
8              raise StopIteration
9          return self.a
10
11 class FibonacciIterator:
12     def __init__(self, ub):
13         self.ub = ub
14
15     def __iter__(self):
16         return Fibonacci(self.ub)
17
18 fibonacciIterator = FibonacciIterator(64)
19 for f in fibonacciIterator:
20     print (f, end = ' ')        # 1 1 2 3 5 8 13 21 34 55
21 print()
22 print(list(fibonacciIterator))
23 # [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
```

```
35
36      def __ge__(self, other):   # >=: to be implemented
37
38      def __le__(self, other):   # <=: to be implemented
39
40  def test():
41      testsuite = [
42          ('Rational(2, 3) + Rational(-70, 40)',
43            Rational(-13, 12)),
44          ('Rational(-20, 3) - Rational(120, 470)',
45            Rational(-976,141)),
46          ('Rational(-6, 19) * Rational(-114, 18)',
47            Rational(2, 1)),
48          ('Rational(-6, 19) / Rational(-114, -28)',
49            Rational(-28,361)),
50
51          ('Rational(-6, 19) == Rational(-14, 41)', False),
52          ('Rational(-6, 19) != Rational(-14, 41)', True),
53          ('Rational(6, -19) > Rational(14, -41)', True),
54          ('Rational(-6, 19) < Rational(-14, 41)', False),
55          ('Rational(-6, 19) >= Rational(-14, 41)', True),
56          ('Rational(6, -19) <= Rational(14, -41)', False),
57          ('Rational(-15, 8) == Rational(120, -64)', True),
58      ]
59      for t in testsuite:
60          try:
61              result = eval(t[0])
62          except:
63              print('Error in evaluating ' + t[0]); continue
64
65          if result != t[1]:
66              print('Error:  %s != %s' % (t[0], t[1]))
67
68  if __name__ == '__main__':
69      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**

```
1  import math
2
3  class Integrator:
4      def __init__(self, a, b, n):
5          self.a, self.b, self.n = a, b, n
6          self.points, self.weights = self.compute_points()
7
8      def compute_points(self):
9          raise NotImplementedError(self.__class__.__name__)
10
11     def integrate(self, f):  # to be implemented
12     # 将self.points和self.weights代入计算公式求和
13
14 class Trapezoidal(Integrator):
15     def compute_points(self):  # to be implemented
```

```
16
17 class Simpson(Integrator):
18     def compute_points(self):  # to be implemented
19
20 class GaussLegendre(Integrator):
21     def compute_points(self):  # to be implemented
22
23 def test():
24     def f(x): return (x * math.cos(x) + math.sin(x)) * \
25                      math.exp(x * math.sin(x))
26     def F(x): return math.exp(x * math.sin(x))
27
28     a = 2; b = 3; n = 200
29     I_exact = F(b) - F(a)
30     tol = 1E-3
31
32     methods = [Trapezoidal, Simpson, GaussLegendre]
33     for method in methods:
34         integrator = method(a, b, n)
35         I = integrator.integrate(f)
36         rel_err = abs((I_exact - I) / I_exact)
37         print('%s:␣%g' % (method.__name__, rel_err))
38         if rel_err > tol:
39             print('Error␣in␣%s' % method.__name__)
40
41 if __name__ == '__main__':
42     test()
```

第六章　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的内建类型，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、min和max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],
```

```
3                dtype=int32)
4 In[2]: a[3], a[1:7:2]
5 Out[2]: (49, array([  9,  49, 121], dtype=int32))
6 In[3]: a[:6:3] = 361; a
7 Out[3]: array([361,   9,  25, 361,  81, 121, 169, 225],
8                dtype=int32)
9 In[4]: a[::-1]
10 Out[4]: array([225, 169, 121,  81, 361,  25,   9, 361],
11                dtype=int32)
12 In[5]: for i in a: print(np.sqrt(i), end='␣')
13 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($0 \leq i \leq n-1$)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**

```
1 In[1]: def f(x, y): return x * 4 + y + 1
2 In[2]: h = np.fromfunction(f, (3, 4), dtype=int); h
3 Out[2]:
4 array([[ 1,  2,  3,  4],
5        [ 5,  6,  7,  8],
6        [ 9, 10, 11, 12]])
7 In[3]: h[1, 2], h[0, 3], h[2, 2]
8 Out[3]: (7, 4, 11)
9 In[4]: h[1:3], h[1:3,], h[1:3,:], h[0]
10 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,  6,  7,  8,  9, 16, 11, 12])
In[15]: p[9]=99; h
Out[15]:
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 99, 11, 12]])
In[16]: a = np.arange(1000000); b = a[:100].copy()
In[17]: del a  # the memory of array a can be released
```

