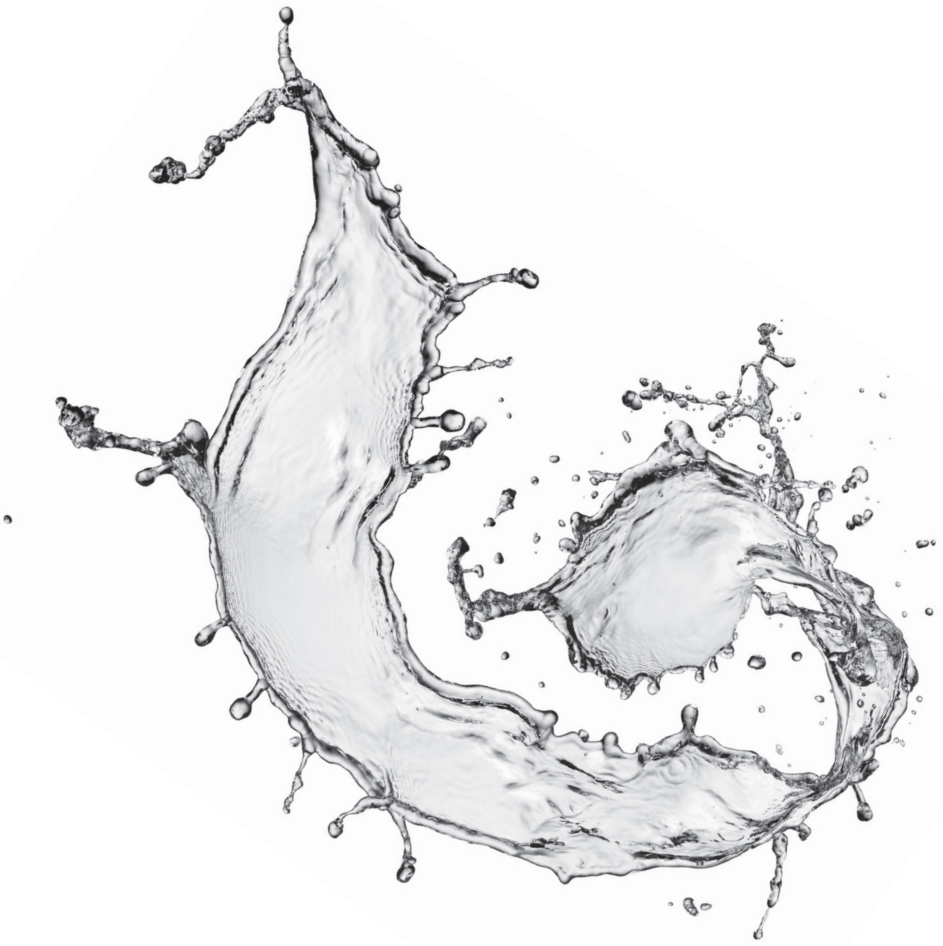
大数据人才培养规划教材

以解决实际问题为**学习目标**
以实战案例贯穿为**学习手段**



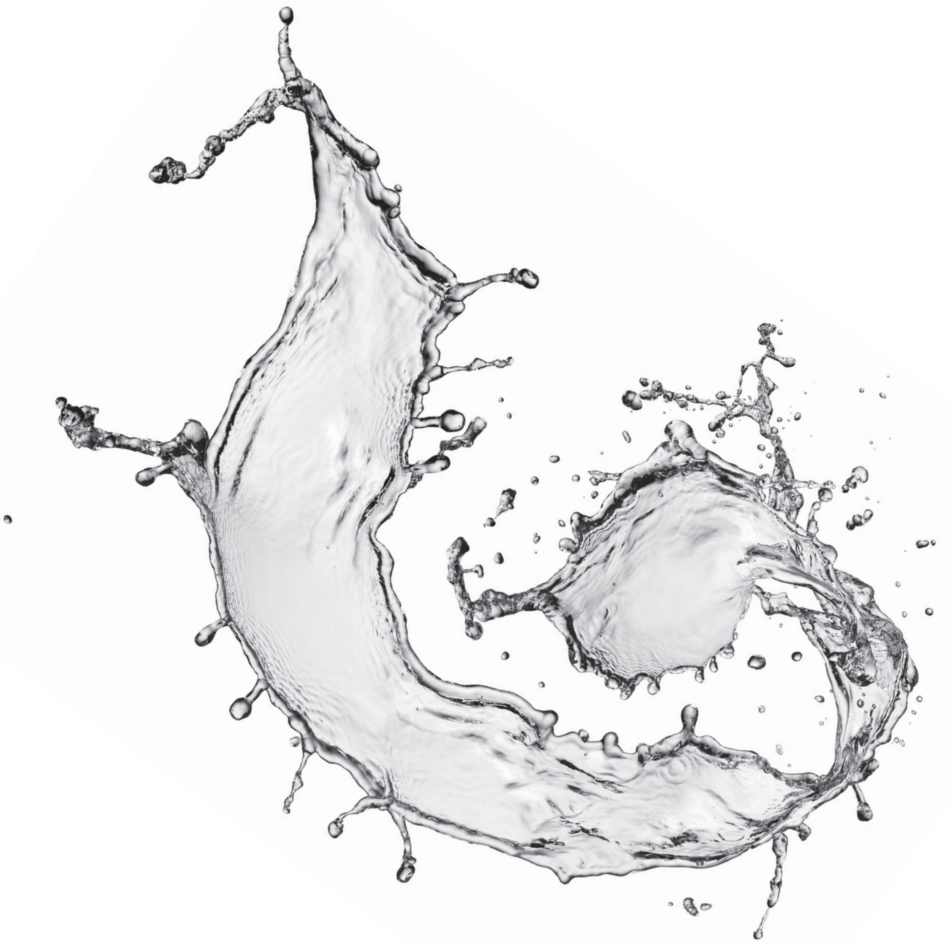
Python 编程基础

Python Programming

张健 张良均 ◉ 主编
何燕 张敏 姜鹏辉 ◉ 副主编

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

大数据人才培养规划教材



Python 编程基础

Python Programming

张健 张良均 ◉ 主编
何燕 张敏 姜鹏辉 ◉ 副主编

人 民 邮 电 出 版 社
北 京

图书在版编目（CIP）数据

Python编程基础 / 张健，张良均主编. -- 北京 : 人民邮电出版社，2018.3
大数据人才培养规划教材
ISBN 978-7-115-47449-0

Ⅰ. ①P… Ⅱ. ①张… ②张… Ⅲ. ①软件工具－程序设计 Ⅳ. ①TP311.561

中国版本图书馆CIP数据核字(2018)第028644号

内 容 提 要

本书采用以任务为导向的编写模式，全面地介绍了 Python 编程基础及其相关知识的应用，讲解了如何利用 Python 的知识解决部分实际问题。全书共 7 章，第 1 章介绍学习 Python 的准备工作，包括 Python 的由来与发展、Python 环境搭建、编辑器介绍与安装等。第 2～5 章和第 7 章主要介绍 Python 的基础知识、数据类型、程序流程控制语句、函数和文件基础等内容。第 6 章讲解了 Python 面向对象的编程。本书每个章节都包含了实训与课后习题。通过习题和操作实践，读者可以巩固所学的内容。

本书可以作为高校大数据技术类专业教材，也可作为大数据技术爱好者的自学用书。

◆ 主　　编　张　健　张良均
副 主 编　何　燕　张　敏　姜鹏辉
责任编辑　左仲海
责任印制　马振武

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京隆昌伟业印刷有限公司印刷

◆ 开本：787×1092　1/16
印张：11.5　　2018 年 3 月第 1 版
字数：256 千字　　2018 年 3 月北京第 1 次印刷

定价：39.80 元

读者服务热线：(010)81055256　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

大数据专业系列图书
编写委员会

序 PREFACE

随着大数据时代的到来，移动互联网络和智能手机迅速普及，多种形态的移动互联应用蓬勃发展，电子商务、云计算、互联网金融、物联网等不断渗透并重塑传统产业，大数据当之无愧地成为了新的产业革命核心。

未来 5～10 年，我国大数据产业将会是一个飞速发展时期，社会对大数据相关专业人才有着巨大的需求。目前，国内各大高校都在争相设立或准备设立大数据相关专业，以适应地方产业发展对战略性新兴产业的人才需求。

人才培养离不开教材，大数据专业是 2016 年才获批的新专业，目前还没有成套的系列教材，已有教材也存在企业案例缺失等亟须解决的问题。由广州泰迪智能科技有限公司和人民邮电出版社策划，校企联合编写的这套图书，尤如大旱中的甘露，可以有效解决高校大数据相关专业教材紧缺的困境。

实践教学是在一定的理论指导下，通过引导学习者的实践活动，从而传承实践知识、形成技能、发展实践能力、提高综合素质的教学活动。目前，高校教学体系的设置有诸多限制因素，过多地偏向理论教学，课程设置与企业实际应用切合度不高，学生无法把理论转化为实践应用技能。课程内容设置方面看似繁多又各自为“政”，课程冗余、缺漏，体系不健全。本套图书的第一大特点就是注重学生的实践能力培养，根据高校实践教学中的痛点，首次提出“鱼骨教学法”的概念。以企业真实需求为导向，学生学习技能紧紧围绕企业实际应用需求，将学生需掌握的理论知识，通过企业案例的形式进行衔接，达到知行合一、以用促学的目的。

大数据专业应该以大数据技术应用为核心，紧紧围绕大数据应用闭环的流程进行教学，才能够使学生从宏观上理解大数据技术在行业中的具体应用场景及应用方法。高校现有的大数据课程集中在如何进行数据处理、建模分析、调整参数，使得模型的结果更加准确。但是，完整的大数据应用却是一个容易被忽视的部分。本套图书的第二大特点就是围绕大数据应用的整个流程，从数据采集、数据迁移、数据存储、数据

分析与挖掘，最终到数据可视化，覆盖完整的大数据应用流程，涵盖企业大数据应用中的各个环节，符合企业大数据应用真实场景。

我很高兴看到这套书的出版，也希望这套书能给更多的高校师生带来教学上的便利，帮助读者尽快掌握本领，成为有用之才！

教育部长江学者特聘教授
国家杰出青年基金获得者
IEEE Fellow
华南理工大学计算机与工程学院院长

2017 年 12 月

前 言 FOREWORD

随着云时代的来临，Python 语言越来越被程序开发人员喜欢和使用，因为其不仅简单易学，而且还有丰富的第三方程序库和相应完善的管理工具。从命令行脚本程序到 GUI 程序，从图形技术到科学计算，从软件开发到自动化测试，从云计算到虚拟化，所有这些领域都有 Python 的身影。Python 已经深入程序开发的各个领域，并且越来越多的人会学习和使用。Python 同时具有面向对象和函数式编程的特点，它的面向对象比 Java 更彻底，它的函数式编程比 Scala 更人性化。作为一种通用语言，Python 几乎可以用在任何领域和场合。其在软件质量控制、开发效率、可移植性、组件集成、丰富的库支持等方面均处于领先地位。Python 作为大数据时代的核心编程基础技术之一，必将成为高校大数据相关专业的重要课程之一。

本书特色

本书采用以任务为导向的编写模式，深入浅出地介绍了 Python 开发环境搭建、Python 基础知识、程序流程控制语句、函数、面向对象编程、文件基础等内容。每章的内容都由任务描述、任务分析、任务实现、小结、实训和课后习题等部分组成。全书按照解决实际任务的工作流程，逐步展开介绍相关的理论知识点，推导生成可行的解决方案，最后落实在任务实现环节。全书大部分章节紧扣任务需求展开，不堆积知识点，着重于解决思路的启发与解决方案的实施。通过从任务需求到实现这一完整工作流程的体验，读者将对 Python 编程技术真正理解与掌握。

本书适用对象

- 开设有大数据相关课程的高校的教师和学生。

目前国内很多高校的数学、计算机、商务数据分析、自动化、电子信息及金融等专业均开设了数据分析技术相关的课程，目前这一课程的教学仍然主要限于理论介绍。单纯的理论教学过于抽象，学生理解起来往往比较困难，教学效果也不理想。本书提供的基于任务导向和实操练习的教学模式，能够使师生充分发挥互动性和创造性，理论联系实际，使师生获得更佳的效果。

- 数据分析开发人员。

这类人员可以在理解数据分析应用需求和开发设计方案的基础上，基于第三方接口，快速完成数据分析应用及实现开发设计的编程。

- 进行数据分析应用研究的科研人员。

许多科研院所为了更好地对科研工作进行管理，纷纷开发了适应自身特点的科研业务管理系统，并在使用过程中积累了大量的科研信息数据。但是，这些科研业务管理系统一般都没有对这些数据进行深入分析，对数据所隐藏的价值并没有充分分析利用。科研人员需要数据分析工具及有关方法来深挖科研信息的价值，进而提高科研水平。

代码下载及问题反馈

为方便读者实践与练习，对于书中全部任务的数据文件及源代码，读者可登录人民邮电出版社教育社区（www.ryjiaoyu.com）或“泰迪杯”全国数据挖掘挑战赛网站（www.tipdm.org/tj/1308.jhtml）下载。为方便广大教师授课，本书还提供了教学课件 PPT，有需要的读者可通过泰迪大数据挖掘微信公众号（TipDataMining）或者热线电话（40068-40020）进行在线咨询和获取。

我们已经尽最大努力避免在文本和代码中出现错误，但是由于水平有限，编写时间仓促，书中难免出现一些不足和疏漏之处。如果您有更多的宝贵意见，欢迎发送邮件至邮箱 13560356095@qq.com，期待能够得到您真挚的反馈。同时，本书更新内容将及时在“泰迪杯”全国数据挖掘挑战赛网站上发布，读者可以登录网站或关注泰迪大数据挖掘微信公众号查阅相关信息。

编　者

2017 年 11 月

目录 CONTENTS

第 1 章 准备工作

最近十年，Python 在网络爬虫、数据分析、AI、机器学习、Web 开发、金融、运维及测试等领域都有不俗的表现，从来没有哪种语言可以同时在这么多领域扎根。它专注于解决问题、自由开放的社区环境以及丰富的第三方库，各种 Web 框架、爬虫框架、数据分析框架、机器学习框架应有尽有。本章首先从 Python 语言发展和特性开始介绍，然后介绍如何获取与安装 Python，最后学习 Python 环境的搭建和编辑器的安装，最后编写并运行程序。

学习目标

（1）初识 Python，并了解 Python 的发展历史和特性。
（2）掌握 Python 在 Windows 和 Linux/UNIX 平台的安装及环境变量配置。
（3）了解常用的 Python IDE。
（4）认识和使用 PyCharm。
（5）创建一个应声虫程序。

任务 1.1 认识 Python

任务描述

Python 具有强大的科学及工程计算能力，它不但具有以矩阵计算为基础的强大数学计算能力和分析功能，而且还具有丰富的可视化表现功能和简洁的程序设计能力。那么了解 Python 的起源，认识 Python 是怎么样的一门语言是学习 Python 的第一步。

任务分析

（1）认识 Python 是什么。
（2）了解 Python 发展的历史。
（3）了解 Python 的 9 个特性。

1.1.1 初识 Python

Python 是一种结合了解释性、编译性、互动性和面向对象的高层次计算机程序语言，也是一种功能强大而完善的通用型语言，已经具有二十多年的发展历史，成熟且稳定。Python 具有非常简洁而清晰的语法特点，因为它的设计指导思想是，对于一个特定的问题，应该用最好的方法来解决。

Python 具备垃圾回收功能，能够自动管理内存的使用，常被当作脚本语言，用于处理

系统管理任务和网络程序编写；同时支持命令式程序设计、面向对象程序设计、函数式编程、泛型编程多种编程范式，也非常适合完成各种高级任务。

1.1.2 了解 Python 发展历史

Python 的创始人是 Guido van Rossum。1989 年圣诞节期间，Guido 为了打发圣诞节的无趣，开发了这个新的脚本解释程序。Python 这个名字不是来源于蟒蛇，而是源于 Guido 是一个名为“Monty Python”的飞行马戏团的爱好者。

Python 继承了 ABC 语言的特点，Guido 认为，ABC 这种语言非常优美和强大，是专门为非专业程序员设计的。但是 ABC 语言并没有成功，Guido 认为失败的原因是该语言不是开源性语言。于是，Guido 决心在 Python 中避免这种情况，并获取了非常好的效果，完美结合了 C 语言和其他一些语言的特点。

同时，Guido 还想实现在 ABC 中提出过但未曾实现的东西，所以 Python 是在 ABC 的基础上发展起来的，其中受到了 Modula-3（另一种相当优美且强大的语言，为小型团体所设计）的影响，并且结合了 UNIX shell 和 C 语言用户的习惯，一跃成为众多 UNIX 和 Linux 开发者所青睐的开发语言。

1.1.3 了解 Python 特性

Python 语言能广泛用于多种编程领域，无论对于初学者，还是对于在科学计算领域具备一定经验的工作者，它都极具吸引力。其关键特性如下所述。

（1）**简单**。Python 有相对较少的关键字，结构简单；有一个明确定义的语法规则，学习起来更加容易。

（2）**易学**。Python 有极其简单的语法，容易上手。

（3）**免费、开源**。Python 是 FLOSS（自由/开放源码软件）之一。简单地说，用户可以自由地发布这个软件的副本，查看和更改其源代码，并在新的免费程序中使用它。

（4）**广泛的标准库**。Python 最大的优势之一是具有丰富的库，支持许多常见的编程任务，如连接到 Web 服务器、使用正则表达式搜索文本、读取和修改文件等。

（5）**互动模式**。可以从终端输入执行代码并获得结果，可以互动地测试和调试代码。

（6）**可移植**。由于具有开源的本质，Python 已经被移植在许多平台上（经过改动，可以使它能够工作在不同平台上）。

（7）**可扩展**。如果需要一段运行很快的关键代码，或者是编写一些不愿开放的算法，那么可以使用 C 语言或 C++语言完成那部分程序，然后从 Python 程序中调用。

（8）**可嵌入**。Python 可以嵌入 C/C++程序中，为程序用户提供“脚本”功能。

（9）**数据库**。Python 提供与主流数据库对接的接口。

任务 1.2 搭建 Python 环境

根据自己计算机的系统，从 Python 官网下载对应的 Python 3.6.0 版本，成功安装后配置环境变量。在 Windows 系统命令提示符窗口中输入“python”命令，能得到图 1-1 所示

的效果；在 Linux/UNIX 系统终端输入“python3.6”命令，能得到图 1-2 所示的效果。

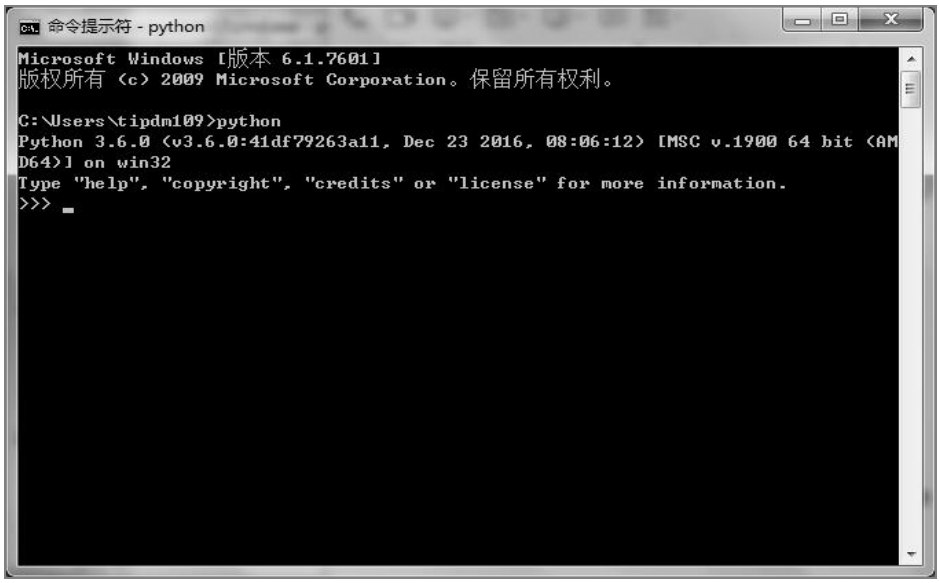


图 1-1　Windows 命令操作界面

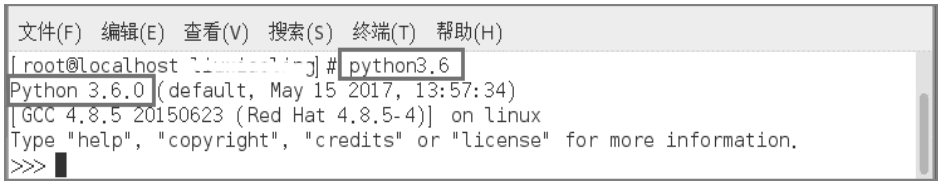


图 1-2　在 Linux/UNIX 中打开 Python

Python 是开源自由软件，Python 的所有开发环境基本都可以从网络上免费下载。目前 Python 有两种主流版本，一个是 2.*x* 版，一个是 3.*x* 版，这两个版本是不兼容的。下载和安装 Python 3.6.0 可以按以下步骤进行。

（1）检查自己的计算机系统，在 Python 官网中下载对应的 Python 3.6.0 版本。

（2）按操作步骤安装 Python 3.6.0。

（3）配置环境变量。

（4）检查 Python 3.6.0 是否安装成功。

1.2.1　在 Windows 系统平台安装 Python 与配置环境变量

1. 在 Windows 系统平台安装 Python

在 Windows 系统平台安装 Python 的具体操作步骤如下。

（1）打开浏览器，访问 Python 官网 https://www.Python.org，如图 1-3 所示。

（2）选择“Downloads”菜单下的“Windows”命令，如图 1-4 所示。

（3）找到 Python 3.6.0 的安装包，如果 Windows 系统版本是 32 位，则单击“Windows x86 executable installer 超链接，然后下载；如果 Windows 系统版本是 64 位的，则单击