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$\mathbf{A} = \mathbf{U\Sigma V^T} = [\mathbf{u}_1, ..., \mathbf{u}_m]\mathbf{\Sigma}[\mathbf{v}_1, ..., \mathbf{v}_n]^T$，其中$\mathbf{\Sigma} = \left(\begin{array}{c|c} diag(\sigma_1, ..., \sigma_r) & \mathbf{0}_{r,n-r} \\ \hline \mathbf{0}_{m-r,r} & \mathbf{0}_{m-r,n-r} \end{array}\right)$，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$\mathbf{\Sigma}$。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1.15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1.17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1.1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**

```
import numpy as np
import scipy.sparse as sps; import scipy.sparse.linalg as spla
import matplotlib.pyplot as plt
```

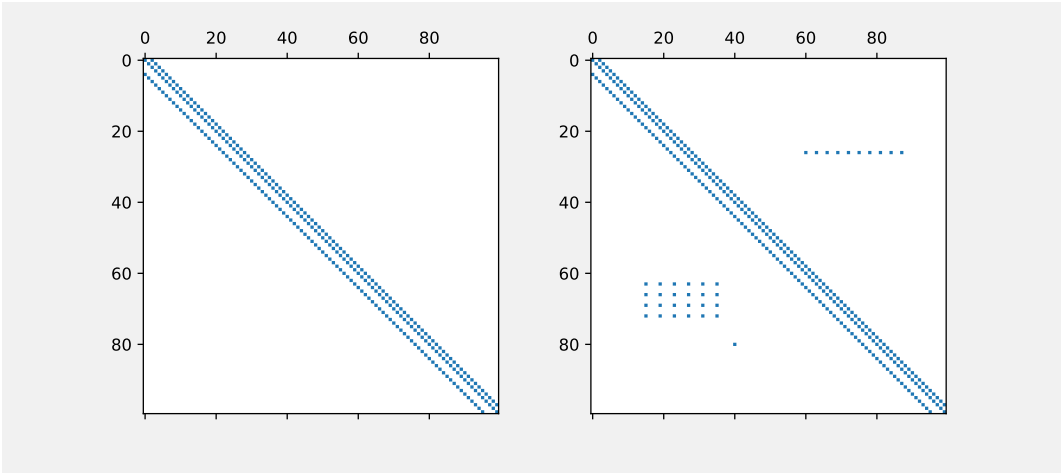

图 6.1: 稀疏矩阵的结构

```
4
5  fig, axes = plt.subplots(1, 2, figsize=(9, 4), dpi = 300)
6  N = 100
7  m = sps.diags([1, -2, 3], [-4, 0, 2], [N, N], format='dok')
8  axes[0].spy(m, markersize=1)
9  m[63:75:3, 15:39:4] = np.random.randint(10, 20, (4, 6))
10 m[80, 40] = 9
11 m[26, 60:90:3] = np.random.randint(40, 80, 10)
12 axes[1].spy(m, markersize=1)
13
14 evals, evecs = spla.eigs(m, k=5, which='LM')
15 print(evals.shape, evecs.shape) # (5,) (100, 5)
16 t = [np.allclose(m.dot(evecs[:,i]), evals[i] * evecs[:,i])
17      for i in range(5)]
18 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数组进行矩阵计算。

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方程1.1。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]
```

第七章　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**

```
1 sum = 0
2 for i in range(1, 10):
3     sum += i
4 print("The␣sum␣of␣1␣to␣10␣is␣%d" % sum)
5 # 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[HL2020]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除以零的警告。

Listing 7.2: 高斯消去法第一个版本