“Windows x86-64 executable installer”超链接，然后下载，如图 1-5 所示。

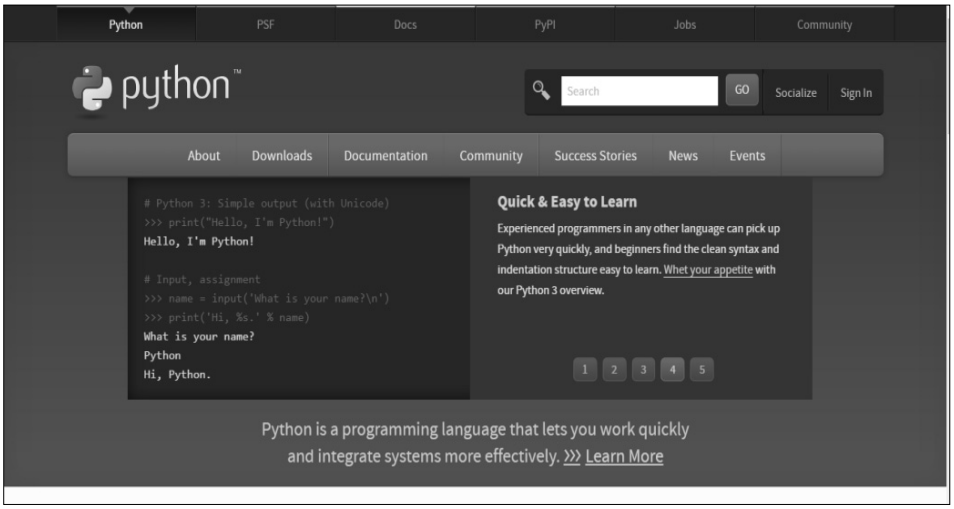


图 1-3　Python 官网

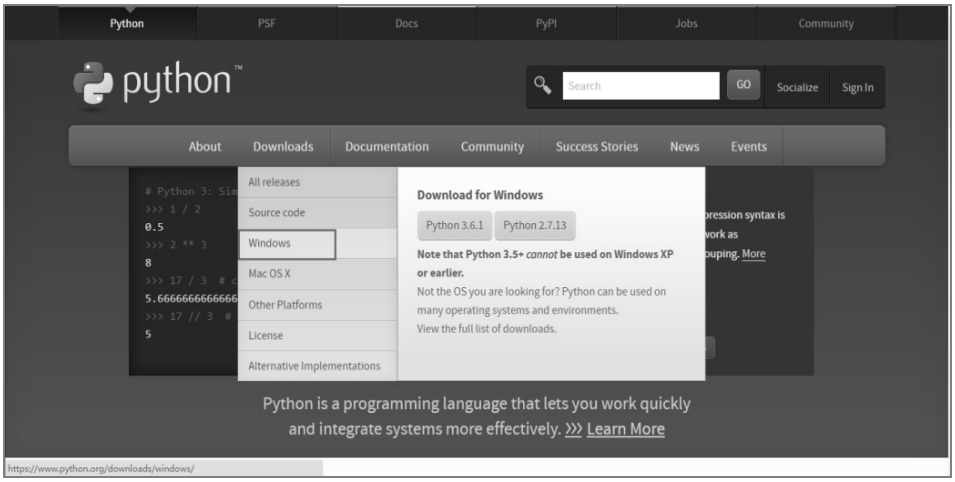


图 1-4　选择 Windows 版本

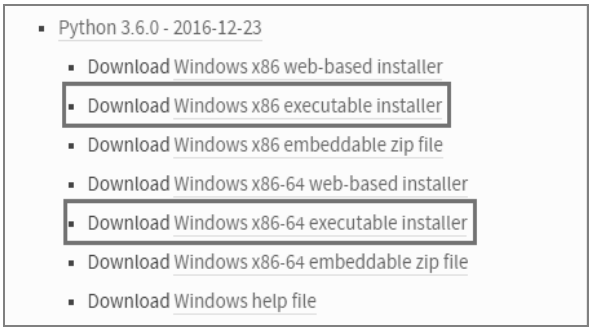


图 1-5　下载安装包

（4）下载完成后，双击运行所下载的文件，弹出 Python 安装向导窗口，如图 1-6 所示，勾选“Add Python 3.6 to PATH”复选框，然后单击“Customize installation”按钮。

图 1-6　安装向导窗口

（5）弹出界面如图 1-7 所示，保持默认选择，单击“Next”按钮，在弹出的界面中可以修改安装路径，如图 1-8 所示。

图 1-7　单击“Next”按钮

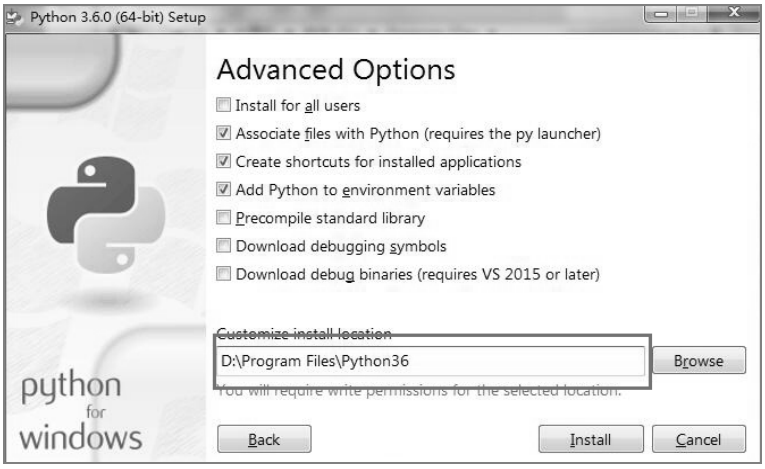


图 1-8　安装路径

（6）安装完之后，会弹出安装成功的提示界面，如图 1-9 所示。

图 1-9　安装完成

2. PATH 环境变量设置

打开命令提示符窗口（操作方法详见 1.2.3 小节），输入“python”命令，会出现以下两种情况。

情况一：出现图 1-1 所示的界面，说明 Python 已经安装成功。

情况二：出现图 1-10 所示的界面，这是因为 Windows 系统会根据一个 PATH 环境变量设定的路径去查找 python.exe，如果没有找到就会报错。

图 1-10　找不到 Python

如果出现情况二，则需要将 python.exe 所在的路径添加到 PATH 中，以 Windows 7 为例，具体步骤如下。

（1）右击“计算机”图标，选择“属性”命令，如图 1-11 所示。

图 1-11　选择“属性”命令

（2）在弹出的窗口中选择“高级系统设置”选项，如图 1-12 所示。

图 1-12　选择“高级系统设置”选项

（3）在弹出的对话框中单击“环境变量”按钮，如图 1-13 所示。

（4）在弹出的对话框中找到“系统变量”列表框中的“Path”选项，如图 1-14 所示。

（5）双击“Path”选项，在弹出的对话框中可编辑变量值，在“变量值”文本框中添加 Python 的安装路径，并用“;”（英文状态下的分号）隔开。例如安装路径为 D:\Program Files\Python36，则添加的变量值为“; D:\Program Files\Python36 \”，如图 1-15 所示。

（6）单击“确定”按钮。再次打开命令提示符窗口，输入“python”命令，即会出现图 1-1 所示的界面，说明已经配置好 Python 的环境变量。

图 1-13　单击“环境变量”按钮

图 1-14　找到“Path”

图 1-15　添加路径

1.2.2　在 Linux/UNIX 系统平台安装 Python 与配置环境变量

大多数 Linux 系统发行版，如 CentOS、Debian、Ubuntu 等，都自带了 Python 2.*x* 版本的主程序。目前最新版的 Ubuntu 已经自带了 Python 3.*x* 版本的主程序，对于没有安装 Python 3.*x* 版本的系统，用户可自行安装。

以 CentOS 7 为例，安装 Python 3.6.0 的步骤如下。

（1）打开浏览器，访问 Python 官网 https://www.Python.org，如图 1-16 所示，单击“Linux/UNIX”超链接。

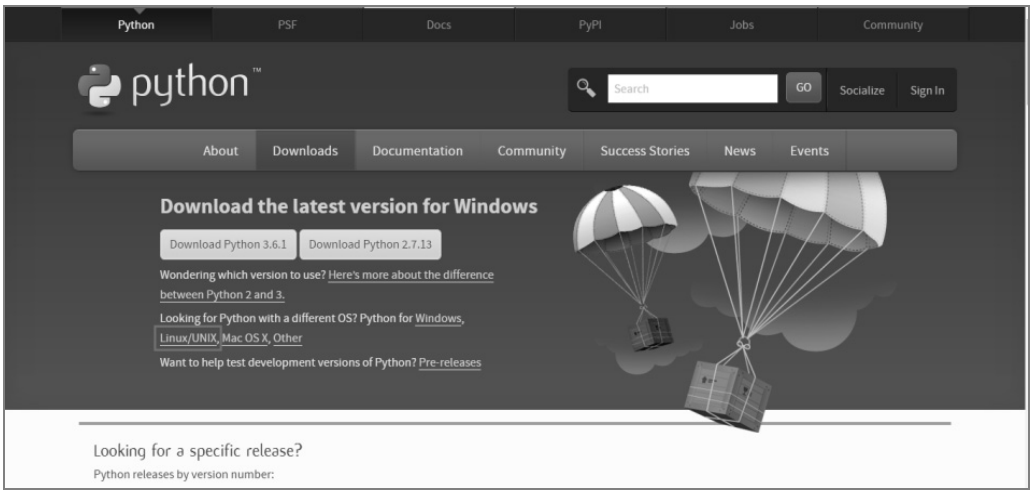


图 1-16　选择 Linux/UNIX 版本

（2）找到 Python 3.6.0 的压缩包，单击“Gzipped source tarball”超链接，如图 1-17 所示，然后下载 Gzipped source tarball。

图 1-17　下载安装包

（3）在安装 Python 之前，要确保系统中已经有了所有必要的开发依赖。执行命令 1-1 即可安装所有的依赖。

命令 1-1　安装必要的开发依赖

```
yum -y groupinstall development
yum -y install zlib-devel
```

（4）解压下载好的“Python-3.6.0.tgz”文件，打开终端（Terminal），并进入解压后的 Python-3.6.0 目录，如图 1-18 所示。

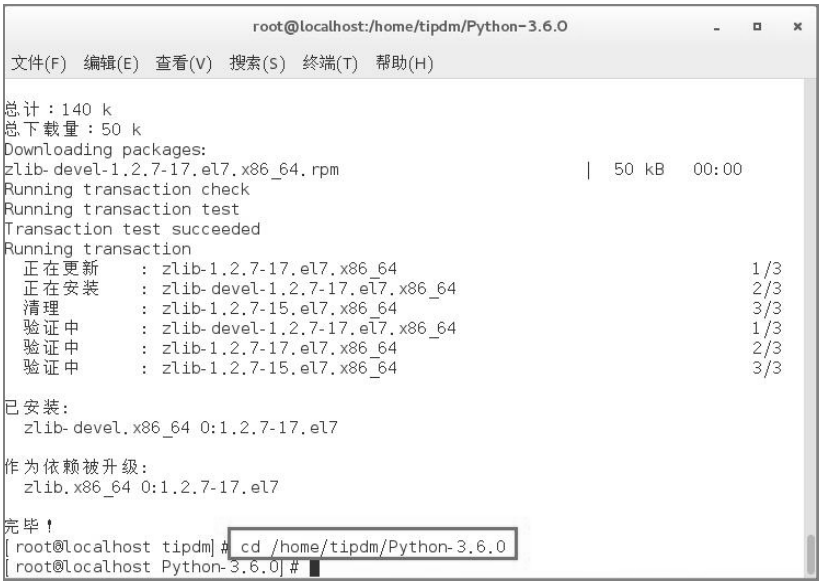


图 1-18　解压下载好的“Python-3.6.0.tgz”文件

（5）自定义安装目录后进行安装，如安装到/ usr/local/python3 目录下，可执行命令 1-2。

命令 1-2　安装到所需路径下

```
./configure --prefix=/usr/local/python3
make && make install
```

其中，--prefix 选项是配置安装的路径。如果不配置该选项，安装后可执行文件默认放在/usr/local/bin，库文件默认放在/usr/local，配置文件默认放在/usr/local/etc，其他资源文件放在/usr/local，这样会比较凌乱。如果配置了--prefix 选项，则可以把所有资源文件放在自定义目录下。

./configure 命令执行完毕之后，会创建一个文件 creating Makefile，供 make 命令使用，执行 make install 之后就会把程序安装到指定的目录中去。

（6）安装成功之后，进入自定义安装目录，执行“ln -s -f /usr/local/python3/bin/python3.6 /usr/bin/python3.6”命令，创建软连接，如图 1-19 所示。

（7）执行“python3.6 -v”命令，查看 Python 3.6.0 是否安装成功。执行“python3.6”命令，如果出现图 1-2 所示的界面，即说明安装成功。

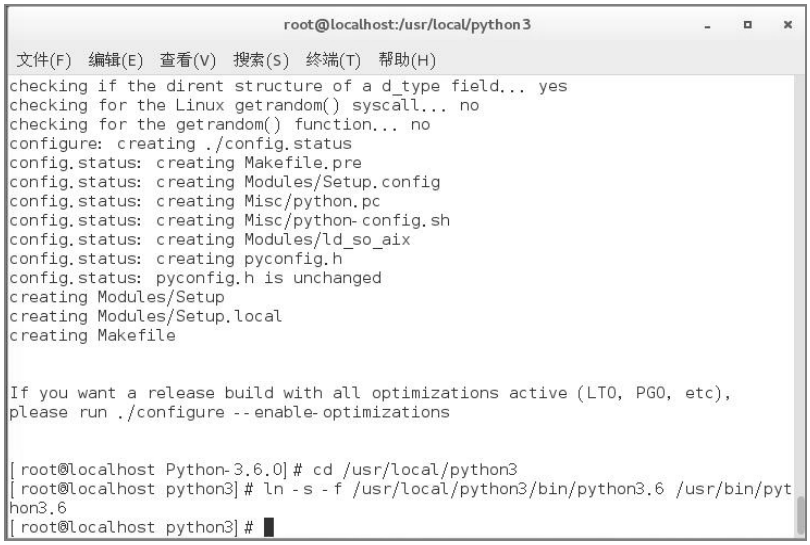


图 1-19　创建软连接

1.2.3　开启 Python 之旅

安装 Python 成功之后，就可以正式开始 Python 之旅了。Python 的打开方式有 3 种：Windows 系统的命令行工具（cmd）、带图形界面的 Python Shell——IDLE、命令行版本的 Python Shell——Python 3.6。下面简单介绍这 3 种方式的具体操作。

1．Windows 系统的命令行工具（cmd）

cmd 即计算机命令行提示符，是 Windows 环境下的虚拟 DOS 窗口。在 Windows 系统下，打开 cmd 有 3 种方法。

（1）按“Win+R”组合键，其中“Win”键是键盘上的开始菜单键，如图 1-20 所示，在弹出的对话框中输入“cmd”，如图 1-21 所示。单击“确定”按钮，即可打开 cmd。

图 1-20　Win 键

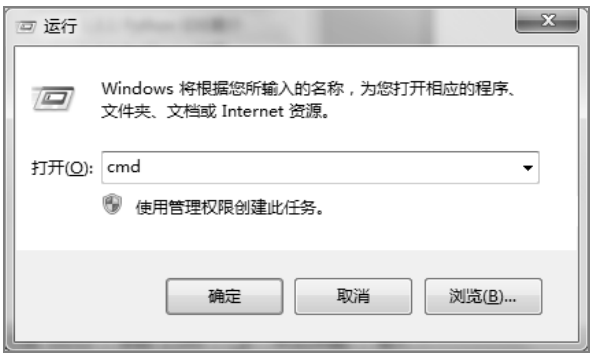


图 1-21　输入“cmd”

（2）通过“所有程序”列表查找搜索到 cmd，如图 1-22 所示。选择“cmd.exe”选项或按回车键即可打开 cmd。

（3）在 C:\Windows\System32 路径下找到 cmd.exe，如图 1-23 所示，双击“cmd.exe”文件。

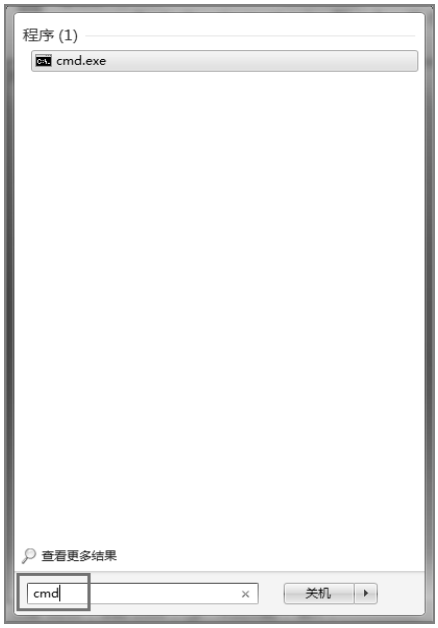


图 1-22　搜索界面

图 1-23　双击“cmd.exe”

（4）打开 cmd，输入“python”，按回车键，如果出现“>>>”符号，说明已经进入 Python 交互式编程环境，如图 1-24 所示。此时输入“exit()”即可退出。

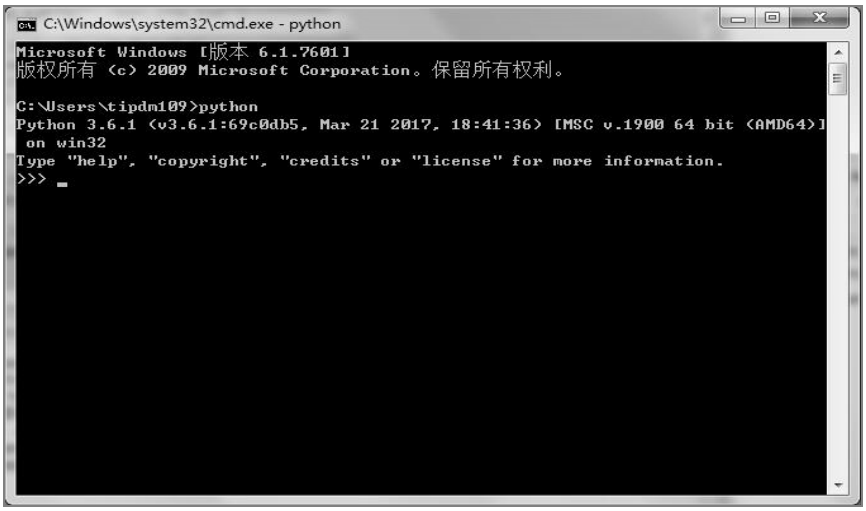


图 1-24　Python 交互式编程环境

2. 带图形界面的 Python Shell——IDLE（Python GUI）

IDLE 是开发 Python 程序的基本集成开发环境，由 Guido van Rossum 亲自编写（至少最初的绝大部分由他编写）。一般 IDLE 适合用来测试，演示一些简单代码的执行效果。

在 Windows 系统下安装好 Python 后，可以在“开始”菜单中找到 IDLE，如图 1-25 所示，选择“IDLE (Python 3.6 64-bit)”选项即可打开环境界面，如图 1-26 所示。

图 1-25　单击“IDLE”按钮

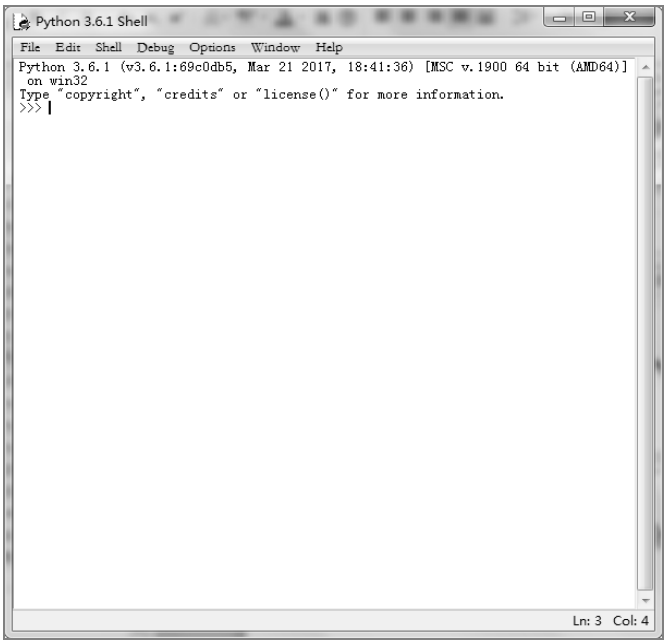


图 1-26　IDLE 界面

3. 命令行版本的 Python Shell——Python 3.6

命令行版本的 Python Shell——Python 3.6 的打开方法和 IDLE 的打开方法是一样的。在 Windows 系统下，在“开始”菜单中找到命令行版本的 Python 3.6（64-bit），如图 1-27 所示，单击即可打开，界面如图 1-28 所示。

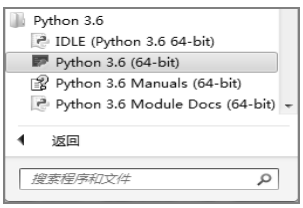


图 1-27　选择 Python 3.6（64-bit）

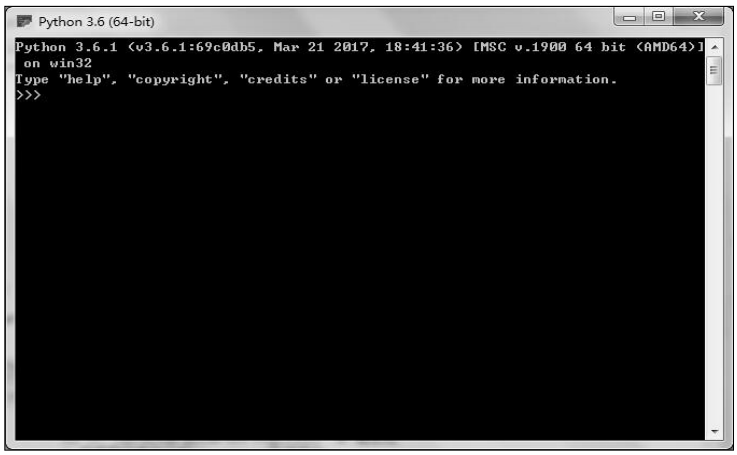


图 1-28　Python 3.6（64-bit）界面

任务 1.3　安装 PyCharm 并创建一个应声虫程序

任务描述

在 Windows 系统下安装 PyCharm，创建一个名为“python”的项目，在此项目下新建一个名为“study.py”的文件。在 study.py 文件里用 4 种方式输出“hello world”。

任务分析

（1）在 Windows 系统下安装 PyCharm。

（2）设置控制台，新建一个项目并命名文件。

（3）直接输出“hello world”。

（4）用逗号隔开“hello”和“world”，输出“hello world”。

（5）采用输入函数 input 输入“hello world”，并赋值给 character 变量，输出 character 变量。

（6）采用输入函数 input 分别输入“hello”“world”，并分别赋值给 x 变量和 y 变量，输出 $x+y$。

1.3.1　了解常用 Python IDE

集成开发环境（Integrated Development Environment，IDE）是一种辅助程序开发人员进行开发工作的应用软件，在开发工具内部就可以辅助编写代码，并编译打包，使其成为可用的程序，有些甚至可以设计图形接口。IDE 是集成了代码编写功能、分析功能、编译功能、调试功能等于一体的开发软件服务套（组），通常包括编程语言编辑器、自动构建工具和调试器。

在 Python 的应用过程中少不了 IDE，这些工具可以帮助开发者加快开发速度，提高效率。在 Python 中常见的 IDE 有 Python 自带的 IDLE、PyCharm、Jupyter Notebook、Spyder 等，简单介绍如下。

（1）IDLE。IDLE 完全由 Python 编写，并使用 Tkinter UI 工具集。尽管 IDLE 不适用于大型项目开发，但它对小型的 Python 代码和 Python 不同特性的实验非常有帮助。

（2）PyCharm。PyCharm 由 JetBrains 公司开发。此公司还以 IntelliJ IDEA 闻名。它们都共享着相同的基础代码，PyCharm 中的大多数特性都能通过免费的 Python 插件带入 IntelliJ 中，本书会着重介绍 PyCharm。

（3）Jupyter Notebook。Jupyter Notebook 是网页版的 Python 编写交互模式，使用过程类似于使用纸和笔，可轻松擦除先前写的代码，并且可以将编写的代码进行保存记录，可用来做笔记以及编写简单代码，相当方便。

（4）Spyder。Spyder 是专门面向科学计算的 Python 交互开发环境，集成了 pyflakes、pylint 和 rope。Spyder 是开源的（免费的），提供了代码补全、语法高亮、类和函数浏览器以及对象检查等功能。

1.3.2　认识 PyCharm

PyCharm 是由 JetBrains 打造的一款 Python IDE，带有一整套可以帮助 Python 开发者提

高工作效率的功能，包括调试、语法高亮、Project 管理、代码跳转、智能提示、自动完成、单元测试及版本控制。

PyCharm 还提供了一些高级功能，用于支持 Django 框架下的专业 Web 开发，同时支持 Google App Engine 和 IronPython。这些功能在先进代码分析程序的支持下，使 PyCharm 成为了 Python 专业开发人员和刚起步人员的有力工具。

1.3.3　使用 PyCharm

1. 安装 PyCharm

PyCharm 可以跨平台使用，分为社区版和专业版，其中社区版是免费的，专业版是付费的。对于初学者来说，两者差距不大。在使用 PyCharm 之前需安装，具体安装步骤如下。

（1）打开 PyCharm 官网（https://www.jetbrains.com/pycharm），如图 1-29 所示，单击“DOWNLOAD NOW”按钮。

图 1-29　PyCharm 官网

（2）选择 Windows 系统的社区版，单击“DOWNLOAD”按钮即可进行下载，如图 1-30 所示。

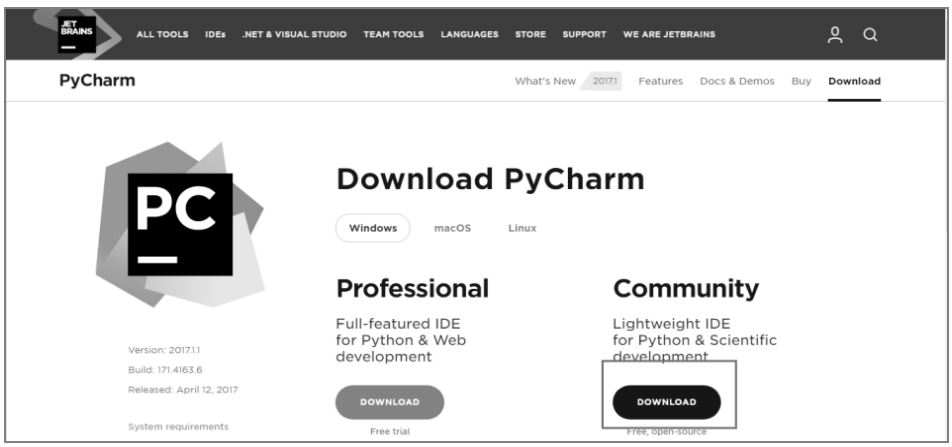


图 1-30　选择社区版并下载

（3）下载完成后，双击安装包打开安装向导，如图 1-31 所示，单击“Next”按钮。

图 1-31　欢迎安装界面

（4）在进入的界面中自定义软件安装路径，建议不要使用中文字符，如图 1-32 所示，单击“Next”按钮。

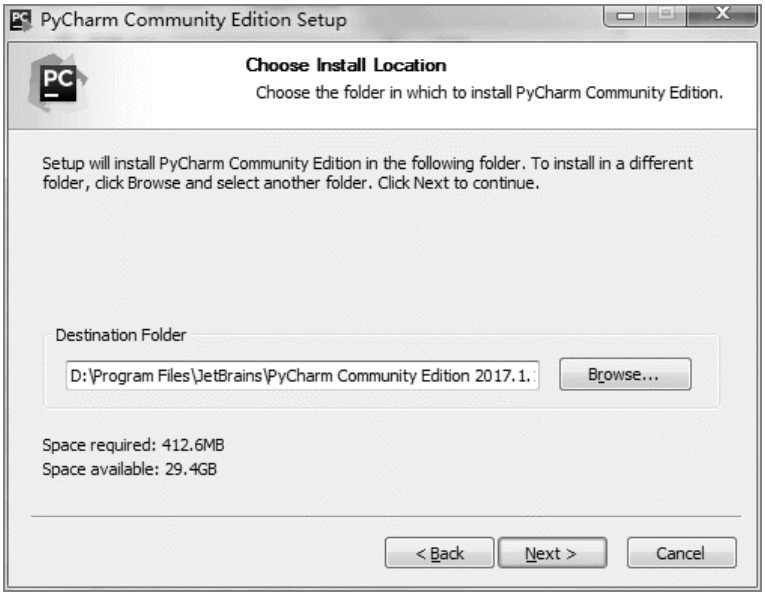


图 1-32　选择安装路径

（5）在进入的界面中根据自己计算机的系统选择位数，创建桌面快捷方式并关联.py 文件，如图 1-33 所示，单击“Next”按钮。

（6）在进入的界面中单击“Install”按钮默认安装。安装完成后单击“Finsh”按钮，如图 1-34 所示。

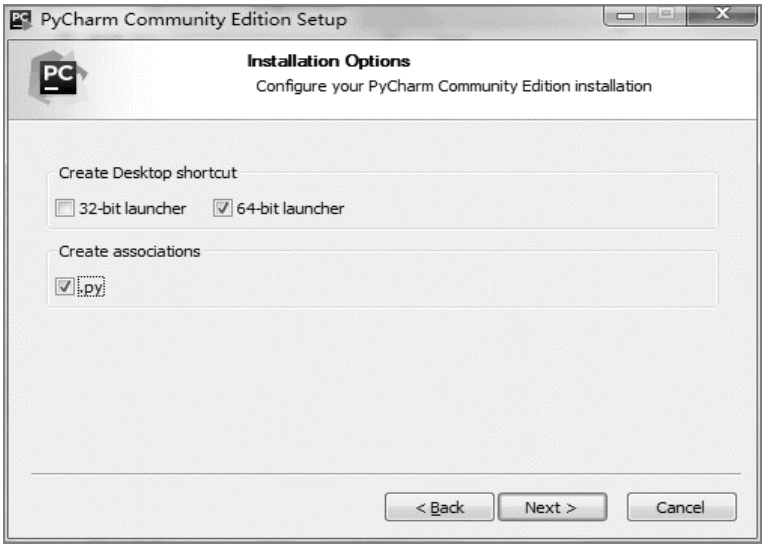


图 1-33　选择位数和文件

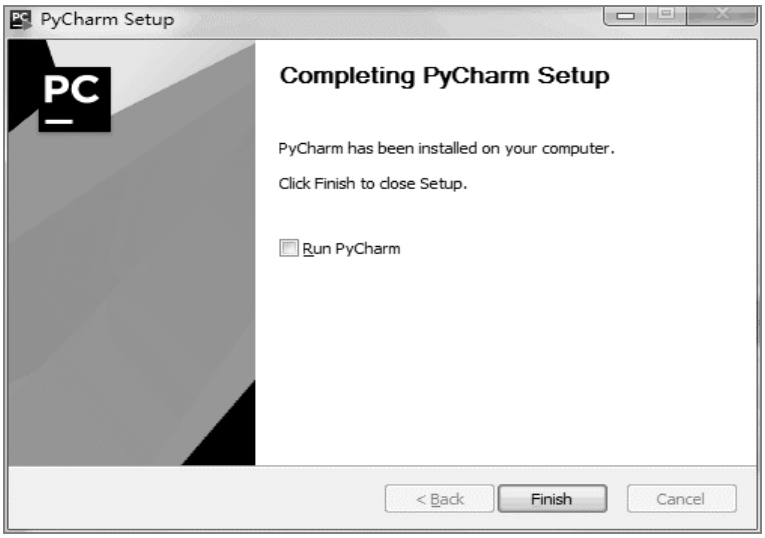


图 1-34　安装完成

（7）双击桌面上的快捷方式，在弹出的对话框中选择不导入开发环境配置文件，如图 1-35 所示，单击“OK”按钮。

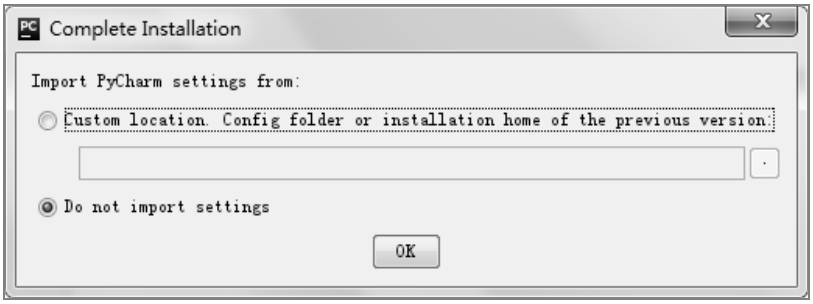


图 1-35　选择不导入文件选项

（8）根据自己的喜好选择 IDE 主题与编辑区主题，本书选择使用 Darcula 主题，如图 1-36 所示。由于更改了主题，所以需要重启 IDE，单击“OK”按钮即可。

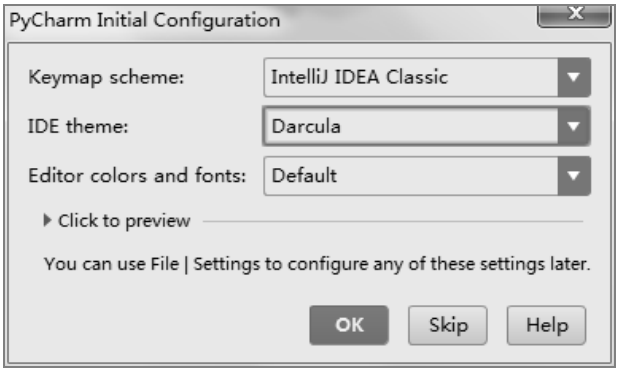


图 1-36　选择主题

（9）重启后，会弹出图 1-37 所示的窗口，选择“Create New Project”选项创建新项目，如图 1-37 所示。

图 1-37　创建新项目

（10）打开“New Project”窗口，自定义项目存储路径，IDE 默认关联 Python 解释器，单击“Create”按钮，如图 1-38 所示。

（11）此时弹出提示信息，选择在启动时不显示提示，如图 1-39 所示，单击“Close”按钮。

这样就进入了 PyCharm 界面，如图 1-40 所示，单击左下角的图标可显示或隐藏功能侧边栏。

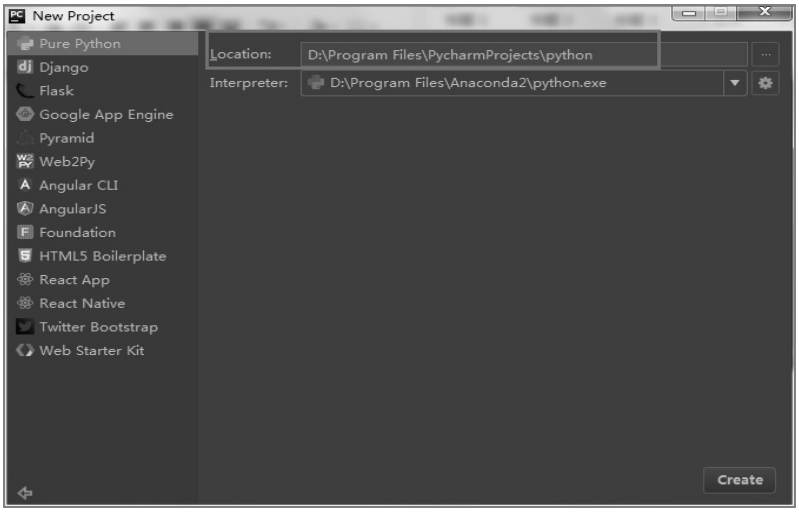


图 1-38　自定义路径

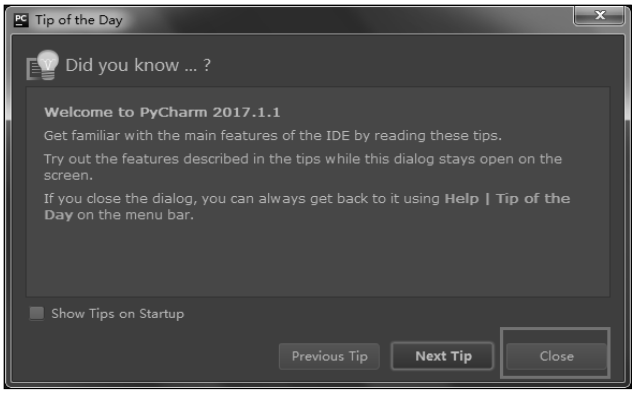


图 1-39　IDE 提示

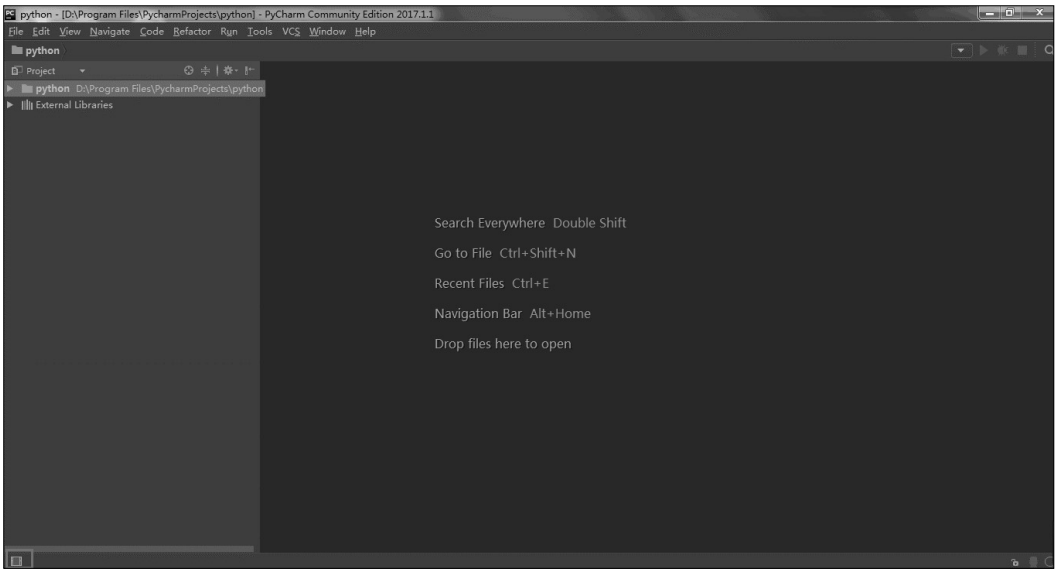


图 1-40　PyCharm 界面

2. 使用 PyCharm

（1）新建好项目（此处项目名为 python）后，还要新建一个.py 文件。右击项目名“python”，选择“New”→“Python File”命令，如图 1-41 所示。

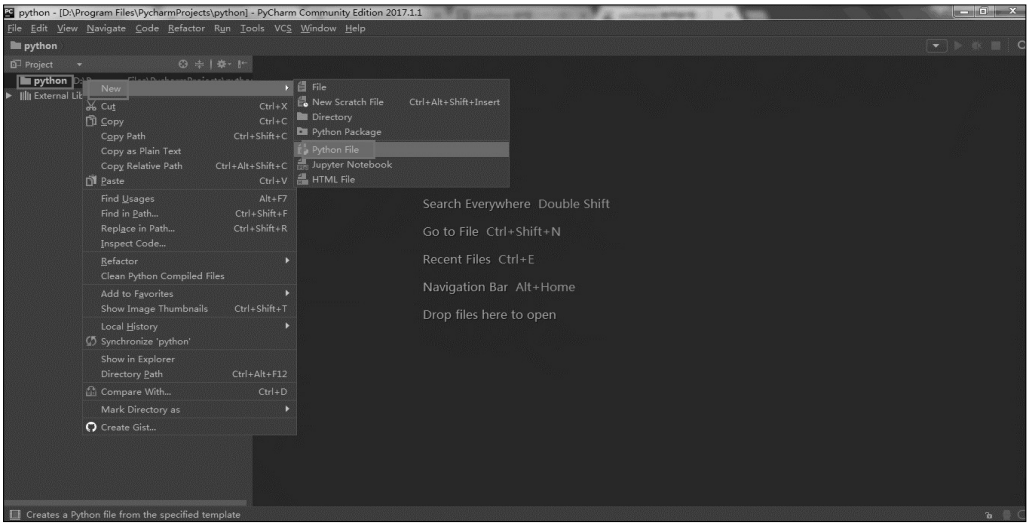


图 1-41　新建文件

（2）在弹出的对话框中输入.py 文件名，如图 1-42 所示。单击“OK”按钮即可打开此脚本文件，如图 1-43 所示。如果是首次安装，则此时运行的符号是灰色的，处于不可触发的状态，需要设置控制台。

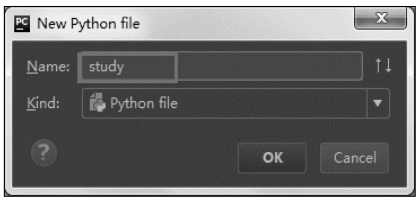


图 1-42　输入文件名

（3）单击运行符号左边的倒三角符号，如图 1-44 所示，进入“Run/Debug Configurations”窗口，单击加号，新建一个配置项，并选择 Python，如图 1-45 所示。

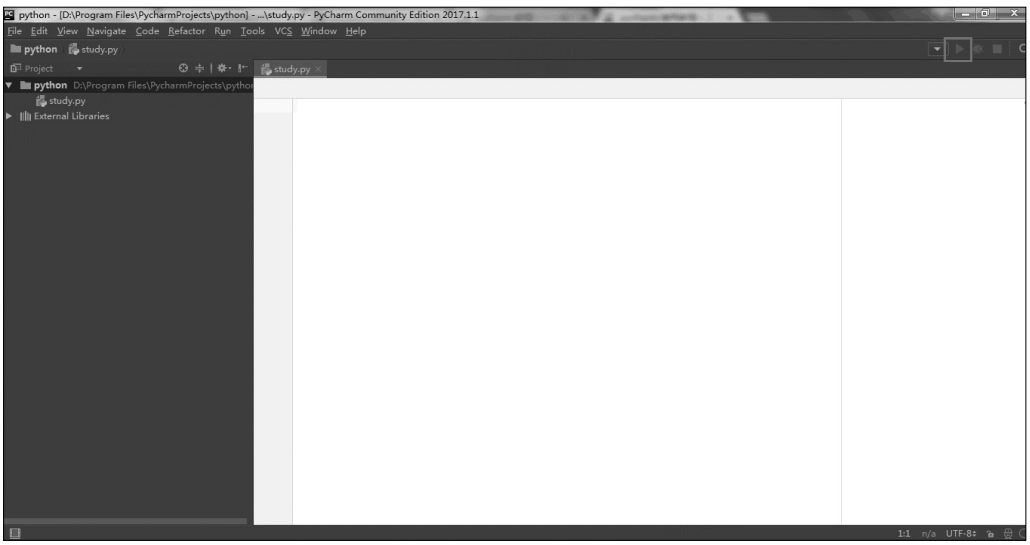


图 1-43　打开脚本文件

图 1-44　单击倒三角符号

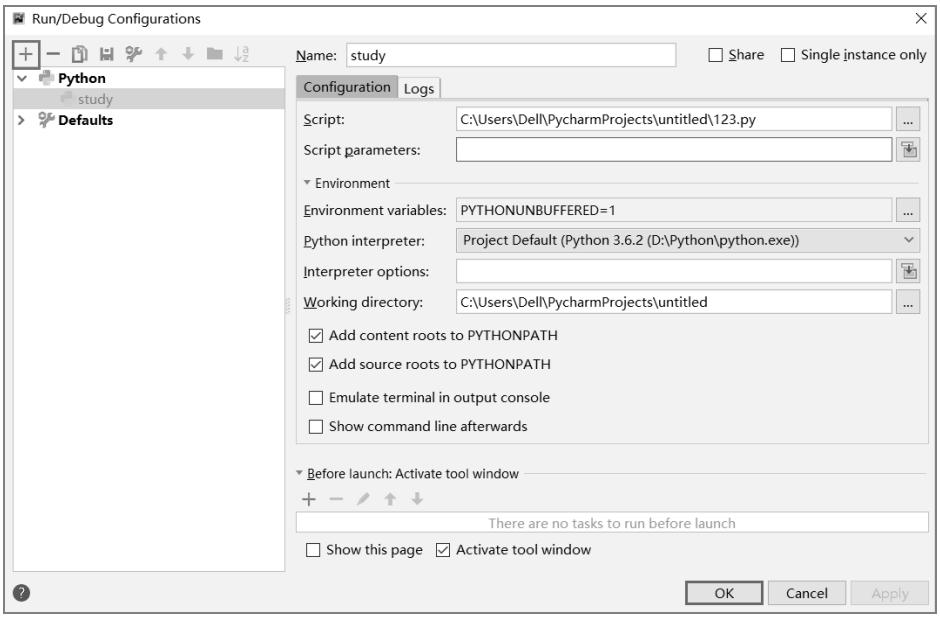


图 1-45　新建配置项

（4）在右侧窗格中的“Name”文本框中输入名称，单击“Script”选项右侧的“浏览”按钮，找到刚刚新建的 study.py 文件，如图 1-46 所示。单击“OK”按钮之后，运行的符号就会变成绿色的，此时就可以正常编程了。

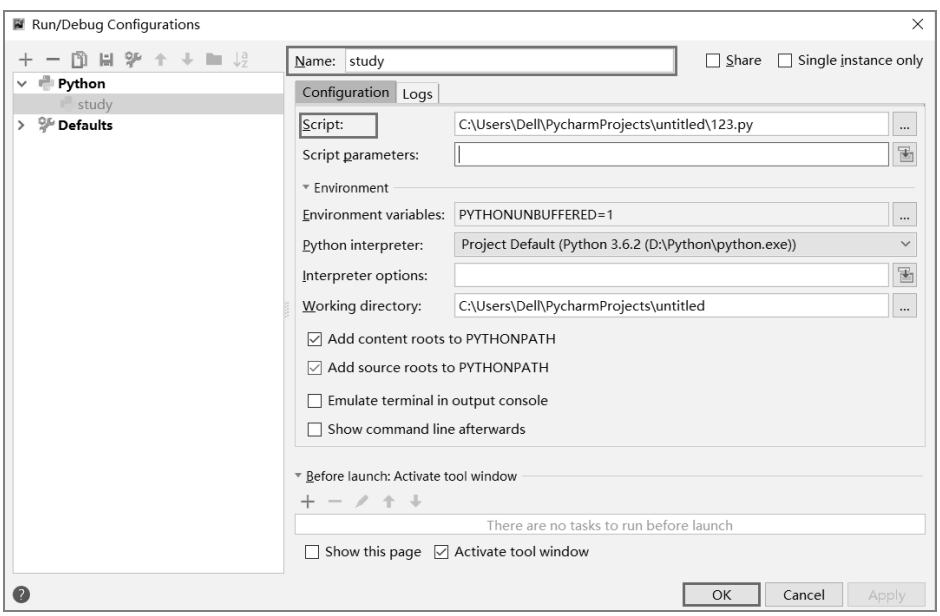


图 1-46　设置 Script 选项

1.3.4 创建应声虫程序

Python 和 PyCharm 安装好之后，就可以开始编写第一个应声虫程序了。Python 和其他高级语言一样，程序的基本构架都会有输出和输入部分。下面简单介绍 Python 的输出和输入应声虫程序。

1. 输出

在 Python 语言中，实现数据输出的方式有两种：一种是使用 print 函数；另一种是直接使用变量名来查看该变量的原始值。

（1）print 函数

print 函数是可以打印输出数据的输出操作，其语法结构如下。

```
print( < expressions >)
```

print 函数语法结构里的< expressions >单词为复数，其含义是表达式可以是多个。

Python 在执行 print 函数时，首先计算 print 函数后边的 expressions 表达式的值，之后将表达式的值打印输出。

如果有多个< expression >，则表达式之间用逗号隔开，语法格式如下。

```
print( < expression >,< expression >,...,< expression >)
```

在新建的.py 文件中输出 print 语句，如代码 1-1 所示，。

代码 1-1　print 函数输出

```
>>> print ('hello world')
hello world
>>>print ('hello', 'world')
hello world
```

可以看到，第 2 条 print 语句用逗号连接两个字符串，在输出的时候，字母“o”和“w”中间有空格。

（2）直接使用变量名来查看该变量的原始值

在交互式环境中，为了方便，可以直接使用变量名来查看该变量的原始值，以达到输出的目的，如代码 1-2 所示。

代码 1-2　先赋值，再输出

```
>>> character = "hello world"
>>> character
'hello world'
```

将“hello world”赋值给 character，然后直接输出 character，即可查看该 character 的原始值。直接在交互式环境中运行“hello world”语句，也可以实现输出，如代码 1-3 所示。