```
import numpy as np

def Gaussian_elimination_v1(A):
    m, n = np.shape(A)
    for j in range(n - 1):
        for i in range(j + 1, m):
            A[i, :] -= (A[i, j] / A[j, j]) * A[j, :]
    return A

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\temp.py:7:
8    RuntimeWarning: divide by zero encountered in double_scalars
9    A[i, :] -= (A[i, j] / A[j, j]) * A[j, :]
10 C:\Users\user\.spyder-py3\temp.py:7:
11   RuntimeWarning: invalid value encountered in multiply
12   A[i, :] -= (A[i, j] / A[j, j]) * A[j, :]
```

程序中只有第7行有除法运算。为了查明出错的原因，需要在第7行设置断点进行调试运行。由于程序对**B**的输出有错，所以先在第14行以**B**作为实参调用函数的语句处设置断点。首先用鼠标点击左边编辑窗口中的第14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。此时编辑窗口第14行行号的右边出现了一个红色的圆点，表示这一行已经设置断点。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。

和正常运行不同，调试运行会在所有设置的断点处停下，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。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14行行号的右边出现一个蓝色箭头，表示已在这一行停下。右下角的控制台也显示了部分程序，并在第14行左边显示一个箭头。此时需要在第7行设置断点。为了使程序继续运行直到下一个断点(即第7行)，点击Debug菜单的菜单项“Continue”或使用快捷键Ctrl+F5。程序停下后，为了观察程序中变量的值，在控制台输入“A, i, j”，输出结果如图7.1所示。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”。

为了使程序单步运行，点击Debug菜单的菜单项“Step Into”或使用快捷键Ctrl+F11。此时编辑窗口第6行行号的右边出现一个蓝色箭头，表示已在这一行停下。继续上述操作，则程序又

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

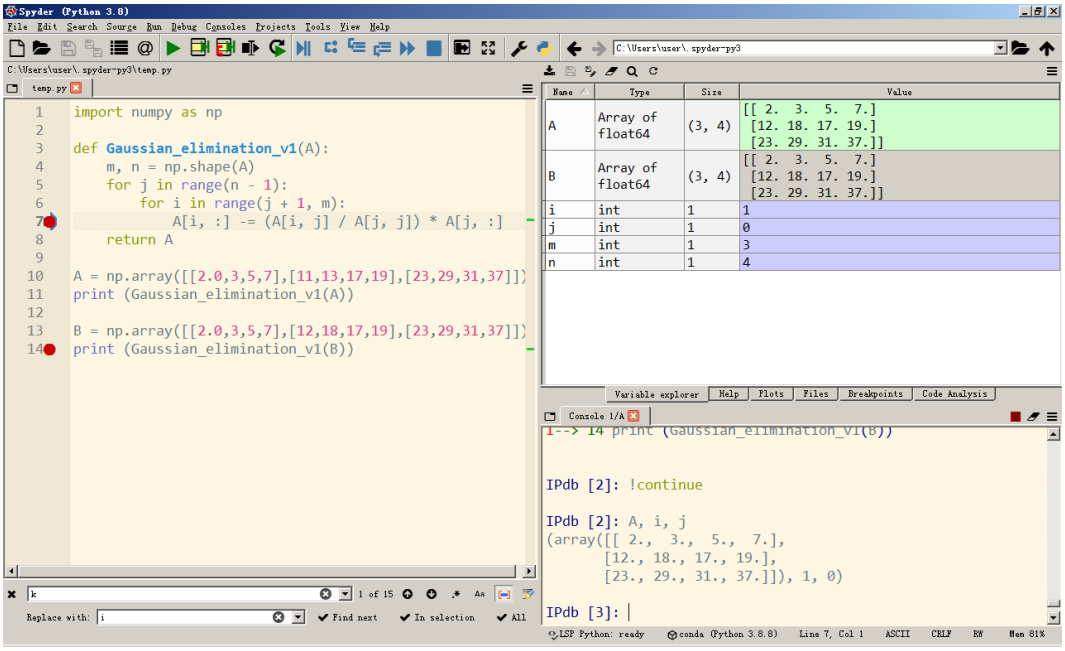

图 7.1: 设置断点进行调试

停在第7行。反复进行以上“Step Into”或“Continue”操作若干次，当i=2且j=1时程序的状态如图7.2所示。 此时作为除数的A[j,j]的值为0，导致了错误。已经找到出错原因后，为了终止

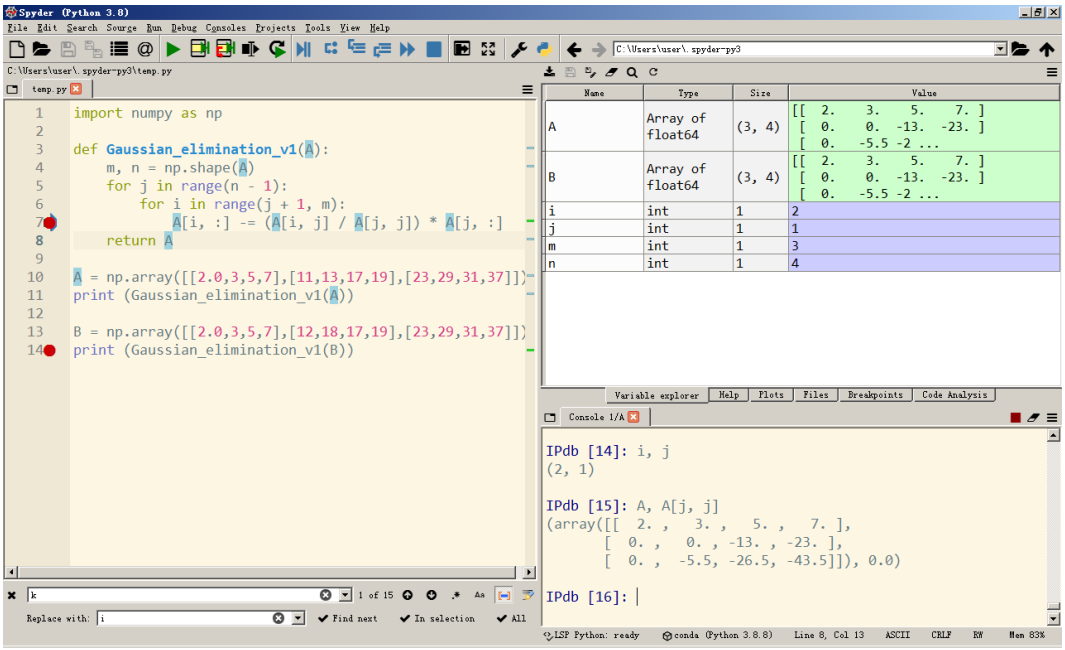

图 7.2: 设置断点进行调试

调试运行，点击Debug菜单的菜单项“Stop”或使用快捷键Ctrl+Shift+F12。找到以上出错处的另一种效率更高的方式是在第7行之前加上一个条件语句判断A[j,j]的值是否为0。在判断为是的分支处第8行设置断点(程序7.4)，

Listing 7.4: **设置条件断点**