代码 1-3　直接输出

```
>>> "hello world"
'hello world'
```

2. 输入

在 Python 中可以通过 input 函数从键盘输入数据，其语法结构如下。

```
input(< prompt >)
```

input 函数的形参 prompt 是一个字符串，用于提示用户输入数据。input 函数的返回值是字符串型的，如代码 1-4 所示。

代码 1-4　input 输入

```
>>>character = input('input your character:')
>>>print(character)
input your character:
```

第 1 行语句使用 input 函数输入数据。用户输入数据后，input 函数会把数据传给等号左边的 character 变量来保存。第 2 行调用 print 函数打印 character 变量的值，所以执行第 2 行语句后会打印出字符串“input your character:”，以此作为新的提示符，输入“hello world”后按回车键，即可出现图 1-47 所示的结果，完整地输出“hello world”。

图 1-47　输出结果

若想依次打印出“first:”和“second:”，可以用字符串拼接的方式，如代码 1-5 所示。

代码 1-5　input 函数输入

```
>>>x = input("first: ")
>>>y = input("second: ")
>>>print(x + y)
```

在执行第 3 行语句后，会依次打印出“first:”和“second:”，依次输入“hello”和“world”，即可出现图 1-48 所示的结果，完整地输出“hello world”。

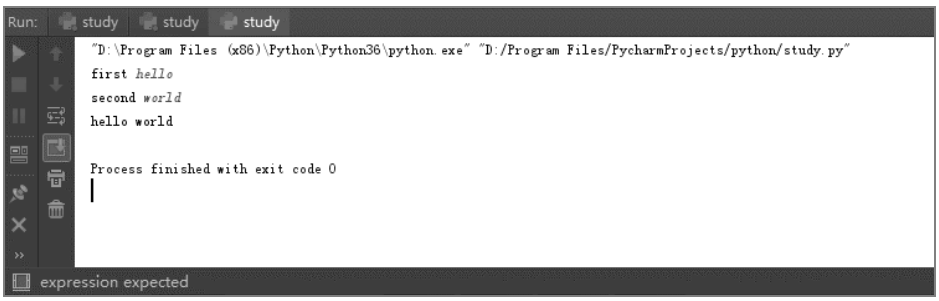


图 1-48　执行结果

1.3.5 任务实现

根据任务分析，本任务的具体实现过程可以参考如下操作。

（1）按照 1.3.3 小节中的安装方法，在 Windows 系统下安装 PyCharm。

（2）参照 1.3.3 小节中 PyCharm 的使用方法，新建一个名为“python”的项目，再新建一个名为“study”的.py 文件，并设置控制台。

（3）直接输出“hello world”。

（4）用逗号隔开“hello”和“world”，输出“hello world”。

（5）采用输入函数 input 输入“hello world”，并赋值给 character 变量，再用 print 函数输出 character 变量。

（6）采用输入函数 input 分别输入“hello”“world”，并分别赋值给 x 变量和 y 变量，用 print 函数输出 x＋y。

参考代码如任务实现 1-1 所示。

任务实现 1-1

```
# -*-coding:utf-8-*-

print ('hello world')
print ('hello', 'world')
character = input('input your character:')
print(character)
x = input("first: ")
y = input("second: ")
print(x + y)
```

小结

本章介绍了 Python 的历史和 Python 的特性，同时还介绍了 Python 的安装方法，并对比了 4 种 Python IDE，着重介绍了 PyCharm 的使用。下面对本章内容做一个小结。

（1）Python 的关键性特性：简单易学，免费、开源，有广泛的标准库，有互动模式，可移植，可扩展，可嵌入。

（2）安装 PyCharm 之后新建项目和.py 文件，在“Run/Debug Configurations”窗口设置控制台。

（3）应声虫程序的输入/输出有 4 种方法：直接输出；用逗号隔开多个表达式，然后输出；直接采用输入函数 input 赋值，然后输出；采用输入函数 input 分别赋值，然后输出。

实训 输入/输出

1. 训练要点

（1）掌握输出的多种方法。

（2）掌握输入的多种方法。

2. 需求说明

（1）至少使用两种方式运行 PyCharm，输出“my name is python”。

（2）使用 PyCharm 输入“my name is”和“python”，且能输出“my name is python”。

3. 实现思路及步骤

（1）打开 PyCharm，新建一个名为“python”的项目，在此项目下新建一个名为“training_output”的.py 文件。

（2）在 training_output.py 文件里执行输出语句。

（3）在“python”项目中新建一个名为“training_input”的.py 文件，并执行输入/输出语句。

课后习题

1. 选择题

（1）以下（　　）不是 Python 的特性。

A．收费使用　　B．跨平台　　C．可拓展　　D．可嵌入

（2）Python 程序的文件扩展名是（　　）。

A．.python　　B．.p　　C．.py　　D．.pyth

（3）以下（　　）不是 Python IDE。

A．PyCharm　　B．Spyder

C．Rstudio　　D．Jupyter Notebook

2. 操作题

（1）至少使用两种方式运行 PyCharm，输出“welcome to this world”。

（2）使用 PyCharm 输入“welcome to this world”，并输出“welcome to this world”。

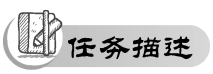
第 2 章 Python 基础知识

本章将介绍 Python 的基础内容，读者可以学习到丰富的 Python 知识。本章首先介绍 Python 的固定语法，然后比较全面地介绍 Python 基础变量的特点和使用方法，以及两种基础数据类型的操作、运算等。

学习目标

（1）掌握 Python 的固定语法。
（2）了解 Python 基础变量类型。
（3）掌握 Python 的数值型变量。
（4）掌握 Python 的字符型变量。
（5）掌握 Python 的常用操作运算符。

任务 2.1 掌握 Python 的固定语法



任务描述

Python 是一门简单又优雅的语言，在使用之前，读者需要了解并掌握它的基础语法，这样有助于代码的学习和运用，并有利于保持一个良好的编程风格。读者需要认识计算机语言并学习 Python 语言的编程规则，掌握 Python 作为计算机语言的固定语法要求。

任务分析

（1）认识计算机语言。
（2）掌握 Python 的编码，以及单行代码注释和多行代码注释。
（3）掌握 Python 中使用多行语句的方法和缩进代码的格式。
（4）了解标识符与保留字符的命名。
（5）掌握调试 Python 代码的方法。

2.1.1 认识计算机语言

众所周知，人与人之间可以通过语言进行交流沟通，那么人与计算机怎么进行交流呢？这就需要将人类语言转化成计算机能够理解的语言，这种计算机能够理解的语言即为计算机语言。计算机语言的种类很多，总体可以分为三大类，分别是高级语言、汇编语言、机器语言。

机器语言是指一台计算机的全部指令集合，是由“0”和“1”组成的指令序列。汇编

语言在机器语言的基础上进行了改进，以英文单词代替 0 和 1，例如 Add 代表相加，Mov 代表传递数据等。汇编语言实际上就是机器语言的一个记号。高级语言并不是特指某一种语言，它泛指很多编程语言，比如 Python、C、C++、Java 等。大多数编程者都会选择高级语言。相对于汇编语言，高级语言将许多相关的机器指令合成为单条指令，并且去掉了与具体操作有关但与完成工作无关的细节，例如使用堆栈、寄存器等，极大地简化了程序中的指令。高级语言源程序可以通过解释和编译两种方式执行，一般使用后一种。由于 Python 省略了很多编译细节，所以更容易上手。

Python 是一种结合解释性、编译性、互动性的面向对象的高层次脚本语言，也是一种高级语言。由于 Python 易学习，并且具有广泛而丰富的标准库及第三方库的特点，因此它可以和其他语言很好地融合，所以也称为“胶水语言”。Python 的设计目标之一是让代码具有高度的可阅读性，使用它设计时，尽量使用其他语言经常使用的标点符号和英文单词，让代码看起来整洁美观，而不像其他静态语言（如 C、Pascal 等）那样需要重复书写声明语句，避免了经常出现特殊情况和意外。

2.1.2　声明 Python 编码

Python 3 安装后，系统默认其源码文件为 UTF-8 编码。在此编码下，全世界大多数语言的字符都可以同时在字符串和注释中得到准确编译。

大多数情况下，通过编辑器编写的 Python 代码默认保存 UTF-8 编码脚本文件，系统通过 Python 执行该文件时就不会出错。但是如果编辑器不支持 UTF-8 编码的文件，或者团队合作时有人使用了其他编码格式，Python 3 就无法自动识别脚本文件，就会造成程序执行错误，这时候对 Python 脚本文件进行编码声明就显得尤为重要了。比如，GBK 脚本文件在没有编码声明时执行就会出错，经编码声明后，脚本文件就可以正常执行。

为源文件指定特定的字符编码，需要在文件的首行或者第二行插入一行特殊的注释行，通常使用的编码声明格式如下。

```
#-*-coding:utf-8-*-
```

通过上述声明，源文件中的所有字符都会被当成 coding 指代的 UTF-8 编码对待。当然这不是唯一的声明格式，上述格式只是普遍使用的一种形式。其他形式的声明，如“#coding:utf-8”和“#coding=utf-8”，也都是可以的。

在编写 Python 脚本时，除了要声明编码以外，还需要注意路径声明。路径声明的格式如下。

```
#!e:/Python/Python36
```

上述语句声明的路径为 Python 的安装路径。路径声明的目的是告诉 OS 调用“e:/Python/Python36”目录下的 Python 解释器执行文件，一般放在脚本首行。

2.1.3　加入代码注释

注释对于机器编程来说是不可少的，即使是简短的几行 Python 代码，如果使用了一些生僻的方法，那么程序开发人员也需要花一定时间才能弄明白。更何况，实际应用中常常要面临成千上万行晦涩难懂的代码，如果对代码注释得不够彻底，时间久了恐怕连程序开发人员自己也会弄不清代码的含义。下面将介绍 Python 注释行的用法。

1. 单行注释

单行注释通常以井号（#）开头，如代码 2-1 所示。

代码 2-1　单行注释

```
# 这是一个单独成行的注释
>>>print("Hello,World!")                    # 这是一个在代码后面的注释
```

注释行是不会被机器编译的，在这里需要提一下的是前文介绍过的编码声明也是以#号开头，但不属于注释行，而且编码声明需要放在首行或第二行，否则不会被机器解释。

2. 多行注释

在实际应用中常常会有多行注释的需求，同样也可以使用#号进行注释，只需在每一行前加#号就行。

（1）#号注释

使用#号进行注释，如代码 2-2 所示。

代码 2-2　多行注释

```
# 这是一个使用#号的多行注释
# 这是一个使用#号的多行注释
# 这是一个使用#号的多行注释
>>>print("Hello,World!")
```

显然，这种方法看起来有些笨拙。Python 中对多行注释还有另一种更加方便优雅的方式，就是使用 3 个单引号或者 3 个双引号将注释内容括起来，达到多行或者整段内容注释的效果。

（2）单引号注释

使用单引号注释，如代码 2-3 所示。

代码 2-3　单引号注释

```
'''
该多行注释使用的是 3 个单引号
该多行注释使用的是 3 个单引号
该多行注释使用的是 3 个单引号
'''
>>>print("Hello,World!")
```

（3）双引号注释

使用双引号注释，如代码 2-4 所示。

代码 2-4　双引号注释

```
"""
该多行注释使用的是 3 个双引号
该多行注释使用的是 3 个双引号
该多行注释使用的是 3 个双引号
```

```
"""
>>>print("Hello,World!")
```

在使用引号进行多行注释的时候，需要保证前后使用的引号类型保持一致。前面使用单引号，后面使用双引号，或者前面使用双引号，后面使用单引号，都是不被允许的。

2.1.4　使用多行语句

多行语句可以有两种理解：一条语句多行；一行多条语句。

一条语句多行的情况一般是语句太长，一行写完一条语句会显得很不美观，使用反斜杠（\）可以实现一条长语句的换行，也不会被机器识别成多条语句，如代码 2-5 所示。

代码 2-5　长语句换行

```
>>>total = applePrice +\
...     bananaPrice + \
...     pearPrice
```

但是 Python 中，[ ]、{ }、( )里面的多行语句在换行时是不需要使用反斜杠（\）的，如代码 2-6 所示。

代码 2-6　使用逗号换行

```
>>>total = [applePrice ,
...     bananaPrice ,
...     pearPrice]
```

一行多条语句，通常在短语句中应用得比较广泛。使用分号（;）可对多条短语句实现隔离，从而在同一行实现多条语句，如代码 2-7 所示。

代码 2-7　分号实现隔离

```
>>>applePrice = 8;  bananaPrice = 3.5;  pearPrice = 5
```

2.1.5　缩进代码

Python 最具特色的就是以缩进的方式来标识代码块，不再需要使用大括号（{}），代码看起来会更加简洁明朗。

同一个代码块的语句必须保证相同的缩进空格数，否则将会出错。至于缩进的空格数，Python 并没有硬性要求，只需保证空格数一致即可。

正确缩进的 Python 代码块如代码 2-8 所示。

代码 2-8　正确缩进示例

```
>>>if True :
...    print('我的行缩进空格数相同')
>>else :
...    print('我的缩进空格数不同')
```

错误示范如代码 2-9 所示，最后一行的语句缩进空格数与其他行不一致，会导致代码运行出错。

代码 2-9　错误缩进示例

```
>>>if True :
...       print('我的行缩进空格数相同'')
>>>else :
...       print('错误示范'')
...   print('我的缩进空格数不同')
```

此外，在交互式输入复合语句时，必须在最后添加一个空行来标识结束。当代码太复杂时，解释器将难以判断代码块从何结束，而且以空行标识结束也更便于自己进行查阅和理解。

2.1.6　命名标识符与保留字符

标识符在机器语言中是一个被允许作为名字的有效字符串。Python 中的标识符主要用在变量、函数、类、模块、对象等的命名中。

Python 中对标识符有如下规定。

（1）标识符可以由字母、数字和下划线组成。

（2）标识符不能以数字开头。以下划线开头的标识符具有特殊的意义，使用时需要特别注意。

① 以单下划线开头（如_foo）的标识符代表不能直接访问的类属性，需通过类提供的接口进行访问，不能用“from xxx import *”导入。

② 以双下划线开头（如__foo）的标识符代表类的私有成员。

③ 以双下划线开头和结尾（如__foo__）的标识符是 Python 特殊方法专用的标识，如__init__()代表类的构造函数。

（3）标识符字母区分大小写，例如 Abc 与 abc 是两个标识符。

（4）标识符禁止使用 Python 中的保留字。要查看某字符串是否为保留字，可以使用 iskeyword 函数。此外，使用 kwlist 函数可以查看所有保留字，如代码 2-10 所示。

代码 2-10　查看保留字

```
>>>import keyword
>>>keyword.iskeyword("and")          # 查看 and 是否为保留字
True
>>>keyword.kwlist                    # 查看 Python 中的所有保留字
['False','None','True','and','as','assert','break','class','continue','def',
'del','elif','else','except','finally','for','from','global','if','import','
in','is','lambda','nonlocal','not','or','pass','raise','return','try','while
','with','yield']
```

2.1.7　调试 Python 代码

进入程序编程学习前，先来欣赏一个程序员的小笑话：“诸葛亮是一个优秀的程序员，每一个锦囊都是应对不同 case 而编写的！但是再优秀的程序员也敌不过更优秀的 Bug！于是六出祁山、七进中原、鞠躬尽瘁、死而后已的诸葛亮只因为有一个错误的 Case——马谡，

使得整个结构就被 break 了!”。这里可以发现：“千里之堤毁于蚁穴”不是空穴来风，一个小小的 Bug 就会让整个程序运行失败。

程序一次写完并能正确运行的概率非常小，一般会有各种各样的 Bug 需要修正。有的 Bug 很简单，看看错误信息就知道如何解决；而有的 Bug 很复杂，需要判断出错时哪些变量的值是正确的、哪些变量的值是错误的。因此，程序开发人员需要一整套调试程序的手段来修复 Bug。

程序调试就是将写好的程序投入实际运行前，用手工或编译程序等方法进行测试，进而修正语法错误和逻辑错误的过程。这是保证计算机信息系统正确性的必不可少的步骤。写完计算机程序，必须送入计算机中进行测试，然后根据测试时所发现的错误进一步诊断，找出原因和具体的位置并进行修正。

Python 代码可以使用 pdb（Python 自带的包）调试、Python IDE 调试（如 PyCharm）、日志功能等进行调试。接下来介绍对于一些简单的错误怎么调试修改，如代码 2-11 所示。

代码 2-11　语法错误示例

```
>>> print "Hello,World!"            # 缺少括号
SyntaxError: invalid syntax
>>> print(‘Hello,World!’)           # 引号为中文引号
SyntaxError: invalid character in identifier
>>> print（‘Hello,World!’）         # 括号为中文括号
SyntaxError: invalid character in identifier
```

代码 2-11 中的错误都是语法错误，第一行代码在 Python 2 中是正确运行的，但是在 Python 3 中并不能正确运行；后面的两行代码均是因为使用了中文格式符号导致了出错，编写代码一般使用英文输入。当然这只是简单的打印出来并查看错误，还有其他很多调试代码的方法，读者可以参考其他相关内容进行了解。

任务 2.2　创建字符串变量并提取里面的数值

任务描述

Python 基础变量主要有字符型和数值型两种，数值型变量又可分为整数、浮点数、布尔值。创建变量时不需要声明数据类型，Python 能够自动识别数据类型。这里的任务将创建字符串变量“Apple's unit price is 9 yuan.”，并把里面的数值提取出来，转成整型（int）数据。

任务分析

（1）创建一个字符串变量“Apple's unit price is 9 yuan.”。

（2）提取出里面的数字 9 并赋值给新的变量。

（3）查看新变量的数据类型。

（4）将提取的数字 9 转成整型（int）。

（5）确认数据类型是否转换成功。

2.2.1 了解 Python 变量

在 Python 中，变量不需要提前声明，创建时直接对其赋值即可，变量类型由赋给变量的值决定。值得注意的是，一旦创建了一个变量，就需要给该变量赋值。

有一种“平民”的说法是，变量好比一个标签，指向内存空间的一块特定的地址。创建一个变量时，在机器的内存中，系统会自动给该变量分配一块内存，用于存放变量值，如图 2-1 所示。

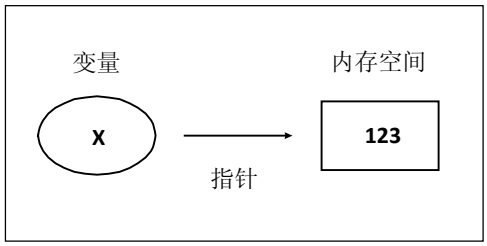


图 2-1 变量存储示意图

通过 id 函数可以具体查看创建变量和变量重新赋值时内存空间的变化过程，如代码 2-12 所示。

代码 2-12 内存空间的变化

```
>>>x = 4
>>>id(x)                          # 查看变量 x 指向的内存地址
30834096L
>>>y = x                          # 将变量 x 重新赋给另一个新变量 y
>>>id(y)
30834096L
>>>x = 2                          # 对变量 x 重新赋值
>>>x,y                            # 同时输出变量 x 和变量 y 的值
(2,4)
>>>id(x)
30834144L
>>>id(y)
30834096L
```

从代码 2-12 中可以直观地看出，一个变量在初次赋值时就会获得一块内存空间来存放变量值。当令变量 y 等于变量 x 时，其实是一种内存地址的传递，变量 y 获得的是存储变量 x 值的内存地址，所以当变量 x 改变时，变量 y 并不会发生改变。此外还可以看出，变量 x 的值改变时，系统会重新分配另一块内存空间存放新的变量值。

要创建一个变量，首先需要一个变量名和变量值（数据），然后通过赋值语句将值赋给变量。

1. 变量名

变量的命名须严格遵守标识符的规则，Python 中还有一类非保留字的特殊字符串（如

内置函数名），这些字符串具有某种特殊功能，虽然用于变量名时不会出错，但会造成相应的功能丧失。如 len 函数可以用来返回字符串长度，但是一旦用来作为变量名，其就失去了返回字符串长度的功能。因此，在取变量名时，不仅要避免 Python 中的保留字，还要避开具有特殊作用的非保留字，以避免发生一些不必要的错误，如代码 2-13 所示。

代码 2-13　变量名注意事项

```
>>>import keyword                                    # 加载 keyword 库
>>>keyword.iskeyword("and")                         # 判断 and 是否为保留字
TRUE
>>>and = "我是保留字"                                # 以保留字作为变量名
  File "<stdin>", line 1
    and = '我是保留字'
      ^
SyntaxError: invalid syntax
>>>strExample = "我是一个字符串"                     # 创建一个字符串变量
>>>len(strExample)                                  # 使用 len 函数查看字符串长度
7
>>>len = "特殊字符串命名"                            # 使用 len 作为变量名
>>>len
特殊字符串命名
>>>len(strExample)                                  # len 函数查看字符串长度出错
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object is not callable
```

如果在一段代码中有大量变量名，而且这些变量没有错，只是取名都很随意，风格不一，这样在解读代码时就会出现一些混淆。接下来介绍几种命名法。

（1）大驼峰（upper camel case）

所有单词的首字母都是大写，例如“MyName,YourFamily”……

大驼峰命名法一般用于类的命名。

（2）小驼峰（lower camel case）

第一个单词的首字母为小写字母，其余单词的首字母都采用大写字母，例如“myName”“yourFamily”等。

小驼峰命名法用在函数名和变量名中的情况比较多。

（3）下划线（_）分隔

首个单词用小写字母，中间用下划线（_）分隔后，单词的首字母为大写字母，例如“my_Name,your_Family”等。

关于要使用哪种方法对变量命名，并没有统一的说法，重要的是一旦选择好了一种命名方式，在后续的程序编写过程中一定要保持风格一致。

2. 变量值

变量值就是赋给变量的数据，Python 中有 6 个标准的数据类型，分别为数字（Number）、

字符串（String）、列表（List）、元组（Tuple）、字典（Dictionary）、集合（Sets）。其中，列表、元组、字典、集合属于复合数据类型。

3. 变量赋值

最简单的变量赋值就是把一个变量值赋给一个变量名，只需要用等号（=）就可以实现。同时，Python 还可以将一个值同时赋给多个变量，如代码 2-14 所示。

代码 2-14　变量赋值

```
>>>a = b = c = 1                          # 一个值赋给多个变量
>>>a
1
>>>b
1
>>>c
1
```

代码 2-14 展示了将数字 1 同时赋给了变量 a、b、c。如果要将数字 1、2 和字符串“abc”分别赋值给变量 a、b、c，就需使用逗号（,）隔开，如代码 2-15 所示。

代码 2-15　多个变量同时赋值

```
>>>a,b,c =1,2,"abc"                       # 多个变量同时赋值
>>>a
1
>>>b
2
>>>c
'abc'
```

2.2.2　相互转化数值型变量

Python 3 支持的数值型数据类型有 int、float、bool、complex，Python 3 中的整数类型 int 表示长整型，没有了 Python 2 中的 long，如表 2-1 所示。

表 2-1　数值类型

数值型数据类型	中文解释	示　例
int	整数类型	10；100；1000
float	浮点数	1.0；0.11；1e-12
bool	布尔型	True；False
complex	复数	1+1j；0.123j；1+0j

int 类型指整数数值，float 类型指既有整数又有小数部分的数据类型，这些都是比较好理解的。bool 类型只有 True（真）和 False（假）两种取值，因为 bool 继承了 int 类型，即在这两种类型中 True 可以等价于数值 1，False 可以等价于数值 0，并且可以直接使用 bool

值进行数学运算。complex 类型由实数部分和虚数部分构成，Python 中的结构形式，如 real ＋imag（J / j 后缀），实数和虚数部分都是浮点数。

在 Python 中可以实现数值型数据类型的转换，使用的内置函数有 int、float、bool、complex。int 函数转换如代码 2-16 所示。

代码 2-16　int 函数转换演示

```
>>>int(1.56) ; int(0.156) ; int(-1.56) ;  int()          # 浮点数转整型
1
0
-1
0
>>>int(True) ; int(False)                                 # 布尔型转整型
1
0
>>>int(1+23j)                                            # 复数转整型
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't convert complex to int
```

代码 2-16 所示的结果都很简单，首先看浮点数转整数的运行结果，可以看出，浮点数转换成整数的过程中，只是简单地将小数部分剔除，保留整数部分，int 空的结果为 0；布尔型转整型时，bool 值 True 被转成整数 1，False 被转成整数 0；复数没办法转换成整型。

bool 函数转换如代码 2-17 所示。

代码 2-17　bool 函数转换演示

```
>>>bool(1) ;   bool(2) ;   bool(0)                       # 整型转布尔型
True
True
False
>>>bool(1.0) ;   bool(2.3) ;   bool(0.0)                 # 浮点数转布尔型
True
True
False
>>>bool(1+23j) ;   bool(23j)                             # 复数转布尔型
True
True
>>>bool();bool("") ;   bool([]);   bool(());   bool({}) # 各种类型的空值转布尔型
False
False
False
False
False
```

从整数、浮点数、复数转布尔型的结果可以总结出一个规律：非 0 数值转布尔型都为 True，数值 0 转布尔型为 False。此外，用 bool 函数分别对空、空字符、空列表、空元组、空字典（或者集合）进行转换时结果都为空，如果是非空，结果是 True（除去非数值 0 的情况）。

2.2.3 字符型数据的创建与基本操作

相比于数值型数据，字符型数据可以理解成是一种文本，在语言领域的应用更加广泛。Python 提供了几种方式去表达字符串，分别是使用单引号（'）、双引号（"）和三引号（'''或者"""），下面介绍这几种方法在 Python 中的实现。

1. 标识字符串

（1）单引号（'）

单引号标识字符串的方法是将字符串用单引号括起来。标准 Python 库允许字符串中包含字母、数字及各种符号。Python 3 的默认编码为 UTF-8，意味着可以在字符串中任意使用中文也不会出错，如代码 2-18 所示。

代码 2-18 单引号标识字符串

```
>>>'This is a sentence.'                    # 单引号标识字符串
'This is a sentence.'
```

（2）双引号（"）

双引号在字符串中的使用与单引号的用法完全相同，需要注意的是，单引号和双引号不能混用，如代码 2-19 所示。

代码 2-19 双引号标识字符串

```
>>>"This is a sentence."                    # 双引号标识字符串
'This is a sentence.'
```

（3）三引号（'''或者"""）

三引号相比于单引号或者双引号，自身有一个比较特殊的功能，它能够标识一个多行的字符串，如一段话的换行、缩进等格式都会被原封不动地保留。三引号是格式化记录一段话的好帮手，但前后引号要保持一致，不要混用，如代码 2-20 所示。

代码 2-20 三引号标识字符串

```
>>>paragraph = '''\                         # 3 个单引号标识的一段字符串
...This is the first sentence.
...   This is the second sentence.
...   This is the third sentence.'''
>>>print(paragraph)
This is the first sentence.
   This is the second sentence.
   This is the third sentence.
```

代码 2-20 展示了 3 个单引号标识的一段字符串，通过 print 函数打印，可以清楚地看

出句子的换行和段落缩进等细节都保持了原状。另外，3 个双引号的用法一样，读者可以动手实践。细心的读者可能会发现，代码 2-20 的命令行中有反斜杠（\），它表示字符串在下一行继续，而不是开始一个新行。

反斜杠（\）不仅可以在字符串中担当特殊换行的角色，还可以是字符串中的转义符。

2. 字符转义

举个简单的例子，用单引号标识一个字符串的时候，如果该字符串中又含有一个单引号，比如“What's happened”，Python 将不能辨识这段字符串从何处开始，又在何处结束。此时需要用到转义符，即前文提到的反斜杠（\），使单引号只是纯粹的单引号，不具备任何其他作用，如代码 2-21 所示。

代码 2-21　单引号转义

```
>>>'What's happened'                    # 单引号标识的字符串中含有单引号
  File "<stdin>", line 1
    'What's happened'
         ^
SyntaxError: invalid syntax
>>>'What\'s happened'                   # 反斜杠（\）转义单引号
"What's happened"
```

比较特殊的是，用双引号标识一个包含单引号的字符串时不需要转义符，但是如果其中包含一个双引号，则需要转义。另外，反斜杠可以用来转义其本身，如代码 2-22 所示。

代码 2-22　双引号与反斜杠转义

```
>>>"What's happened"                    # 双引号标识含有单引号的字符串
"What's happened"
>>>" Double quotes(\")"                 # 双引号标识的字符串里面的双引号需要转义
' Double quotes(")'
>>>print('Backslash(\\)')               # 转义反斜杠
Backslash(\)
```

此外，Python 中还可以通过给字符串加上一个前缀 r 或者 R 来指定原始字符串，如代码 2-23 所示。

代码 2-23　指定原始字符串

```
>>>print('D:\name\python')              # 以反斜杠开头的特殊字符
D:
ame\python
>>>print(r'D:\name\python')             # 用 r(或者 R)指定原始字符串
D:\name\python
```

掌握上述介绍的方法，读者基本就可以自由地创建一个字符串了。

3. 字符串索引

Python 对于字符串的操作还是比较灵活的，包括字符提取、字符串切片、拼接等，但

在介绍字符串操作之前，需要先掌握字符串索引的概念。

字符串索引分为正索引和负索引，通常说的索引就是指正索引。如图 2-2 所示，在 Python 中，索引是从 0 开始的，也就是第一个字母的索引是 0，第二个索引是 1，以此类推。很明显，正索引是从左到右去标记字母的；负索引从右到左去标记字母，然后加上一个负号（–）。负索引的第一个值是-1，而不是-0，如果负索引的第一个值是 0，那么就会导致 0 索引指向两个值，这种情况是不允许的。

字符串	P	Y	T	H	O	N
索引	0	1	2	3	4	5
负索引	-6	-5	-4	-3	-2	-1

图 2-2　字符串索引

4．字符串基本操作

下面介绍提取指定位置的字符、字符串切片和字符串拼接的操作。

（1）提取指定位置的字符

Python 中只需要在变量后面使用方括号（[ ]）将需要提取的字符索引括起来，就可以提取指定位置的字符，如代码 2-24 所示。

代码 2-24　提取指定位置的字符

```
>>>word = 'Python'
>>>word[1]                              # 提取第二个字符
'y'
>>>word[0]                              # 提取第一个字符
'P'
>>>word[-1]                             # 提取最后一个字符
'n'
```

（2）字符串切片

字符串切片就是截取字符串的片段，形成子字符串。字符串切片的方式形如 s[i:j]，s 代表字符串，i 表示截取字符串的开始索引，j 代表结束索引。需要注意的是，在截取子字符串的时候将包含起始字符，但不包含结束字符，这是一个半开闭区间。

Python 在字符串切片的功能上有很好的默认值。省略第 1 个索引，默认为 0；省略第 2 个索引，默认为切片字符串的长度，如代码 2-25 所示。

代码 2-25　字符串切片

```
>>>word[0:3]                            # 截取第 1 ~ 3 个字符
'Pyth'
>>>word[:3]                             # 截取第 1 ~ 3 个字符
'Pyth'
```

```
>>>word[4:]                                    # 截取第五到最后一个字符
'on'
```

事实上，对于这种没有意义的切片索引，Python 还是可以优雅地处理：当第 2 个索引越界时，将被切片字符串实际长度替代；当第 1 个索引大于字符串实际长度时，返回空字符串；当第 1 个索引值大于第 2 个索引值时，也返回空，如代码 2-26 所示。

代码 2-26　索引越界

```
>>>word[3:52]                                  # 第 2 个索引越界
'hon'
>>>word[52:]                                   # 第 1 个索引超出字符串长度
''
>>>word[-1:3]                                  # 第 1 个索引为负，第 2 个索引正常
''
>>>word[5:3]                                   # 第 1 个索引大于第 2 个索引
''
```

Python 中，字符串是不可以更改的，所以，如果给指定位置的字符重新赋值，将会出错，如代码 2-27 所示。

代码 2-27　字符不可修改

```
>>>word[0] = 'p'                               # 字符不可被修改
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    word[0] = 'p'
TypeError: 'str' object does not support item assignment
```

（3）字符串拼接

如果要修改字符串，最好的办法是重新创建一个。如果只需要改变其中的小部分字符，可以使用字符串拼接方法。

字符串拼接时，可以使用加号（+）将两个字符串拼接起来，使用星号（*）表示重复。另外，相邻的两个字符串文本是会自动拼接在一起的，如代码 2-28 所示。

代码 2-28　字符串拼接

```
>>>'Python is' + 3 *  ' good'                 #  加号拼接字符串
'Python is good good good '
>>>'Python is' ' good'                        #  相邻字符串自然拼接
'Python is good'
```

如果要将字符串“Life is short,you need something.”修改成“Life is short,you need Python.”，实现代码如代码 2-29 所示。

代码 2-29　字符串修改

```
>>>sentence = 'Life is short,you need something.'
>>> sentence[:23] + 'Python.'
'Life is short,you need Python.'
```

2.2.4 任务实现

根据任务分析，本任务的具体实现过程可以参考如下操作。

（1）“Apple's unit price is 9 yuan.”中含有单引号（'），使用反斜杠（\）进行转义。

（2）直接使用方括号（[ ]）提取字符串中指定位置的字符。

（3）使用 type 函数查看数据类型。

（4）使用 int 函数将数据转换成整型。

参考代码如任务实现 2-1 所示。

任务实现 2-1

```
# -*-coding:utf8-*

applePriceStr = 'Apple\'s unit price is 9 yuan.'
applePrice = applePriceStr[22]                                    # 提取数值
print('提取了苹果的单价为：',applePrice,'。此刻它的数据类型为：',type(applePrice))
applePrice = int(applePrice)                                      # 字符转数值
print('转换数据类型后：',type(applePrice))
```

任务 2.3 计算圆形的各参数

任务描述

设计一个小程序，运用本节介绍的操作运算符实现输入、输出圆形的基本参数。

首先了解圆形的基本计算公式，如式（2-1）所示。

$$C = 2\pi r,\ S=\pi r^2 \tag{2-1}$$

其中，r 代表圆形的半径，C 代表圆形的周长，S 代表圆形的面积，π 是圆周率。

由式（2-1）可得式（2-2）。

$$r = \frac{C}{2\pi} = \sqrt{\frac{S}{\pi}} \tag{2-2}$$

运用运算符实现上述两个表达式。

任务分析

（1）输入半径，输出面积及周长。

（2）输入面积，输出半径及周长。

（3）输入周长，输出半径及面积。

2.3.1 掌握常用操作运算符

Python 中提供了一系列便利的基础运算符，可用于数据分析研究。满足基本运算需求的运算符主要有算术运算符、比较运算符、赋值运算符。

1. 算术运算符

算术运算符是对操作数进行运算的一系列特殊符号，能够满足一般的运算操作需求。表 2-2 列举了 Python 3 提供的一系列算术运算符。

表 2-2　常用算术运算符

运算符	描　　述	示　　例
+	加，即两个对象相加	10+20 输出结果 30
–	减，即得到负数或是一个数减去另一个数	20–10 输出结果 10
*	乘，即两个数相乘或是返回一个被重复若干次的字符串	10*20 输出结果 200
/	除，即 x 除以 y	20/10 输出结果 2.0
%	取模，即返回除法的余数	23%10 输出结果 3
**	幂，即返回 x 的 y 次方	2**3 输出结果为 8
//	取整除，即返回商的整数部分	23//10 输出结果 2

在进行除法（/）运算时，不管商为整数还是浮点数，结果始终为浮点数。如果希望得到整型的商，需要用到双斜杠（//）。对于其他运算，只要任一操作数为浮点数，结果就是浮点数，如代码 2-30 所示。

代码 2-30　算数运算符示例

```
>>>2 / 1 ;  type(2 / 1)                                          # 单斜杠除法
2.0
<class 'float'>
>>>2 // 1 ;  type(2 // 1)                                        # 双斜杠除法
2
<class 'int'>
>>>print(1 + 2, 'and',1.0 + 2) ; print(1 * 2, 'and',1.0 * 2)   # 加法和乘法
3 and 3.0
2 and 2.0
>>>print('23 除以 10，商为：',23 // 10,'，余数为：',23 % 10)    # 商和余数
23 除以 10，商为： 2 ，余数为： 3
>>>3 *'Python'                                                  # 字符串的 n 次重复
'PythonPythonPython'
```

2. 比较运算符

比较运算符一般用于数值的比较，也可用于字符的比较，常用比较运算符如表 2-3 所示。

当两个数值比较结果是正确时返回 True，否则返回 False。

可能关于字符的比较对于刚接触编程的读者来说会比较生疏，这里对其中的原理做简单的介绍，感兴趣的读者也可以查找更多的资料进行深入了解。Python 中，字符是符合 ASCII 编码的，每个字符都有属于自己的编码，字符的比较本质是字符的 ASCII 编码的比较。

表 2-3　常用比较运算符

运算符	描　　述	示　　例
==	等于，即比较对象是否相等	(1==2)返回 False
!=	不等于，即比较两个对象是否不相等	(1!=2)返回 True
>	大于，即返回 x 是否大于 y	(1>2)返回 False
<	小于，即返回 x 是否小于 y	(1<2)返回 True
>=	大于，等于即返回 x 是否大于等于 y	(1>=2)返回 False
<=	小于，等于即返回 x 是否小于等于 y	(1<=2)返回 True

Python 提供了如下两个可以进行字符与编码转换的函数。

（1）ord 函数：将 ASCII 字符转换为对应的数值。

（2）chr 函数：将数值转换为对应的 ASCII 字符。

比较运算符的应用示例如代码 2-31 所示。

代码 2-31　比较运算符示例

```
>>>1 == 2 ;  1 != 2                                          # 数值的比较
False
True
>>>print('a' == 'b', 'a' != 'b') ;  print('a' < 'b','a' > 'b')    # 字母的比较
False True
True False
>>>print(ord('a'),ord('b'))                                  # 查看字母编码
97 98
>>>print(chr(97),chr(98))                                    # 查看编码对应的字符
a b
>>>'# ' < '$'                                               # 符号的比较
True
```

3. 赋值运算符

赋值运算符用于对变量的赋值和更新，从表 2-4 中可以看出，Python 除了简单的赋值运算符外，还有一类特殊的赋值运算符，比如加法赋值运算符、减法赋值运算符等。除简单赋值运算符外，其他都属于特殊赋值运算符。

表 2-4　常用赋值运算符

运算符	描　　述	示　　例
=	简单的赋值运算符	c=a+b 将 a+b 的运算结果赋值为 c
+=	加法赋值运算符	a+=b 等效于 a=a+b
-=	减法赋值运算符	a-=b 等效于 a=a-b

续表

运算符	描　　述	示　　例
*=	乘法赋值运算符	a*=b 等效于 a=a*b
/=	除法赋值运算符	a/=b 等效于 a=a/b
%=	取模赋值运算符	a%=b 等效于 a=a%b
**=	幂赋值运算符	a**=b 等效于 a=a**b
//=	取整除赋值运算符	a//=b 等效于 a=a//b

关于表 2-4 中的特殊赋值运算符，也可以将其看作是变量的快速更新，更新意味着该变量是存在的。对于一个之前不存在的变量，则不能使用特殊赋值运算符，如代码 2-32 所示。

代码 2-32　赋值运算符示例

```
>>>a = 1 + 2 ;  print(a)                                          # 简单赋值运算
3
>>> print('a: ',a) ;   a += 4 ;  print('a += 4 特殊赋值运算后，a: ',a) # 特殊赋值运算
a: 3
a += 4 特殊赋值运算后，a:  7
>>>f += 4                                                  # 未定义变量不能进行特殊赋值运算
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'f' is not defined
```

4．按位运算符

通常，数字都是使用十进制的，按位运算符会自动将输入的十进制数转为二进制数，再进行相应的运算。

表 2-5 列举了常用按位运算符，示例中，a 为 60，b 为 13，对应的二进制值如下。

```
a = 0011 1100
b = 0000 1101
```

表 2-5　常用按位运算符

运　算　符	描　　述	示　　例
&	按位与运算符：参与运算的两个值如果相应位都为 1，则该位的结果为 1，否则为 0	a & b 输出结果 12， 二进制值：0000 1100
|	按位或运算符：只要对应的两个二进位有一个为 1，结果位就为 1	a | b 输出结果 61， 二进制值：0011 1101
^	按位异或运算符：当两对应的二进位相异时，结果为 1	a ^ b 输出结果 49， 二进制值：0011 0001

续表

运 算 符	描 述	示 例
~	按位取反运算符：对数据的每个二进制位取反，即把 1 变为 0，把 0 变为 1	~a 输出结果–61， 二进制值：1100 0011
<<	左移动运算符：运算数的各二进位全部左移若干位，由“<<”右边的数指定移动的位数，高位丢弃，低位补 0	a << 2 输出结果 240， 二进制值：1111 0000
>>	右移动运算符：把“>>”左边的运算数的各二进位全部右移若干位，“>>”右边的数指定移动的位数	a >> 2 输出结果 15， 二进制值：0000 1111

按位运算符是对二进制数的运算，其中比较难理解的应该就是取反运算，本书后面会详细讲解相关知识。关于位运算的示例代码 2-33 所示。

代码 2-33 按位运算符示例

```
>>>a = 60 ;  b = 13 ;  print('a = 60,b =13')                              # 初始赋值
a = 60,b =13
>>>print('a & b =',a & b) ;   print('a | b =',a | b) ;   print('a ^ b =',a ^ b)
# 或、与、异位运算
a & b = 12
a | b = 61
a ^ b = 49
>>> print('~a =',~a) ;   print('a << 2 =', a << 2) ;   print('a >> 2 =', a >> 2)
# 取反和位移运算
~a = -61
a << 2 = 240
a >> 2 = 15
```

这里以按位与和按位取反运算为例，具体讲解计算过程。

（1）按位与运算

按位与运算：参与运算的两个值如果两个相应位都为 1，则该位的结果为 1，否则为 0。

如下述代码所示，a 和 b 的第 3、4 位都为 1，其他位置的数都没有同时为 1，故对 a 和 b 做按位与运算的结果在第 3、4 位为 1，其余位置都为 0 时。

```
a = 0011 1100
b = 0000 1101
a & b = 0000 1100
```

（2）按位取反运算

按位取反涉及补码的计算，相对比较复杂。

十进制数的二进制原码包括符号位和二进制值。以 60 为例，其二进制原码为“0011 1100”，从左到右第 1 位为符号位，其中，0 代表正数，1 代表负数。

对于正数来说，其补码与二进制原码相同；对于负数而言，其补码为，二进制原码符号位保持不变，其余各位取反后再在最后一位加 1。

例 2-1 对 60 进行取反。
取 60 的二进制原码：0011 1100。
取补码：0011 1100。
每一位取反：1100 0011，得到最终结果的补码（负数）。
取补码：1011 1101 得到最终结果的原码。
转换为十进制数：–61，所以 60 取反后为–61。
例 2-2 对–61 进行取反。
取–61 的二进制原码：1011 1101。
取补码：1100 0011。
每一位取反：0011 1100，得到最终结果的补码（正数）。
取补码：0011 1100 得到最终结果的原码。
转换为十进制：60，所以–61 取反后为：60。
例 2-1 和例 2-2 已经很好地展示了正数和负数的取反操作，可以总结为以下 5 个步骤。
（1）取十进制数的二进制原码。
（2）对原码取补码。
（3）补码取反（得到最终结果的补码）。
（4）取反结果再取补码（得到最终结果的原码）。
（5）二进制原码转十进制数。

5. 逻辑运算符

逻辑运算符包括 and、or、not，具体用法如表 2-6 所示，示例中 a 为 11，b 为 22。

表 2-6　逻辑运算符

运　算　符	逻辑表达式	描　　述	示　　例
and	x and y	布尔“与”，即 x and y，如果 x 为 False，返回 False；否则它返回 y 的计算值	a and b，返回 22
or	x or y	布尔“或”，即 x or y，如果 x 是 True，它返回 True；否则它返回 y 的计算值	a or b，返回 11
not	not x	布尔“非”，即 not(x)，如果 x 为 True，返回 False。如果 x 为 False，它返回 True	not(a and b)，返回 False

逻辑运算符应用举例如代码 2-34 所示。

代码 2-34　逻辑运算符示例

```
>>>a = 11;b = 22;print('a = 11,b =22')                                    # 初始赋值
a = 11,b =22
>>>print('a and b =',a and b); print('a or b =',a or b); print('not(a and b) =',
not(a and b)); # and、or、not 运算
a and b = 22
a or b = 11
```

```
not(a and b) = False
>>>a = 0;b = 22;print('a = 0,b =22')                                    # 重新赋值
a = 0,b =22
>>>print('a and b =',a and b); print('a or b =',a or b); print('not(a and b) =',
not(a and b)); # and、or、not 运算
a and b = 0
a or b = 22
not(a and b) = True
```

按位运算符和逻辑运算符用于 bool 值运算时，按位&和逻辑 and 的运算效果一样，当符号左右两个值都为 True 时，返回结果 True，否则返回 False；按位|和逻辑 or 的运算效果一样，当符号左右两个值中有一个值为 True 时，返回结果 True，否则返回 False，如代码 2-35 所示。

代码 2-35　bool 值运算

```
>>>True & True ;   True and True                    # 按位&、逻辑 and
True
True
>>> True | False ;   True or False;                # 按位|、逻辑 or
True
True
>>>True & False ;   True and False;
False
False
>>> False | False ;  False or False;
False
False
```

6. 成员运算符

成员运算符的作用是判断某指定值是否存在于某一序列中，包括字符串、列表或元组。成员运算符的相关解释如表 2-7 所示。

表 2-7　成员运算符

运 算 符	描　　述	示　　例
in	如果在指定的序列中找到值，返回 True，否则返回 False	x in y，x 在 y 序列中，返回 True
not in	如果在指定的序列中没有找到值，返回 True，否则返回 False	x not y，x 不在 y 序列中，返回 True

在成员运算中，对于成员的运算不仅包含值的大小，还包括类型的判断。通过代码 2-36 可以看出，在 List 中 1 是数值，所以判断数值 1 是否属于 List 时返回 True；但是判断包含在列表中的数值 1 时，就返回结果 False，因为类型不匹配。另外，判断[4,5]是否属于 List

时，返回结果为 True，很明显是因为 List 中包含了该值。

代码 2-36　成员运算符示例

```
>>>List = [1,2,3.0,[4,5],'Python3']          # 初始化列表 List
>>> 1 in List                                 # 查看 1 是否在列表内
True
>>>[1] in List                                # 查看[1]是否在列表内
False
>>> 3 in List                                 # 查看 3 是否在列表内
True
>>>[4,5] in List                              # 查看[4,5]是否在列表内
True
>>> 'Python' in List                          # 查看字符串'Python'是否在列表内
False
>>>'Python3' in List                         # 查看字符串'Python3'是否在列表内
True
```

7. 身份运算符

身份运算符用于比较两个对象的内存地址，说明如表 2-8 所示。

表 2-8　身份运算符

运　算　符	描　　述	示　　例
is	用于判断两个标识符是不是引用自一个对象	x is y，如果 id(x)等于 id(y)，返回结果 1
is not	用于判断两个标识符是不是引用自不同对象	x is not y，如果 id(x)不等于 id(y)，返回结果 1

在身份运算中，内存地址相同的两个变量进行 is 运算时，返回 True；内存地址不同的两个变量进行 is not 运算时，返回 True。如代码 2-37 所示，当 a、b 获取一样的值时，实质上这两个变量也就获取了同样的内存地址。