```
1  import numpy as np
2
3  def Gaussian_elimination_v1(A):
4      m, n = np.shape(A)
5      for j in range(n - 1):
6          for i in range(j + 1, m):
7              if abs(A[j, j]) < 1e-10:
8                  i += 0
9              A[i, :] -= (A[i, j] / A[j, j]) * A[j, :]
10     return A
11 ...
```

程序7.5处理了A[j, j]值为0的情形。第7行在第j列的第i行及以下的元素构成的向量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i的。第8行判断如果p大于0，则在第9行交换索引值为p+i的行和索引值为i的行。第10行判断若 A[i, j]不为0再进行消去。返回的

值i是矩阵的秩。

Listing 7.5: 高斯消去法第二个版本

```
import numpy as np

def Gaussian_elimination_v2(A, tol = 1e-10):
    m, n = np.shape(A)
    i = 0
    for j in range(n):
        p = np.argmax(abs(A[i:m, j]))
        if p > 0:
            A[[i, p + i]] = A[[p + i, i]]   # 交换这两行
        if abs(A[i, j]) > tol:
            for r in range(i + 1, m):
                A[r, j:] -= (A[r, j] / A[i, j]) * A[i, j:]
            i += 1
            if i >= m: break
    return A, i
```

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7.6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 7.6: 异常处理

```
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' %\
19           (x, y, gcd(x, y)))
```

Listing 7.7: **程序7.6的运行结果**

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

Python标准库定义了很多内建异常类，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建异常类，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7.8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 7.8: **自定义异常类**

```
1  class InputRangeError(Exception):
2      """Raised when an input is not in suitable range
3         Attributes:
4             message -- explanation of suitable range
5      """
6      def __init__(self, message):
7          self.message = message
8
9  def gcd(a, b):
10     if a <= 0 or b <= 0:
11         raise InputRangeError('Each␣integer␣should␣be␣positive')
12     while a != b:
13         if a > b:
14             a -= b
15         else:
16             b -= a
17     return a
18
19 import sys
20 try:
21     x = int(sys.argv[1])
22     y = int(sys.argv[2])
23     print('The␣greatest␣common␣divisor␣of␣%d␣and␣%d␣is␣%d' %\
24             (x, y, gcd(x, y)))
25 except IndexError:
26     print('Two␣arguments␣must␣be␣supplied␣on␣the␣command␣line')
27 except ValueError:
```

```
28      print('Each␣argument␣should␣be␣an␣integer.')
29  except InputRangeError as ex:
30      print(ex.message)
31  finally:
32      print("executing␣finally␣clause")
```

Listing 7.9: **程序7.8的运行结果**

```
1  In[1]: run d:\python\src\gcd_ex.py -48 126
2  Out[1]: Each integer should be positive
3          executing finally clause
4  In[2]: run d:\python\src\gcd_ex.py 48 126
5  Out[2]: The greatest common divisor of 48 and 126 is 6
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f，其中filename是文件名，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(7.10)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 7.10: **文本文件rainfall.dat**

```
1  Average rainfall (in mm) in HEFEI: 459 months between 1951 and 1990
2  Jan  32.2
```

```
Feb  53.2
Mar  71.8
Apr  92.5
May  101.5
Jun  117.3
Jul  175.7
Aug  117.7
Sep  85.6
Oct  60.7
Nov  51.2
Dec  27.6
Year 988.7
```

程序7.11的前9行定义了一个函数extract_data，它的形参为文件名。第3行读取文件的第一行。第4行定义了一个空的字典rainfall。第5行至第8行的循环每次从文件中读取一行。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(7.12)中。

Listing 7.11: 读写文本文件

```
def extract_data(filename):
    with open(filename, 'r') as infile:
        infile.readline()
        rainfall = {}
        for line in infile:
            if line.find('Year') >= 0: break
            words = line.split()
            rainfall[words[0]] = float(words[1])
    return rainfall

import sys
rainfall = extract_data('D:/Python/src/rainfall.dat')
max = -sys.float_info.max; min = sys.float_info.max; sum = 0
for month in rainfall.keys():
    rainfall_month = rainfall[month]
    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 7.12: **rainfall_stat.dat**

```
1 The maximum rainfall of 175.7 occurs in Jul
2 The minimum rainfall of 27.6 occurs in Dec
3 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7.13的第2行至第3行从文件 scores.csv(7.14)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(7.15)中。

Listing 7.13: **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 rows = len(table); cols = len(table[0])
6
7 for r in range(1, rows):
```

```
    for c in range(1, cols):
        table[r][c] = float(table[r][c])

table[0].append('Total')
for r in range(1, rows):
    total = 0
    for c in range(1, cols):
        total += table[r][c]
    table[r].append(total)

row = ['Average']
for c in range(1, cols):
    avg = 0
    for r in range(1, rows):
        avg += table[r][c]
    row.append(avg / (rows - 1))
table.append(row)

pprint.pprint(table)

with open('D:/Python/src/scores2.csv', 'w', newline='') as outfile:
    writer = csv.writer(outfile)
    for row in table:
        writer.writerow(row)
```

Listing 7.14: **scores.csv**

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

Listing 7.15: **scores2.csv**

```
Name,Math,Physics,English,Total
Tom,95.0,91.0,81.0,267.0
Jerry,89.0,82.0,86.0,257.0
Mary,83.0,80.0,96.0,259.0
Betty,88.0,96.0,93.0,277.0
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7.16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(7.17)。第12行至第13行从contacts.json中读取数据，然后由第15行输出(7.18)。

Listing 7.16: **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 7.17: **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 7.18: **程序7.16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7.19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(7.20)。

Listing 7.19: **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)
11
12 with open('D:/Python/src/contacts.pickle', 'rb') as infile:
13     x = pickle.load(infile)
14
15 pprint.pprint(x)
```

Listing 7.20: **程序7.19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}]
```

7.2.6 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 7.21: **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7.5、7.11和7.14，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7.21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 7.22: **程序7.21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)
......
```

第八章　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$\Omega$记号：$f(n) = \Omega(g(n))$当且仅当$g(n) = O(f(n))$。第三种记号是$\Theta$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。 Master定理[CL2009]可求解以下形式的方程：$T(n)=aT(n/b)+f(n)$，其中$a\geq 1$和$b>1$是常数，n是非负整数，$f(n)$是一个函数，n/b等于$\lfloor n/b\rfloor$或$\lceil n/b\rceil$。

- 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

```
1  def insertion_sort(s):
2      n = len(s)
3      for i in range(1, n):
4          value = s[i]; print('insert␣%2d:␣' % value, end = '␣')
5          pos = i
6          while pos > 0 and value < s[pos - 1] :
7              s[pos] = s[pos - 1]
8              pos -= 1
9          s[pos] = value
10         print(s)
11
12 s = [21, 73, 6, 67, 99, 60, 77, 5, 51, 32]; print(s)
13 insertion_sort(s)
```

Listing 8.2: 插入排序的运行过程

```
1  [21, 73, 6, 67, 99, 60, 77, 5, 51, 32]
2  insert 73:  [21, 73, 6, 67, 99, 60, 77, 5, 51, 32]
3  insert  6:  [6, 21, 73, 67, 99, 60, 77, 5, 51, 32]
4  insert 67:  [6, 21, 67, 73, 99, 60, 77, 5, 51, 32]
5  insert 99:  [6, 21, 67, 73, 99, 60, 77, 5, 51, 32]
6  insert 60:  [6, 21, 60, 67, 73, 99, 77, 5, 51, 32]
7  insert 77:  [6, 21, 60, 67, 73, 77, 99, 5, 51, 32]
8  insert  5:  [5, 6, 21, 60, 67, 73, 77, 99, 51, 32]
9  insert 51:  [5, 6, 21, 51, 60, 67, 73, 77, 99, 32]
10 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计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

```
def merge_ordered_lists(s1, s2):
    t = []
    i = j = 0
    while i < len(s1) and j < len(s2):
        if s1[i] < s2[j]:
            t.append(s1[i]); i += 1
        else:
            t.append(s2[j]); j += 1
    t += s1[i:]
    t += s2[j:]
    print('%s␣+␣%s␣=>␣%s' % (s1, s2, t));
    return t

def merge_sort(s):
    if len(s) <= 1:
        return s
    mid = len(s) // 2
    print('%s␣->␣%s␣+␣%s' % (s, s[:mid], s[mid:]));
    left = merge_sort(s[:mid])
    right = merge_sort(s[mid:])
    return merge_ordered_lists(left, right)

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

Listing 8.4: 归并排序的运行过程

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

```
def linear_search(s, k):
    for i in range(len(s)):
        if s[i] == k: return i
    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
```

```
3     while low <= high:
4         mid = (high + low) // 2
5         print('(%2d, %2d) low = %d, mid = %d, high = %d'
6               % (k, s[mid], low, mid, high))
7         if k == s[mid]:
8             return mid
9         elif k < s[mid]:
10            high = mid - 1
11        else:
12            low = mid + 1
13    return -1
14
15 s = [5, 6, 21, 32, 51, 60, 67, 73, 77, 99]
16 print(binary_search(s, 77)); print(binary_search(s, 31))
```

Listing 8.7: **二分查找的运行过程**

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