代码 2-37　身份运算符示例

```
>>>a = 11 ;  b = 11 ;  print('a = 11,b = 11')          # 初始化 a、b
a = 11,b = 11
>>> a is b ;  a is not b                               # 身份运算
True
False
>>>id(a) ;  id(b)                                     # 查看 id 地址
1347990912
1347990912
>>>a = 11 ;  b = 22 ;  print('a = 11,b = 22')          # 重新赋值 b
a = 11,b = 22
```

```
>>> a is b ;  a is not b                                    # 身份运算
False
True
>>>id(a) ;  id(b)                                          # 查看 id
1347990912
1347991264
```

2.3.2 掌握运算符优先级

在 Python 的应用中，通常运算的形式是表达式。表达式由运算符和操作数组成。比如 1+2 就是一个表达式，“+”是操作符，“1”和“2”是操作数。

一个表达式往往不只包含一个运算符，当一个表达式存在多个运算符时，各运算符的优先级如表 2-9 所示，处于同一优先级的运算符则从左到右依次运算。

表 2-9 运算符优先级比较

运 算 符	描 述
**	指数（最高优先级）
~ + –	按位翻转、一元加号和减号（最后两个的方法名为+@和–@）
* / % //	乘、除、取模和取整除
+ –	加法减法
>> <<	右移、左移运算符
&	按位与运算符
^	按位或运算符
<= < > >=	比较运算符
<> == !=	等于运算符
= %= /= //= –= += *= **=	赋值运算符
is is not	身份运算符
in not in	成员运算符
not or and	逻辑运算符

对于表 2-9 第二行中的“+”“–”，可以更简单地理解为，放在一个数值前面，标识该数值的正负属性。代码 2-38 展示了简单的表达式运算。

代码 2-38 运算符的优先级示例

```
>>>24 + 12 / 6 ** 2 * 18                    # 24+12/36*18 → 24+(1/3)*18 → 24+6
30.0
>>> 24 + 12 / ( 6 ** 2 ) * 18               # 24+12/36*18 → 24+(1/3)*18 → 24+6
30.0
>>>24 + ( 12 / ( 6 ** 2 ) ) * 18           # 24+(12/36)*18→24+(1/3)*18→24+6
```

```
30.0
>>>24 + ( 12 / 6 ) ** 2 * 18                    # 24+2**2*18 → 24+4*18 → 24+72
96.0
>>>( 24 + 12 ) / 6 ** 2 * 18                   # 36/6**2*18 → 36/36*18 → 1*18
18.0
>>>- 4 * 5 + 3                                 # -20+3
-17
>>>4 * - 5 + 3                                 # -20+3
-17
```

2.3.3　任务实现

根据任务分析，本任务的具体实现过程可以参考如下操作。

（1）使用算术运算符按要求计算圆形指定参数的表达式。

（2）输入一个圆形的半径，通过表达式计算周长和面积。

（3）输入一个圆形的周长，通过表达式计算半径和面积。

（4）输入一个圆形的面积，通过表达式计算半径和周长。

（5）关于公式中的常量 pi，这里取 3.14。

（6）round 函数可以指定保留小数的位数。

参考代码如任务实现 2-2 所示。

任务实现 2-2

```
# -*-coding:utf-8-*-

'''
根据输入计算圆形的其他参数
关于圆形的相关计算公式参考正文
'''
pi = 3.14                                              # 设置常量
# 输入半径，求周长、面积
r = 3                                                  # 输入圆形的半径
C = 2 * pi * r                                         # 计算圆形的周长
S = pi * r ** 2                                        # 计算圆形的面积
print('半径为',r,'的圆形，其周长等于',C,'；面积等于',S,'。')

#  输入周长，求半径、面积
C = 5                                                  # 输入圆形的周长
r = C / ( 2 * pi )                                     # 计算圆形的半径
S = pi * r ** 2                                        # 计算圆形的面积
print('周长为'+str(C)+'的圆形，其半径为'+str(r)+'；面积等于'+str(S)+'。')

#  输入面积，求半径、周长
```

```
S = 5                                                    # 输入圆形的面积
r = round(( S / pi ) ** 0.5 , 2)                         # 计算圆形的半径，并保留两位小数
C = round( 2 * pi * r , 2)                               # 计算圆形的周长，并保留两位小数
str_print = '面积为' + str(S) + '的圆形，其半径为' + str(r) + '；周长等于' + str(C)
+ '。'
print(str_print)
```

小结

Python 相比于其他编程语言，更加优雅简单，在使用中既保持了其他语言风格的基本特点，又有自己独特的使用方法。本章介绍了 Python 的固定语法，从 Python 固定语法中就可以发现其独特的地方。读者掌握好这部分内容，对于将来处理更复杂的程序是有很大帮助的。

此外，本章还介绍了两个简单的数据类型，分别是数值型和字符型。下面总结一下本章涉及的具体知识点。

（1）Python 固定语法主要体现在编码声明、注释、多行语句、行与缩进、标识符与保留字符 5 个方面。学习 Python，固定语法是基础之基础，只有掌握了固定语法，对于后续的知识运用、代码管理、debug 才可以得心应手。

（2）Python 变量赋值属于地址传递，两个值相同的变量事实上都指向同一地址，可以用身份运算符进行检验。

（3）Python 3 的数据类型包含了 4 个子类，分别是 int、float、bool、complex。每种子类之间相互联系，又互相独立。

（4）Python 3 的字符类型可由单引号、双引号、三引号（3 个单引号或者 3 个双引号）进行标识。同时，对于字符串的索引取值、切片、拼接的操作，Python 3 都有自己内置的方法可以简单操作。

（5）Python 一共有 7 种类型的操作符，是算术运算符、比较运算符、赋值运算符、按位运算符、逻辑运算符、成员运算符和身份运算符。一个表达式可以包含一个或多个运算符，这时需要严格遵守运算符的优先级进行运算。

实训

实训 1　对用户星座进行分析并输出结果

1. 训练要点

（1）掌握 input 函数的使用。

（2）掌握格式化打印字符的方法。

（3）掌握字符转换成数值的方法。

（4）掌握索引的使用要点。

2. 需求说明

使用字符串“请输入您的名字:”提醒用户输入名字，接着格式化打印星座对应日期信息，使用字符串“请根据如上提示选择对应编号”（例如，水瓶座请输入：1）提醒用户根

据信息输入数字，最后根据用户的输入直接打印分析结果，输出结果为“名字，您好！星座的您星座分析结果：结果”。

格式化展示的星座日期信息如表 2-10 所示。

表 2-10　星座日期对应表

编　号	星　座	日　期
1	水瓶	1 月 20 ~ 2 月 18
2	双鱼	2 月 19 ~ 3 月 20
3	白羊	3 月 21 ~ 4 月 19
4	金牛	4 月 20 ~ 5 月 20
5	双子	5 月 21 ~ 6 月 21
6	巨蟹	6 月 21 ~ 7 月 22
7	狮子	7 月 23 ~ 8 月 22
8	处女	8 月 23 ~ 9 月 22

3. 实训思路及步骤

（1）利用函数 input(‘字符串’)，可以直接输入提示语。

（2）字符串格式化打印使用 3 个单引号或者 3 个双引号进行标识。

（3）通过 input 函数获取的值都是字符型，所以即使输入数字，也会被转换成字符。如果需要保留数值类型，可以通过 int 函数进行转换。

实训 2　通过表达式计算给定 3 个数值的均值、方差、标准差

1. 训练要点

（1）均值、方差、标准差的计算公式如式（2-3）所示。

$$\mu=(n_1+n_2+\cdots+n_n)/m,\ \sigma^2=\frac{(n_1-\mu)^2+(n_2-\mu)^2+\cdots+(n_m-\mu)^2}{m},\ \sigma=\sqrt{\sigma^2} \quad (2\text{-}3)$$

其中，m 代表进行计算的数值个数，n_1，n_2，…，n_m 则表示进行计算的具体数值，μ 表示均值，σ^2 表示方差，σ 表示标准差。

（2）合理应用算术运算符构建上述表达式。

2. 需求说明

输入 3 个数值，这里指定为 11、2、5，计算它们的均值、方差、标准差，并打印输出。

3. 实训思路及步骤

（1）数值变量赋值。

（2）运用运算符构造均值、方差、标准差计算表达式。

（3）通过 print 函数打印输出结果。

课后习题

1. 选择题

（1）Python 3 支持多行语句，下面对于多行语句描述有误的是（　　）。

A．一行可以书写多个语句　　B．一个语句可以分多行书写

C．一行多语句可以用分号隔开　　D．一个语句多行书写时直接按回车即可

（2）标识符可以用于变量、函数、对象等的命名，对于标识符描述有误的的是（　　）。

A．标识符不可以以数字开头　　B．标识符可以由数字、字母和下划线组成

C．标识符不区分大小写　　D．保留字符做标识符时会出错

（3）对于字符串的标识，Python 中可使用的方法很多，下面正确的是（　　）。

A．"What's happened to you?"　　B．'What's happened to you?'

C．'What\\'s happened to you?'　　D．""Oh!" It sounds terrible."

（4）下列运算符中优先级最高的是（　　）。

A．&　　B．is　　C．/　　D．**

（5）实际应用中变量的使用是避免不了的，而在 Python 中，如下变量使用正确的是（　　）。

A．numvalue=10　　B．numSum+=10，不需要事先声明变量

C．“Val”和“val”是同一个变量　　D．yield=‘str’

（6）下面不属于按位运算符的是（　　）。

A．|　　B．//　　C．~　　D．^

（7）如下对于字符串拼接有误的是（　　）。

A．"Life is short, "　"you need Python."

B．"Life is short, " + "you need Python."

C．" Life is short, "　2 * "you need Python."

D．"Life is short, " + 2 * "you need Python."

（8）下列（　　）是“3 and 4”的运算结果。

A．0　　B．1　　C．3　　D．4

（9）在书写 Python 脚本时，需要进行必要的编码声明，关于编码声明错误的是（　　）。

A．在首行声明有效　　B．在第二行声明有效

C．在第三行声明有效　　D．只有在首行或第二行声明才有效

（10）Python 的赋值功能很强大，当 a=11 时，运行 a+=11 后，a 的结果是（　　）。

A．11　　B．12　　C．True　　D．22

2. 操作题

（1）使用 int 函数分别对 5.20、–5.20、5.60、–5.60 四舍五入后取整。

（2）当 Bet 等于 6 时，利用 Python 表达式判断 Bet 是否在(1,20)区间内。是否在(-∞，10)和(20，∞)区间呢?

第 3 章 Python 数据结构

第 2 章介绍了 Python 的两种基础数据类型——数值型和字符型，要实现 Python 更复杂更强大的功能，仅靠这两种数据类型是不够的，还需要数据结构来完成。本章将介绍 Python 的一些基础数据结构，及其各自的特性和常用基本操作等。

学习目标

（1）认识 Python 数据结构类型，并区分可变数据类型与不可变数据类型。
（2）掌握列表的创建，以及增删改查等操作。
（3）掌握元组与列表的区别，以及取值操作。
（4）掌握字典的创建，以及增删改查等操作。
（5）掌握集合的创建，并进行几个运算。

任务 3.1 认识 Python 数据结构的组成

任务描述

Python 有 4 个内建的数据结构，它们可以统称为容器（Container），因为它们实际上是由一些“东西”组合而成的结构。这些“东西”可以是数字、字符甚至列表，或是它们的组合。在介绍各种数据结构的具体内容之前，本书将先介绍 Python 数据结构的组成方式和区分可变数据类型与不可变数据类型的方法。

任务分析

认识 Python 数据结构主要分为以下两个部分，如图 3-1 所示。
（1）认识序列（如列表和元组）、映射（如字典）及集合 3 种基本的数据结构类型。
（2）掌握可变数据类型和不可变数据类型的区别。

3.1.1 认识数据结构类型

Python 中的数据结构是根据某种方式将数据元素组合起来形成的一个数据元素集合，其中主要包含序列（如列表和元组）、映射（如字典）以及集合 3 种基本的数据结构类型。几乎所有的 Python 数据结构都可以归结为这 3 种数据结构类型。

1. 序列类型

序列是数据结构对象的有序排列，数据结构对象作为序列的元素都会被分配一个位置

编号（也称为索引），序列就相当于数学中数列的概念。Python 中的序列类型包括字符串（string）、列表（list）、元组（tuple）、Unicode 字符串、buffer 对象、xrange 对象等数据结构，其中字符串、列表和元组最为常用。

2. 映射类型

映射类型就是存储了对象与对象之间的映射关系的数据结构类型，Python 中唯一的映射类型数据结构是字典（dictionary），字典中的每个元素都存在相应的名称（称为键）与之一一对应。字典相当于带有各自名称的元素组成的集合。与序列不同的是，字典中的元素并没有排列顺序。

3. 集合类型

除了上述基本数据结构类型外，Python 还提供了一种称为集合的数据结构。集合当中的元素不能重复出现，即集合中的元素是相对唯一的，并且元素不存在排列顺序。由此可以看出，Python 中的集合概念相当于数学中的集合概念。集合类型包括可变集合（set）与不可变集合（frozenset）。

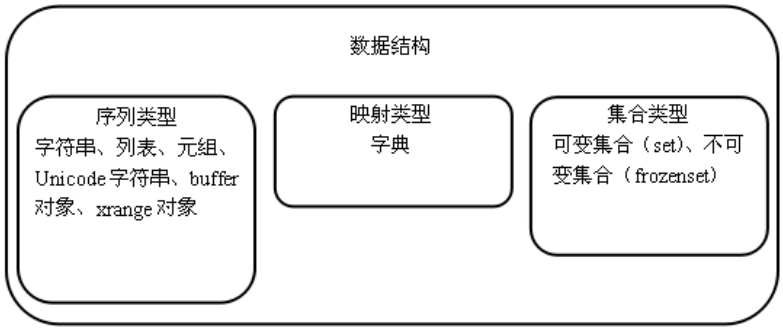


图 3-1　数据结构

3.1.2　区分可变数据类型与不可变数据类型

另外，在 Python 中，还有两个比较重要的关于数据结构的概念，即可变数据类型与不可变数据类型。

1. 可变数据类型

通过可变数据类型，可以直接对数据结构对象的内容进行修改（并非是重新对对象赋值操作），即可以对数据结构对象进行元素的赋值修改、删除或增加等操作。由于可变数据类型对象能直接对自身进行修改，所以修改后的新结果仍与原对象引用同一个 id 地址值，即由始至终只对同一个对象进行了操作。Python 中比较重要的可变数据类型包括列表、字典、可变集合等。

2. 不可变数据类型

与可变数据类型不同，不可变数据类型不能对数据结构对象的内容进行修改操作，不可对对象中的元素进行增加、删除和赋值修改。若需要对对象进行内容修改，则需要对其变量名进行重新赋值，赋值操作会把变量名指向一个新对象，新旧对象两者引用两个不同的 id 地址值。常用的不可变数据类型包括数字、字符串、元组、不可变集合等。

任务 3.2　创建一个列表（list）并进行增删改查操作

任务描述

在列表[110,‘dog’,‘cat’,120,‘apple’]中关于动物的字符串之间插入一个空列表，并删除关于水果的字符串，同时查找出列表中的数值并在列表中增大 10 倍。

任务分析

通过如下步骤实现上述任务。

（1）创建一个列表对象[110,‘dog’,‘cat’,120,‘apple’]。

（2）在字符串“dog”和“cat”之间插入空列表。

（3）删除列表中“apple”这个字符串。

（4）分别查找出列表中的数值 110、120。

（5）对查找出来的数值 110 和 120 以 10 为乘数做自乘运算操作。

（6）打印列表对象，确认是否完成以上操作。

3.2.1　了解列表的概念与特性

列表（list）是 Python 对象作为其元素并按顺序排列构成的有序集合，列表中的每个元素都有各自的位置编号，称为索引。列表当中的元素可以是各种类型的对象，无论是数字、字符串、元组、字典，还是列表类型本身，都可以作为列表当中的一个元素。此外，列表当中的元素可以重复出现。要注意的是，列表是可变数据类型，因此可以对列表对象自身进行内容修改，即对列表进行增添、删除、修改元素等操作。

3.2.2　创建列表

使用 Python 可以很轻松地创建一个列表对象，只需将列表元素传入特定的格式或函数中就能实现。常用的创建列表的方法有两种，一种是使用方括号[ ]进行创建，另一种是使用 list 函数进行创建。

1．使用方括号[ ]创建

使用方括号[ ]创建列表对象，只需要把所需的列表元素以逗号隔开，并用方括号[ ]将其括起来。当使用方括号[ ]而不传入任何元素时，就可创建一个空列表。Python 的列表对象中允许包括任意类型的对象，其中也包括列表对象，这说明可以创建嵌套列表，如代码 3-1 所示。

代码 3-1　使用方括号[ ]创建列表

```
>>> mylist1=[1,2.0,['three','four',5],6.5,True]  # 创建包含混合数据类型的嵌套列表
>>> mylist1                                      # 查看列表内容
[1, 2.0, ['three', 'four', 5], 6.5, True]
>>> empty_list=[]                                # 创建空列表
>>> empty_list
[]
```

2. 使用 list 函数创建

Python 中 list 函数的作用实质上是将传入的数据结构对象转换成列表类型，例如向函数传入一个元组对象，就会将对象从元组类型转变为列表类型。由于其返回的是一个列表对象，因此可以看作是创建列表的一个方法，使用时可以用圆括号或方括号把元素按顺序包括起来，元素之间以逗号隔开，并传入函数当中。若不传入任何对象到 list 函数中，则会创建一个空列表。使用 list 函数创建列表对象的实现如代码 3-2 所示。

代码 3-2　使用 list 函数创建列表

```
>>> mylist1=list((1,2.0,['three','four',5],6.5,True))   # 向 list 函数传入一个对象
>>> mylist1
[1, 2.0, ['three', 'four', 5], 6.5, True]
>>> type(mylist1)                                       # 查看对象类型
<class 'list'>
>>> empty_list=list()                                   # 创建空列表
>>> empty_list
[]
>>> mylist2=list(['one','two','three'])              # 向 list 函数传入一个列表对象
>>> mylist2
['one', 'two', 'three']
```

从代码 3-2 的 mylist 变量可以看出，用圆括号括起来的数据结构集合传入 list 函数，这是本书 3.3 节将要介绍的元组对象。

list 函数实际上是将传入对象转换为列表类型，如果将字符串传入函数，list 函数会把字符串中的每个字符元素作为一个列表元素，然后将这些元素放入一个列表，看起来就像字符串被“拆开”成一个个字符一样，如代码 3-3 所示。

代码 3-3　字符串传入 list 函数

```
>>> list('hello world!')                              # 向函数 list 传入一个字符串
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '!']
```

3.2.3　列表的基础操作

1. 列表索引

序列类型的数据结构都可以通过索引和切片操作对元素进行提取，字符串、列表和元组都属于序列类型，因此，对列表元素的提取方法与第 2 章介绍的字符串提取方法一样。列表的索引也是从 0 开始，以 1 为步长逐渐递增，这种索引的定义方式或许与日常生活中所理解的有所出入，但这里建议读者可以尝试将索引理解为元素相对于第一个元素的位置偏移量，例如第一个元素的位置偏移量是 0，故其索引为 0；第二个元素的位置偏移量是 1，所以索引为 1；其他元素以此类推。列表的负索引概念与字符串一样，也是按从右到左的方向标记元素，最右边元素的负索引为–1，然后向左依次为–2、–3 等。

类似于字符串，列表元素的提取方法有两种，索引访问提取和列表切片操作提取。其中，索引访问提取仅返回列表的一个对应元素，而列表切片操作则会返回列表中对应的子

列表。

2. 列表索引访问提取

为提取列表中的某个元素，可以在列表对象后面紧接方括号[ ]包括索引，这样就能提取出列表中对应的元素。序列的索引访问提取具体格式为 sequence_name[index]，即序列对象[索引]。正是由于 Python 允许传入负索引并进行元素提取，所以可以很方便地从对列表尾端提取元素，如代码 3-4 所示。

代码 3-4　列表元素提取

```
>>> mylist3=['Sunday','Monday','Tuesday',
...         'Wednesday','Thursday','Friday']
>>> mylist3[1]                                  # 提取列表中第 2 个元素
'Monday'
>>> mylist3[-3]                                 # 提取列表中倒数第 3 个元素
'Wednesday'
```

注意，当传入的索引超出列表正索引或负索引范围时，即小于第 1 个元素的负索引或大于最后一个元素的正索引时，Python 会返回一个错误，如代码 3-5 所示。

代码 3-5　索引错误示例

```
>>> mylist3[7]                                  # 传入的索引大于最后一个元素的正索引
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> mylist3[-10]                                # 传入的索引小于第 1 个元素的负索引
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

3. 列表的切片操作提取

通常对列表进行处理的时候，除了需要提取当中某个元素外，还可能需要提取列表中的子列表元素，这就需要通过列表的切片操作来完成。在进行切片操作时，只需要传入所提取子序列起始元素的索引、终止元素的索引以及提取步长值，此时得到的序列切片将包含从起始元素开始，以提取步长为间隔，到终止元素之前的所有元素。这里需要注意，切片操作在取到终止元素索引为止，并不包含终止元素，相当于数学中的半开半闭区间。具体切片操作格式为 sequence_name[start:end:step]，即序列对象[起始元素:终止元素:步长值]。

在切片操作格式当中，省略步长值时取，默认步长值为 1，此时格式中的第 2 个冒号可以省略。当步长值为正数时，表示切片从左往右方向提取元素，一般需要起始元素位置小于终止元素位置；若为负数，则表示从右往左方向提取，此时起始元素位置应该大于终止元素位置。Python 步长值为 0 时会报错，因为搜索元素时一步都不迈出去毫无意义，如代码 3-6 所示。

代码 3-6　列表切片

```
# 步长为正数时的切片操作
>>> mylist4=[10,20,30,40,50,60,70,80,90,100]
>>> mylist4[2:7]              # 提取从第 2 ~ 8 个元素之间的元素
[30, 40, 50, 60, 70]
>>> mylist4[1:9:2]            # 提取从第 1 ~ 10 个元素之间的元素，步长为 2
[20, 40, 60, 80]
# 步长为负数时的切片操作
>>> mylist4[-2:-8:-2]         # 提取倒数第 1 ~ 8 个元素之间的元素，步长为 2
[90, 70, 50]
>>> mylist4[1:4:0]            # 步长为 0 时会报错
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: slice step cannot be zero
```

除了步长值可以省略以外，也可以省略格式中的起始索引和终止索引，但第 1 个冒号必须存在。若只省略起始索引，切片操作会默认使用开头或结尾的索引（0 或–1，视具体提取方向而定），即会从序列开头或结尾开始提取元素；若只省略终止索引，切片操作会从起始索引开始，按提取方向搜索到序列一端的最后一个元素，这里切片操作会包含那一端最后一个元素，类似于数学中的闭区间；若两者同时省略，切片操作就会从某端开始对全体元素搜索提取（从哪端开始视具体提取方向而定）。这里有一个小技巧：使用切片操作 list_name[::-1]可以将列表反转，其实这里就是从列表右端开始，进行逐个元素提取，直至提取所有元素，如代码 3-7 所示。

代码 3-7　列表反转

```
# 省略起始索引
>>> mylist4[:-7:-2]           # 提取从结尾向左到倒数第 7 个元素前的所有元素，步长为 2
[100, 80, 60]
# 省略终止索引
>>> mylist4[6:]               # 提取从第 7 个元素到列表右端最后一个元素之间的所有元素
[70, 80, 90, 100]
# 同时省略起始和终止索引
>>> mylist4[::-2]             # 提取从右端开始到左端之间的全体元素，步长为 2
[100, 80, 60, 40, 20]
>>> mylist4[::-1]             # 提取从右端开始到左端之间的全体元素，步长为 1，即列表反转
[100, 90, 80, 70, 60, 50, 40, 30, 20, 10]
```

与列表索引提取方式不同，切片操作无须担心传入的索引超出列表索引范围，如果传入的索引小于列表第 1 个元素的负索引，切片操作会将其当作 0；若大于列表最后一个元素的正索引，则将其当作–1。注意，在这种情况下，切片操作包含终止元素。另外，当切片操作从起始索引根据提取方向无法达到终止索引时，Python 将返回一个空列表，如代码 3-8 所示。

代码 3-8　传入索引