```
1 def exhaustive_search_3_sum(s, x):
2     n = len(s)
3     for i in range(n - 2):
4         for j in range(i + 1, n - 1):
```

```
5                 for k in range(j + 1, n):
6                     if s[i] + s[j] + s[k] == x:
7                         return s[i], s[j], s[k]
8         return ()
9
10 s = [21, 73, 6, 67, 99, 60, 77, 5, 51, 32]
11 print(exhaustive_search_3Sum(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          subset = []
5          for j in range(n):
6              if (i >> j) % 2 == 1: subset.append(s[j])
7          if sum(subset) == x: return subset
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单行语句**

```
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。

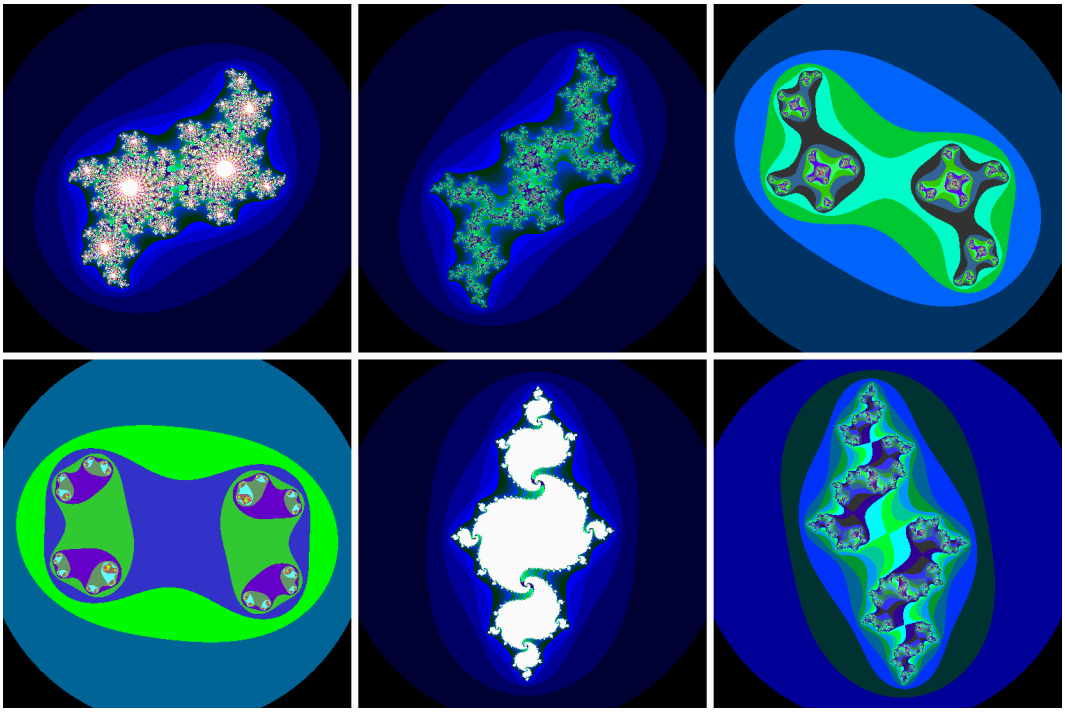

图 8.1: Julia集生成的图片

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)。

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

```
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_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
```

```
3 setup(ext_modules=cythonize("calc_z_cython.pyx"))
```

Listing 8.18: **将程序8.16编译成为扩展模块**

```
1 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**

```
1  import time; import array; import numpy as np
2  from calc_z_cython import calc_z
3
4  ...
5
6  def calc_Julia(show, length, max_iter):
7      ...
8      start_time = time.time()
9      output = calc_z_python(max_iter, zs, c)
10     end_time = time.time()
11     print("%.4fs" % (end_time - start_time))  #  0.2891s
12     if show: show_image(output, length, max_iter)
13
14 ...
```

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，分别测量插入排序(8.1)、归并排序(8.3)和快速排序(1.15)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. 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。

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.

[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.