```
>>> mylist4[3:100:2]                    # 提取从第 4 个元素到右端之间的全体元素，步长为 2
[40, 60, 80, 100]
>>> mylist4[-5:-20:-1]                  # 提取从倒数第 5 个元素到左端之间的全体元素
[60, 50, 40, 30, 20, 10]
>>> mylist4[6:2]                        # 提取从第 7 个元素向右到第 3 个元素之间的所有元素
[]
```

3.2.4　掌握列表常用函数和方法

Python 的列表类型包含丰富灵活的列表方法，而且 Python 中也有很多函数支持对列表对象进行操作，可以对列表对象进行更复杂的处理，一般常用的处理包括对列表对象进行元素的增添、删除、修改、查询等。

1．增添列表元素

使用列表方法 append、extend 和 insert 向列表对象中增添元素，这 3 种方法有各自的特点。

（1）append

向 append 传入需要添加到列表对象的一个元素，则该元素会被追加到列表尾部，如代码 3-9 所示。注意，append 一次只能追加一个元素。

代码 3-9　追加元素

```
>>> month=['January','February','March','April','May','June']
>>> month.append('July')                # 使用 append 函数向列表尾部追加元素
>>> month                               # 查看列表内容
['January', 'February', 'March', 'April', 'May', 'June', 'July']
```

（2）extend

使用 extend 能够将另一个列表添加到列表末尾，相当于两个列表进行拼接。类似于字符串拼接，两个列表对象也可以通过加号（＋）进行拼接，而且 extend 得到的效果与使用自增运算（＋＝）相同，如代码 3-10 所示。

代码 3-10　追加多个元素

```
>>> month_copy= month.copy()            # 创建一个列表对象 month 的副本，理由稍后解释
>>> month_copy
['January', 'February', 'March', 'April', 'May', 'June', 'July']
>>> others=['August','September','November','December']
>>> month.extend(others)                # 使用 extend 函数将两个列表进行合并
>>> month
['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August',
'September', 'November', 'December']
>>> month_copy+=others                  # 对副本进行自增运算
>>> month_copy
```

```
['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August',
'September', 'November', 'December']
```

（3）insert

类似于 append，使用 insert 也能够向列表中添加一个元素。不同的是，insert 可以在指定位置添加，就像在列表某个位置插入一个元素一样。只要向 insert 中传入插入位置和要插入的元素，即可在列表中的相应位置添加该元素。若插入位置超出列表尾端，则会插入列表最后，这相当于 append 的效果，如代码 3-11 所示。

代码 3-11　插入元素

```
>>> month.insert(9,'October')                    # 在列表第 10 个位置上插入元素
>>> month
['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August',
'September', 'October', 'November', 'December']
>>> month.insert(20,'None')                      # 插入位置超出列表尾端
>>> month
['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August',
'September', 'October', 'November', 'December', 'None']
```

2．删除元素

代码 3-11 中的列表对象 month 包含了一个与其他元素格格不入的元素“None”，为了让 month 只包含表示月份的字符串，需要把元素“None”从列表对象中删除，具体方法如下所述。

（1）使用 del 语句删除列表元素

在 Python 中，使用 del 语句可以将对象删除，实质上 del 语句是赋值语句（=）的逆过程，若把赋值语句看作是“向对象贴变量名标签”，则 del 语句就是“将对象上的标签撕下来”，即将一个对象与它的变量名进行分离操作。使用 del 语句可以将列表中提取出的元素删除，如代码 3-12 所示。

代码 3-12　使用 del 语句删除元素

```
>>> month_copy=month.copy()                      # 创建一个列表对象 month 副本
>>> del month_copy[-1]                           # 删除副本中最后一个元素
>>> month_copy
['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August',
'September', 'October', 'November', 'December']
```

（2）使用 pop 语句删除列表元素

利用元素位置可以对元素进行删除操作。将元素索引传入 pop 语句中，将会获取该元素，并将其在列表中删除，相当于把列表中的元素抽离出来。若不指定元素位置，pop 语句将默认使用索引–1，如代码 3-13 所示。

代码 3-13　使用 pop 语句删除元素

```
>>> month_copy=month.copy()          # 创建一个列表对象 month 副本
```

```
>>> month_copy.pop(3)                    # 获取并删除第 4 个元素
'April'
>>> del_element=month_copy.pop()         # 将最后一个元素赋值给一个变量并在副本中删除
>>> del_element                          # 查看删除元素
'None'
>>> month_copy                           # 查看副本
['January', 'February', 'March', 'May', 'June', 'July', 'August', 'September',
'October', 'November', 'December']
```

（3）使用 remove 语句删除列表元素

除了利用元素位置进行元素删除外，还可以将指定元素进行删除。将指定元素传入 remove 语句，则列表中第一次出现的该元素会被删除，如代码 3-14 所示。

代码 3-14　使用 remove 语句删除元素

```
>>> month.remove('None')                 # 删除列表中的元素'None'
>>> month
['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August',
'September', 'October', 'November', 'December']
```

3. 修改列表元素

列表对象 month 现在已经包含 12 个月的英文字符串，或许这些字符串显得过长，可以将月份变为缩写形式，这时需要对列表元素进行修改。

由于列表是可变的，修改列表元素最简单的方法是提取该元素并进行赋值操作，如代码 3-15 所示。

代码 3-15　修改列表元素

```
>>> month[0]='Jan'                       # 将第 1 个元素改为缩写形式
>>> month
['Jan', 'February', 'March', 'April', 'May', 'June', 'July', 'August',
'September', 'October', 'November', 'December']
```

前面方法的处理都是直接作用在列表对象上，而且会创建一些所谓的“副本”进行处理，下面将解释创建“副本”的理由。对于可变类型的数据结构，直接在对象上进行元素的增删改查等修改操作，处理结果将直接影响对象本身，如代码 3-16 所示。

代码 3-16　修改操作作用于对象

```
>>> a=[1,2,3,4]                          # 变量名 a 指向列表对象[1,2,3,4]
>>> b=a                                  # 变量名 b 也指向列表对象[1,2,3,4]
>>> a.append(5)                          # 列表尾端追加元素 5
>>> a
[1, 2, 3, 4, 5]
>>> b                                    # 通过变量名 b 查看列表
[1, 2, 3, 4, 5]
```

代码 3-16 展示了修改操作会直接作用在对象上，列表对象有 a 和 b 两个变量名，通过变量名 a 对列表对象进行修改，此时列表对象的内容发生改变，所以无论通过变量名 a 还是变量名 b 来查看列表对象，结果都是一样的。如果不希望修改操作直接作用于列表对象本身，可以使用列表的 copy 方法创建一个完全一样的“副本”，将修改操作作用在“副本”上，列表本身并不会发生变化。实质上，这个“副本”已经是另一个列表对象，只是内容与原列表对象完全相同而已。除了 copy 方法外，使用切片操作和 list 函数也能达到同样效果，如代码 3-17 所示。

代码 3-17　copy 方法操作

```
>>> a=[10,20,30,40,50]
>>> b=a.copy()                                          # 使用 copy 方法创建副本
>>> c=a[:]                                              # 使用切片操作创建副本
>>> d=list(a)                                           # 使用 list 函数创建副本
>>> id(a),id(b),id(c),id(d)                             # 查看各变量对象 id
(2617832796104, 2617832795848, 2617832794568, 2617832795592)
>>> b[2]='three'                                        # 修改副本第 3 个元素
>>> b
[10, 20, 'three', 40, 50]
>>> a                                                   # 原列表并没有发生变化
[10, 20, 30, 40, 50]
>>> c
[10, 20, 30, 40, 50]
>>> d
[10, 20, 30, 40, 50]
```

4. 查询列表元素位置

元素查询也是处理列表的重要操作，可以利用列表方法 index 来查询指定元素在列表中第 1 次出现的位置索引。若列表不包含指定元素，则会出现错误提示。对于判断列表是否包含某个元素，可以使用 Python 中的 in 函数，具体格式为“元素 in 列表对象”。若元素至少在列表中出现过一次，返回 True，否则返回 False，如代码 3-18 所示。

代码 3-18　查询列表元素位置

```
>>> letter=['A','B','A','C','B','B','C','A']
>>> letter.index('C')                          # 查询元素“C”在列表中第 1 次出现的位置
3
# 使用 in 函数判断列表是否包含元素
>>> 'A' in letter
True
```

5. 其他常用操作

以上是关于列表的重要的常用处理方法，这是熟练掌握列表类型数据结构的重要基础。列表的操作和应用非常丰富，可以实现更加高级复杂的处理，有兴趣的读者可以查阅相关

资料进行深入学习。下面再介绍其他几个比较常用的列表方法，如表 3-1 所示。

表 3-1　列表常用操作

列表方法和函数	说　明
list.count	记录某个元素在列表中出现的次数
list.sort	对列表中的元素进行排序，默认为升序，可以通过参数 reverse=True 进行降序排序。结果会改变原列表内容
sorted	与 list.sort 作用一样，但不改变原列表内容
list.reverse	反转列表中的各元素
len	获得列表长度，即元素个数
+	将两个列表合并为一个列表
*	重复合并同一个列表多次

表 3-1 中列举的方法应用示例如代码 3-19 所示。

代码 3-19　列表常用操作

```
# 使用 count 函数进行元素计数
>>> letter=['B','A','C','D','A','C','D','A']
>>> letter.count('A')                    # 获取元素'A'在列表中出现的次数
3
# 使用 sort 函数和 sorted 函数对列表进行排序
>>> sorted(letter)                       # 使用 sorted 函数对列表进行排序，不改变列表
['A', 'A', 'A', 'B', 'C', 'C', 'D', 'D']
>>> letter
['B', 'A', 'C', 'D', 'A', 'C', 'D', 'A']
>>> letter.sort()                        # 使用列表方法 sort 进行排序，改变列表内容
>>> letter
['A', 'A', 'A', 'B', 'C', 'C', 'D', 'D']
>>> letter.sort(reverse=True)            # 对列表进行倒序排序
>>> letter
['D', 'D', 'C', 'C', 'B', 'A', 'A', 'A']
# 使用 reverse 函数反转列表
>>> season=['spring','summer','autumn','winter']
>>> season.reverse()                     # 反转列表
>>> season
['winter', 'autumn', 'summer', 'spring']
# 使用函数 len 获取列表长度
>>> len(season)
4
# 使用列表加法合并两个列表
```

```
>>> [1,2,3]+[4,5,6]
[1, 2, 3, 4, 5, 6]
# 使用列表乘法重复合并列表
>>> [10,20,30,40]*3
[10, 20, 30, 40, 10, 20, 30, 40, 10, 20, 30, 40]
```

3.2.5 任务实现

根据任务分析，本任务的具体实现过程可以参考如下操作。

（1）使用方括号[ ]创建列表对象[110,‘dog’,‘cat’,120,‘apple’]。

（2）使用 insert 函数在元素之间插入空列表。

（3）使用 remove 函数删除字符串元素 “apple”。

（4）使用 index 函数查询指定元素位置。

（5）通过索引访问提取数字元素，并用自乘操作进行赋值修改。

（6）使用 print 函数打印修改后列表的内容。

参考代码如任务实现 3-1 所示。

任务实现 3-1

```
# -*-coding:utf-8-*-

task_list=[110,'dog','cat',120,'apple']
task_list.insert(2,[])                          # 插入空列表
task_list.remove('apple')                       # 删除元素
num_index=task_list.index(110)                  # 查询元素位置
task_list[num_index]*=10                        # 将查询出来的元素进行自乘运算,并赋值修改
num_index=task_list.index(120)
task_list[num_index]*=10
print(task_list)                                # 查看修改后的列表对象
```

任务 3.3 转换一个列表为元组（tuple）并进行取值操作

任务描述

列表和元组都是序列结构，它们本身相似，但又有一点不同的地方。将列表[‘pen’,‘paper’,10,False,2.5]转换为元组类型，并提取出当中的布尔值。

任务分析

可通过如下步骤实现上述任务。

（1）使用方括号创建列表[‘pen’,‘paper’,10,False,2.5]，并赋值给变量。

（2）查看变量的数据类型。

（3）将变量转变成 tuple 类型。

（4）查看变量的数据类型，确定是否转换为元组。

（5）查询元组中元素 False 的位置。

（6）根据获得的位置提取元素。

3.3.1　区分元组和列表

在前面介绍列表的过程中，已经介绍过元组类型数据结构。元组与列表非常相似，都是有序元素的集合，并且可以包含任意类型元素。不同的是，元组是不可变的，这说明元组一旦创建后就不能修改，即不能对元组对象中的元素进行赋值修改、增加、删除等操作。列表的可变性可能更方便处理复杂问题，例如更新动态数据等，但很多时候不希望某些处理过程修改对象内容，例如敏感数据，这时就需要用到元组的不可变性。

3.3.2　创建元组

类似于列表，创建元组只需传入有序元素即可，常用的创建方法有使用圆括号( )创建和使用 tuple 函数创建。

1. 使用圆括号( )创建

使用圆括号将有序元素括起来，并用逗号隔开，可以创建元组。注意，这里的逗号是必须存在的，即使元组当中只有一个元素，后面也需要有逗号。在 Python 中定义元组的关键是当中的逗号，圆括号却可以省略。当输出元组时，Python 会自动加上一对圆括号。同样，若不向圆括号中传入任何元素，则会创建一个空元组，如代码 3-20 所示。

代码 3-20　()创建元组

```
>>> mytuple1=(1,2.5,('three','four'),[True,5],False)   # 使用圆括号( )创建元组
>>> mytuple1
(1, 2.5, ('three', 'four'), [True, 5], False)
>>> mytuple2=2,True,'five',3.5                          # 省略圆括号
>>> mytuple2
(2, True, 'five', 3.5)                                  # 结果自动加上圆括号
>>> empty_tuple=()                                      # 创建空元组
>>> empty_tuple
()
```

2. 使用 tuple 函数创建

tuple 函数能够将其他数据结构对象转换成元组类型。先创建一个列表，将列表传入 tuple 函数中，再转换成元组，即可实现创建元组。

使用 tuple 函数对代码 3-20 中的元组对象进行再次创建。注意，在 tuple 函数中传入元组需要加上圆括号，如代码 3-21 所示。

代码 3-21　使用 tuple 函数创建元组

```
>>> mytuple1=tuple([1,2.5,('three','four'),[True,5],False])
# 使用 tuple 函数将列表转换为元组
>>> mytuple1
(1, 2.5, ('three', 'four'), [True, 5], False)
>>> mytuple2=tuple((2,True,'five',3.5))
```

```
>>> mytuple2
(2, True, 'five', 3.5)
>>> empty_tuple=tuple()
>>> empty_tuple
()
```

通过代码 3-20 和代码 3-21 可以看出，创建元组与创建列表的方法极其类似，只是元组使用圆括号来包括元素，而列表使用方括号。

3.3.3 掌握元组常用函数和方法

元组是不可变的，类似于对列表元素的增添、删除、修改等处理都不能作用在元组对象上，但元组属于序列类型数据结构，因此可以在元组对象上进行元素索引访问提取和切片操作。特别的，对于元组元素的提取，可以使用元组解包简化赋值操作。

1. 元组元素提取

利用序列的索引进行访问提取和切片操作，可以提取元组中的元素和切片。

（1）元组索引访问提取

与列表索引访问提取元素一样，只要传入元素索引，就能够获得对应元素。同样，若传入的索引超出元组索引范围，结果会返回一个错误，如代码 3-22 所示。

代码 3-22 元组索引访问提取

```
>>> mytuple3=('China','America','England','France')
>>> mytuple3[0]                          # 提取元组第 1 个元素
'China'
>>> mytuple3[10]                         # 传入的索引超出元组索引范围
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
```

（2）元组切片操作提取

类似的，通过操作也可以获取元组的切片，并且无须考虑超出索引范围的问题，如代码 3-23 所示。

代码 3-23 元组切片操作提取

```
>>> mytuple3[-2::-1]                     # 提取元组倒数第 2 个元素到左端之间的所有元素
('England', 'America', 'China')
>>> mytuple3[1:10]                       # 超出元素索引范围
('America', 'England', 'France')
```

2. 元组解包

将元组中的各个元素赋值给多个不同变量的操作通常称为元组解包，使用格式为 obj_1,obj_2,…,obj_n=tuple。由于创建元组时可以省略圆括号，因此元组解包可以看成是多条赋值语句的集合。可见，Python 在赋值操作上的处理非常灵活，一句简单的元组解包代码就可以实现多条赋值语句的功能，如代码 3-24 所示。

代码 3-24　元组解包

```
>>> A,B,C,D=mytuple3                    # 将元组中的各元素分别赋值给对应变量
>>> A
'China'
>>> C
'England'
>>> x,y,z=1,True,'one'                  # 利用元组解包进行多个变量赋值
>>> x
1
>>> z
'one'
```

3. 元组常用方法和函数

相比于列表，由于元组无法修改元素，所以元组的方法和函数相对较少，但仍然能够对元组进行元素位置查询等操作。表 3-2 列出了一些常用的元组方法和函数。

表 3-2　元组常见操作

元组方法和函数	说　明
tuple.count	记录某个元素在元组中出现的次数
tuple.index	获取元素在元组当中第 1 次出现的位置索引
sorted	创建对元素进行排序后的列表
len	获取元组长度，即元组元素个数
+	将两个元组合并为一个元组
*	重复合并同一个元组为一个更长的元组

表 3-2 给出的方法的应用示例如代码 3-25 所示。

代码 3-25　元组常用操作

```
# 使用 count 函数进行元素计数
>>> mytuple4=('A','D','C','A','C','B','B','A')
>>> mytuple4.count('B')
2
# 使用 index 函数获取元素在元组中第一次出现的位置索引
>>> mytuple4.index('C')
2
# 使用 sorted 函数对元组元素进行排序
>>> sorted(mytuple4)
['A', 'A', 'A', 'B', 'B', 'C', 'C', 'D']
# 使用 len 函数获取元组长度
>>> len(mytuple4)
```

```
8
# 使用元组加法合并两个元组
>>> (1,2,3)+(4,5,6)
(1, 2, 3, 4, 5, 6)
# 使用元组乘法重复合并元组
>>> (10,20,30,40)*3
(10, 20, 30, 40, 10, 20, 30, 40, 10, 20, 30, 40)
```

3.3.4 任务实现

根据任务分析，本任务的具体实现过程可以参考如下操作。

（1）使用方括号创建列表对象['pen','paper',10,False,2.5]，并赋值给变量。

（2）使用 type 函数查看此时变量的数据类型。

（3）使用 tuple 函数将变量转换成元组类型。

（4）再次使用 type 函数确定是否完成转换。

（5）使用元组方法 index 分别查询元素 False 的位置索引。

（6）提取元素 False 并赋值。

参考代码如任务实现 3-2 所示。

任务实现 3-2

```
# -*-coding:utf-8-*-

task_tuple=['pen','paper',10,False,2.5]
type(task_tuple)
task_tuple=tuple(task_tuple)                    # 转换列表对象为元组类型
type(task_tuple)                                # 查看对象的数据类型
Index=task_tuple.index(False)                   # 查询元素位置索引
bool=task_tuple[Index]                          # 提取元组元素
print(bool)                                     # 查看提取元素
```

任务 3.4 创建一个字典（dict）并进行增删改查操作

任务描述

创建一个字典{'Math'：96,'English'：86,'Chinese'：95.5,'Biology'：86,'Physics'：None}，代表某学生一些学科的考试成绩，向字典当中添加历史成绩（88 分），并删除没有成绩的科目，然后将字典中的成绩四舍五入后取整，最后查看该学生的数学成绩。

任务分析

通过如下步骤可实现上述任务。

（1）创建字典{'Math':96,'English':86,'Chinese':95.5,'Biology':86,'Physics':None}。

（2）在字典当中添加键值对{'History':88}。

（3）删除{'Physics':None}键值对。

（4）将键“Chinese”所对应的值 95.5 进行四舍五入后取整。

（5）查询键“Math”的对应值。

3.4.1　了解字典的概念与特性

很多时候，数据对应的元素之间的顺序是无关紧要的，因为各元素都具有特别的意义，例如存储一些朋友的手机号码，此时用序列来存储数据并不是一个好的选择，Python 提供了一个很好的解决方案——使用字典数据类型。

在 Python 中，字典是属于映射类型的数据结构。字典包含以任意类型的数据结构作为元素的集合，同时各元素都具有与之对应且唯一的键，字典主要通过键来访问对应的元素。字典与列表、元组有所不同，后两者使用索引来对应元素，而字典的元素都拥有各自的键，每个键值对都可以看成是一个映射对应关系。此外，元素在字典中没有严格的顺序关系。由于字典是可变的，所以可以对字典对象进行元素的增删改查等基本操作。

3.4.2　解析字典的键与值

字典中的每个元素都具有对应的键，元素就是键所对应的值，键与值共同构成一个映射关系，即键→值，每个键都可以映射到相应的值，就像身份证号码可以映射到名字一样。键和值的这种映射关系在 Python 中具体表示为 key:value，键和值之间用冒号隔开，这里称为键值对，字典中会包含多组键值对。注意，字典中的键必须使用不可变数据类型的对象，例如数字、字符串、元组等，并且键是不允许重复的；而值则可以是任意类型的，且在字典中可以重复。

3.4.3　创建字典

字典中最关键的信息是含有对应映射关系的键值对，创建字典需要将键和值按规定格式传入特定的符号或函数中，Python 中常用的两种创建字典的基本方法分别是使用花括号{ }创建和使用函数 dict 创建。

1. 使用花括号{ }创建

只要将字典中的一系列键和值按键值对的格式（key:value，即键:值）传入花括号{ }中，并以逗号将各键值对隔开，即可实现创建字典，具体创建格式如下。

```
dict={key_1:value_1,key_2:value_2,…,key_n:value_n}
```

若在花括号{ }中不传入任何键值对，则会创建一个空字典。如果创建字典时重复传入相同的键，因为键在字典中不允许重复，所以字典最终会采用最后出现的重复键的键值对，如代码 3-26 所示。

代码 3-26　使用花括号{ }创建字典

```
>>> mydict1={'myint':1,'myfloat':3.1415,'mystr':'name',
# 使用花括号创建字典
...          'myint':100,'mytuple':(1,2,3),'mydict':{}}
# 对于重复，键采用最后出现的对应值
>>> mydict1
{'myint': 100, 'myfloat': 3.1415, 'mystr': 'name', 'mytuple': (1, 2, 3), 'mydict':
```

```
{}}
>>> empty_dict={}                                          # 创建空字典
>>> empty_dict
{}
```

2. 使用 dict 函数创建

创建字典的另一种方法就是使用 dict 函数。Python 中的 dict 函数的作用实质上主要是将包含双值子序列的序列对象转换为字典类型，其中各双值子序列中的第 1 个元素作为字典的键，第 2 个元素作为对应的值，即双值子序列中包含了键值对信息。所谓双值子序列，实际上就是只包含两个元素的序列，例如只包含两个元素的列表[‘name’,’Lily’]、元组(‘age’,18)、仅包含两个字符的字符串’ab’等。将字典中的键和值组织成双值子序列，然后将这些双值子序列组成序列，例如组成元组([‘name’,’Lily’],(‘age’,18),’ab’)，再传入 dict 函数中，即可转换为字典类型，得到字典对象。除了通过转换方式创建字典外，还可以直接向 dict 函数传入键和值进行创建，其中须通过 “=” 将键和值隔开。注意，这种创建方式不允许键重复，否则会返回错误，具体格式如下。

```
dict(key_1=value_1,key_2=value_2,…,key_n=value_n)。
```

对 dict 函数不传入任何内容时，就可创建一个空字典，如代码 3-27 所示。

代码 3-27　使用 dict 函数创建字典

```
>>> mydict1=dict([('myint',1),('myfloat',3.1415),('mystr','name'),
...('myint',100),('mytuple',(1,2,3)),('mydict',{})])
# 使用 dict 函数转换列表对象为字典
>>> mydict1
{'myint': 100, 'myfloat': 3.1415, 'mystr': 'name', 'mytuple': (1, 2, 3), 'mydict':
{}}
>>> mydict2=dict(zero=0,one=1,two=2)
>>> mydict2
{'zero': 0, 'one': 1, 'two': 2}
>>> empty_dict=dict()
>>> empty_dict
{}
```

代码 3-27 中涵盖了创建字典的基本方法，并且能够看到字典中可以包含各种数据类型对象，字典中的值都可以对应到有具体意义的键，可见字典是一种非常灵活和重要的数据结构。

3.4.4　提取字典元素

与序列类型不同，字典作为映射类型数据结构，并没有索引的概念，也没有切片操作等处理方法，字典中只有键和值对应起来的映射关系，因此字典元素的提取主要是利用这种映射关系来实现。通过在字典对象后紧跟方括号[ ]包括的键可以提取相应的值，具体使用格式为 dict[key]，即字典[键]。同时应注意，传入的键要存在于字典中，否则会返回一个错误，如代码 3-28 所示。

代码 3-28　提取字典元素

```
>>>mydict3={'spring':(3,4,5),'summer':(6,7,8),'autumn':(9,10,11),'winter':(1
2,1,2)}
>>> mydict3['autumn']                    # 提取键为'autumn'的对应值
(9, 10, 11)
>>> mydict3['Spring']                    # 提取字典中不存在的键'Spring'所对应的值
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Spring'
```

为避免出现上述传入键不存在而导致出错的现象，Python 提供了两种处理方法。

1. 提取前使用 in 语句测试键是否存在

错误主要是因为传入的键不存在而导致的，因此在传入键之前，尝试去检查字典中是否包含这个键；若不存在，则不进行提取操作。这种功能具体可以使用 in 语句来实现，如代码 3-29 所示。

代码 3-29　使用 in 语句测试键是否存在

```
>>> 'Spring' in mydict3                          # 使用 in 检查传入键是否存在
False
```

2. 使用字典方法 get

字典方法 get 能够灵活地处理元素的提取，向 get 函数传入需要的键和一个代替值即可，无论键是否存在。若只传入键，当键存在于字典中时，函数会返回对应的值；当键不存在时，函数会返回 None，屏幕上什么都不显示。若同时也传入代替值，当键存在时，返回对应值；当键不存在时，返回这个传入的代替值，而不是 None，如代码 3-30 所示。

代码 3-30　使用函数 get 提取元素

```
>>> mydict3.get('summer')                        # 传入存在的键并返回对应值
(6, 7, 8)
>>> mydict3.get('Spring')                        # 仅传入不存在的键，不显示任何东西
>>> print(mydict3.get('Spring'))                 # 打印函数返回的结果
None
>>> mydict3.get('Spring','Not in this dict')     # 传入不存在的键并返回代替值
'Not in this dict'
```

3.4.5　字典常用函数和方法

在 Python 的内置数据结构当中，列表和字典是最为灵活的数据类型。类似于列表，字典属于可变数据类型，因此字典也含有丰富而且功能强大的方法和函数。下面将介绍如何对字典元素进行增添、删除、修改和查询等最常用的处理，由于上述这些字典处理会直接作用在字典对象上，而且各种处理方式包含多种方法，为了能更好地展示各种方法的处理效果，与列表一样，字典中也有 copy 方法，其作用也是复制字典内容并创建一个副本对象，

下面的示例将会利用 copy 方法取得副本对象后再进行处理。

1. 增添字典元素

直接利用键访问赋值的方式，可以向字典中增添一个元素。若需要添加多个元素，或将两个字典内容合并，可以使用 update 方法。接下来将具体介绍这两种元素增添的方法。

（1）使用键访问赋值增添

利用字典元素提取方法传入一个新的键，并对这个新键进行赋值操作，即 dict_name[newkey]=new_value，字典中会产生新的键值对，这种赋值操作可能会因为键不存在而出现错误，如代码 3-31 所示。

代码 3-31　使用键访问赋值增添

```
>>> country=dict(China='Beijing',                    # 使用 dict 函数创建字典
...              America='Washington',
...              Britain='London',
...              French='Paris',
...              Canada='Ottawa')
>>> country_copy=country.copy()                      # 创建一个副本字典对象
>>> country_copy['Russian']='Moscow'                 # 增添元素
>>> country_copy
{'China': 'Beijing', 'America': 'Washington', 'Britain': 'London', 'French':
'Paris', 'Canada': 'Ottawa', 'Russian': 'Moscow'}
```

（2）update

字典方法 update 能将两个字典中的键值对进行合并，传入字典中的键值对会复制添加到调用函数的字典对象中。若两个字典中存在相同键，传入字典中的键所对应的值会替换掉调用函数字典对象中的原有值，实现值更新的效果，如代码 3-32 所示。

代码 3-32　键值对合并

```
>>> others=dict(Australia='Canberra',
...             Japan='tokyo',
...             Canada='OTTAWA')
>>> country.update(others)                  # 使用 update 函数增添多个元素
>>> country
{'China': 'Beijing', 'America': 'Washington', 'Britain': 'London', 'French':
'Paris', 'Canada': 'OTTAWA', 'Australia': 'Canberra', 'Japan': 'tokyo'}
```

2. 删除字典元素

使用 del 语句可以删除某个键值对。另外，字典也包含 pop 函数，只要传入键，函数就能将对应的值从字典中抽离，不同的是必须传入参数。若需要清空字典内容，可以使用字典方法的 clear 函数，结果返回空字典。

（1）使用 del 语句删除字典元素

使用 del 语句删除元素的具体格式为 del dict_name[key]，如代码 3-33 所示。

代码 3-33　使用 del 语句删除字典元素

```
>>> country_copy=country.copy()
>>> del country_copy['Canada']                    # 使用 del 删除副本字典中的元素
>>> country_copy
{'China': 'Beijing', 'America': 'Washington', 'Britain': 'London', 'French':
'Paris', 'Australia': 'Canberra', 'Japan': 'tokyo'}
```

（2）使用 pop 语句删除字典元素

向 pop 语句传入需要删除的键，则会返回对应的值，并在字典当中移除相应的键值对。若将函数返回的结果赋值给变量，就相当于从字典当中抽离出了值，如代码 3-34 所示。

代码 3-34　使用 pop 语句删除字典元素

```
>>> old_value=country.pop('Canada')          # 将键对应的值赋值给变量，并删除键值对
>>> old_value
'OTTAWA'
>>> country
{'China': 'Beijing', 'America': 'Washington', 'Britain': 'London', 'French':
'Paris', 'Australia': 'Canberra', 'Japan': 'tokyo'}
```

（3）使用 clear 删除字典元素

clear 会删除字典中的所有元素，最终会返回一个空字典，如代码 3-35 所示。

代码 3-35　使用 clear 删除字典的所有元素

```
>>> country_copy=country.copy()
>>> country_copy.clear()                     # 清空副本字典内容
>>> country_copy
{}
```

3. 修改字典元素

现在，字典 country 已经被删除了字母全为大写的值，但发现还有一个值全为小写，为统一字典中各元素的格式，需要对这个值进行修改。

为修改字典中的某个元素，同样可以使用键访问赋值来修改，格式为 dict_name[key]= new_value。由此可以看出，赋值操作在字典当中非常灵活，无论键是否存在于字典中，所赋予的新值都会覆盖或增添到字典中，这很大程度上方便了对字典对象的处理，如代码 3-36 所示。

代码 3-36　修改字典元素

```
>>> country['Japan']='Tokyo'                      # 直接将新值赋值给对应元素
>>> country
{'China': 'Beijing', 'America': 'Washington', 'Britain': 'London', 'French':
'Paris', 'Australia': 'Canberra', 'Japan': 'Tokyo'}
```

4. 查询和获取字典元素信息

在实际应用当中，往往需要查询某个键或值是否在字典当中，除了可以使用字典元素

提取的方式进行查询外，还可以使用 Python 中的 in 进行判断。字典方法中有 3 种方式可以用于提取键值信息。

（1）keys：用于获取字典中的所有键。

（2）values：用于获取字典中的所有值。

（3）itmes：得到字典中的所有键值对。

这 3 种方式所返回的结果是字典中键、值或键值对的迭代形式，都可以通过 list 函数将返回结果转换为列表类型，同时可以配合 in 的使用，判断值和键值对是否存在于字典当中，如代码 3-37 所示。

代码 3-37　提取键值信息

```
# 判断键是否存在于字典当中
>>> 'Canada' in country
False
# 获取所有键
>>> all_keys=country.keys()                    # 使用 keys 函数得到全部键
>>> all_keys
dict_keys(['China', 'America', 'Britain', 'French', 'Australia', 'Japan'])
# 判断值是否存在于字典当中
>>> all_values=country.values()                # 使用 values 函数得到全部值
>>> all_values
dict_values(['Beijing', 'Washington', 'London', 'Paris', 'Canberra', 'Tokyo'])
>>> 'Beijing' in all_values                    # 判断字典是否包含值
True
>>> list(all_values)                           # 将值的迭代形式转换为列表
['Beijing', 'Washington', 'London', 'Paris', 'Canberra', 'Tokyo']
# 判断键值对是否存在于字典当中
>>> all_items=country.items()                  # 使用 items 函数得到全部键值对
>>> all_items
dict_items([('China', 'Beijing'), ('America', 'Washington'), ('Britain',
'London'), ('French', 'Paris'), ('Australia', 'Canberra'), ('Japan', 'Tokyo')])
>>> ('America','Washington') in all_items      # 判断字典是否包含键值对
True
>>> list(all_items)                            # 将键值对迭代形式转换为列表
[('China', 'Beijing'), ('America', 'Washington'), ('Britain', 'London'),
('French', 'Paris'), ('Australia', 'Canberra'), ('Japan', 'Tokyo')]
```

以上便是字典所使用的常用处理方法，具体实现了字典元素的增删改查等重要操作。这里所介绍的字典方法和函数可以实现对字典的一些简单处理，如果需要对字典进行更复杂、更高级的处理，就需要将这些方法进行灵活组合运用。例如利用值来查询所有与之对应的键，如代码 3-38 所示。

代码 3-38　利用值查询键

```
# 提取字典中值为 True 所对应的键
>>> test={'A':100,'B':300,'C':True,'D':200}
>>> keys=list(test.keys())                          # 提取字典中的所有键
>>> values=list(test.values())                      # 提取字典中的所有值
>>> keys
['A', 'B', 'C', 'D']
>>> values
[100, 300, True, 200]        # 提取的全体键和值的索引正好是一一对应，构成原字典中的键值对
>>> keys[values.index(True)]    # 故可利用值 True 的索引来提取对应的键
'C'
```

3.4.6　任务实现

根据任务分析，本任务的具体实现过程可以参考如下操作。

（1）使用花括号{ }创建某学生各科成绩组成的字典对象，并赋值给变量。

（2）使用键访问赋值方式向字典增添键值对{'History':88}。

（3）使用 del 语句删除键“Physics”。

（4）利用键“Chinese”访问对应元素，并使用 round 函数进行四舍五入后取整。

（5）将取整结果进行赋值来覆盖字典中键“Chinese”的对应值。

（6）直接使用键“Math”查询对应值。

参考代码如任务实现 3-3 所示。

任务实现 3-3

```
# -*-coding:utf-8-*-

score={'Math':96,'English':86,
      'Chinese':95.5,'Biology':86,
      'Physics':None}
score['History']=88                              # 增添键值对
del score['Physics']                             # 删除键值对
new_value=round(score['Chinese'])                # 将成绩进行四舍五入取整
score['Chinese']=new_value                       # 修改对应值
score['Math']                                    # 查看键的对应值
score                                            # 查看处理后的字典
```

任务 3.5　将两个列表转换为集合（set）并进行集合运算

任务描述

Python 内置了集合这一数据结构，与数学上的集合概念基本上是一致的。本任务拟将列表['apple','pear','watermelon','peach']和['pear','banana','orange','peach','grape']都转换为

集合，同时求出两个集合的并集、交集和差集。

通过如下步骤可实现上述任务。

（1）使用方括号创建列表[‘apple’,’pear’,’watermelon’,’peach’]并赋值给变量。

（2）使用 list 函数创建列表[‘pear’,’banana’,’orange’,’peach’,’grape’]并赋值给变量。

（3）将创建的两个列表对象转换为集合类型。

（4）对得到的两个集合对象求出并集。

（5）对得到的两个集合对象求出交集。

（6）求出两个集合的差集。

3.5.1 了解集合的概念与特性

Python 中的集合类型数据结构是将各不相同的不可变数据对象无序地集中起来的容器，就像是将值抽离，仅存在键的字典。类似于字典中的键，集合中的元素都是不可重复的，并且是属于不可变类型，元素之间没有排列顺序。集合的这些特性，使得它独立于序列和映射类型之外，Python 中的集合类型就相当于数学集合论中所定义的集合，人们可以对集合对象进行数学集合运算（并集、交集、差集等）。

3.5.2 创建集合

若按数据结构对象是否可变来分，集合类型数据结构包括可变集合与不可变集合。

1. 可变集合

可变集合对象是可变的，可对其进行元素的增添、删除等处理，处理结果直接作用在对象上。使用花括号{ }可以创建可变集合，这里与创建字典不同，传入的不是键值对，而是集合元素，注意，传入的元素对象必须是不可变的，即不能传入列表、字典甚至可变集合等。另外，可变集合的 set 函数能够将数据结构对象转换为可变集合类型，即将集合元素存储为一个列表或元组，再使用 set 函数转换为可变集合。在创建时，无须担心传入的元素是否重复，因为结果会将重复元素删除。若需要创建空集合，只能使用 set 函数且不传入任何参数进行创建，如代码 3-39 所示。

代码 3-39 创建可变集合

```
# 使用花括号创建可变集合
>>> myset1={'A','C','D','B','A','B'}
>>> myset1
{'C', 'D', 'B', 'A'}
# 使用函数 set 创建可变集合
>>> myset2=set([2,3,1,4,False,2.5,'one'])
>>> myset2
{False, 1, 2, 3, 4, 2.5, 'one'}
>>> empty_set=set()                                    # 创建空可变集合
```

```
>>> empty_set
set()
>>> type(empty_set)
<class 'set'>
```

2. 不可变集合

不可变集合对象属于不可变数据类型，不能对其中的元素进行修改处理。创建不可变集合的方法是使用 frozenset 函数。它与 set 函数用法一样，不同的是得出的结果是一个不可变集合。注意，元素必须为不可变数据类型。使用不可变集合作为元素，当 frozenset 函数不传入任何参数时，则会创建一个空不可变集合，如代码 3-40 所示。

代码 3-40　创建不可变集合

```
>>> myset3=frozenset([3,2,3,'one',frozenset([1,2]),True])
# 使用 frozenset 函数创建不可变集合
>>> myset3
frozenset({True, 2, 3, 'one', frozenset({1, 2})})
>>> empty_frozenset=frozenset()                                    # 创建空不可变集合
>>> empty_frozenset
frozenset()
>>> type(empty_frozenset)
<class 'frozenset'>
```

3.5.3　集合运算

集合是由互不相同的元素对象所构成的无序整体。集合包含多种运算，这些集合运算会获得满足某些条件的元素集合。常用的集合运算包括并集、交集、差集、异或集等，当需要获得两个集合之间的并集、交集、差集等元素集合时，这些集合运算能够获取集合之间的某些特殊信息。例如，学生 A 喜欢的体育运动的集合为{‘足球’,’游泳’,’羽毛球’,’乒乓球’}；而学生 B 喜欢的集合为{‘篮球’,’乒乓球’,’羽毛球’,’排球’}，要获取两个学生都喜欢的体育运动，或者除了学生 B 喜欢的运动项目外，还有哪些是 A 喜欢的，就可以通过集合运算来实现。

1. 并集

由属于集合 A 或集合 B 的所有元素组成的集合称为集合 A 和 B 的并集，数学表达式为 $A \cup B=\{x \mid x \in A$ 或 $x \in B\}$。并集与集合 A 和 B 之间的关系如图 3-2 所示，其中阴影部分即为并集。

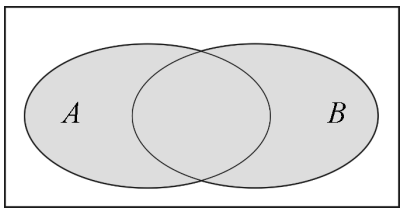


图 3-2　并集与集合 A 和 B 之间的关系

根据并集的数学定义，上述例子中，两学生的集合 A 和 B 的并集 $A \cup B$ 为{‘足球’,’游泳’,’羽毛球’,’乒乓球’,’篮球’,’排球’}，这表示学生 A 和 B 喜欢的运动项目都有哪些。在 Python 中可以使用符号“|”或者集合方法 union 函数来获得两个集合的并集，如代码 3-41 所示。

代码 3-41　求并集

```
>>> A={'足球','游泳','羽毛球','乒乓球'}
>>> B={'篮球','乒乓球','羽毛球','排球'}
>>> A|B                                          # 使用符号'|'获取并集
{'羽毛球', '排球', '乒乓球', '足球', '篮球', '游泳'}
>>> A.union(B)                                   # 使用集合方法 union 函数获取并集
{'羽毛球', '排球', '乒乓球', '足球', '篮球', '游泳'}
```

2. 交集

同时属于集合 A 和 B 的元素组成的集合称为集合 A 和 B 的交集，数学表达式为 $A \cap B=\{x \mid x \in A$ 且 $x \in B\}$。交集与集合 A 和 B 之间的关系如图 3-3 所示，其中阴影部分即为交集。

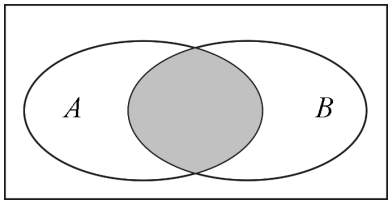


图 3-3　交集与集合 A 和 B 之间的关系

由交集的定义可知，学生 A 和 B 都喜欢的运动项目为集合{‘羽毛球’,’乒乓球’}。利用符号“&”或者集合方法 intersection 函数可以获取两个集合对象的交集，如代码 3-42 所示。

代码 3-42　求交集

```
>>> A&B                                          # 使用符号“&”获取交集
{'乒乓球', '羽毛球'}
>>> A.intersection(B)                            # 使用集合方法 intersection 函数获取交集
{'乒乓球', '羽毛球'}
```

3. 差集

属于集合 A 而不属于集合 B 中的元素所构成的集合即为 A 和 B 的差集，数学表达式为 $A-B=\{x \mid x \in A, x \notin B\}$。反过来，也有差集 $B-A=\{x \mid x \in B, x \notin A\}$。如图 3-4 所示，其中阴影部分即为差集 $A-B$。

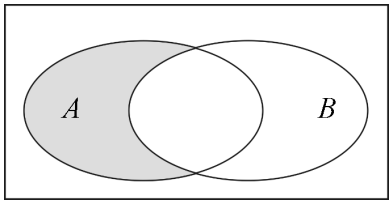


图 3-4　差集与集合 A 和 B 之间的关系

除 A、B 都喜欢的体育项目外，若需要知道学生 A 还喜欢哪些项目，则可以通过求差集 $A-B$ 来获取这种信息。在 Python 中可以简单地使用减号“—”来得到相应的差集，或者通过集合方法 difference 函数来实现，如代码 3-43 所示。

代码 3-43　求差集

```
>>> A-B                                          # 使用减号“—”来获取差集
{'足球', '游泳'}
>>> A.difference(B)                              # 使用集合方法 difference 函数获取差集
{'足球', '游泳'}
```

4. 异或集

属于集合 *A* 或集合 *B*，但不同时属于集合 *A* 和 *B* 的元素所组成的集合，称为集合 *A* 和 *B* 的异或集，其相当于（*A*∪*B*）—（*A*∩*B*）。异或集与集合 *A* 和 *B* 之间的关系如图 3-5 所示，其中阴影部分为异或集。

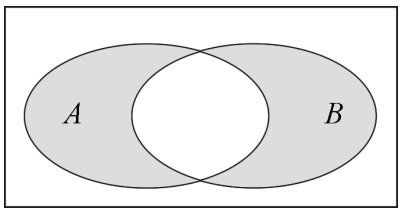


图 3-5　异或集与集合 *A* 和 *B* 之间的关系

通过求得例子中两集合 *A* 和 *B* 的异或集，可以得知两个学生都有哪些不相同的体育爱好，只要利用符号“^”或者集合方法 symmetric_difference 函数，即可求出两个集合对象的异或集，如代码 3-44 所示。

代码 3-44　求异或集

```
>>> A^B                                  # 获取异或集
{'足球', '排球', '游泳', '篮球'}
>>> A.symmetric_difference(B)   # 使用集合方法 symmetric_difference 函数获取异或集
{'足球', '排球', '游泳', '篮球'}
```

除上述基本集合运算外，集合之间的关系也是非常重要的。两个集合之间通常存在子集、真子集、超集、真超集等关系，它们揭示了集合之间的包含关系。例如，现在知道学生 C 喜欢的体育项目为{‘足球’,’乒乓球’,’游泳’}，是否能大致地认为学生 A 比学生 C 的体育爱好更广泛一点，此时可以使用集合关系进行判断。在 Python 中实现这些集合关系所用的函数和符号如表 3-3 所示。

表 3-3　集合关系常用函数和符号

集合关系函数和符号	函数说明
<= 或 issubset( )	判断一个集合是否为另一个集合的子集，即判断是否有 $A \subseteq B$ 的关系。如果是，则集合 *A* 中所有元素都是集合 *B* 中的元素
<	判断一个集合是否为另一个集合的真子集，即判断是否有 $A \subset B$ 的关系。如果是，则集合 *B* 中除了包含集合 *A* 中的所有元素，还包括 *A* 中没有的其他元素
>= 或 issuperset( )	判断一个集合是否为另一个集合的超集，即判断是否有 $A \supseteq B$ 的关系。如果是，则集合 *A* 包含了 *B* 中所有元素
>	判断一个集合是否为另一个集合的真超集，即判断是否有 $A \supset B$ 的关系。如果是，则集合 *A* 包含了 *B* 中的所有元素，也包含了 *B* 中没有的其他元素

表 3-3 列举的这些方法的应用示例如代码 3-45 所示。

代码 3-45　集合关系函数和符号的操作

```
>>> C={'足球','乒乓球','游泳'}
>>> C<=A                                    # 判断子集
True
>>> C.issubset(A)                           # 使用 issubset 函数判断子集
True
>>> C<A;A<A                                 # 判断真子集
True
False
>>> A>=C                                    # 判断超集
True
>>> A.issuperset(C)                         # 使用 issuperset 函数判断超集
True
>>> A>C;C>C                                 # 判断真超集
True
False
```

3.5.4　集合常用函数和方法

集合类型数据结构分为可变集合与不可变集合两种，与其他可变类型数据对象一样，对于可变集合对象，也可以进行元素的增添、删除、查询等处理，相关常用方法和函数如表 3-4 所示。

表 3-4　可变集合常用方法和函数

可变集合方法和函数	说　明
set.add	向可变集合当中增添一个元素
set.update	向可变集合增添其他集合的元素，即合并两个集合
set.pop	删除可变集合中的一个元素，当集合对象是空集时，则返回错误
set.remove	删除可变集合中指定的一个元素
set.clear	清空可变集合中的所有元素，返回空集
in	使用 Python 中的 in 方法可以查询元素是否存在于集合当中
len	获取集合当中元素的个数
set.copy	复制可变集合的内容并创建一个副本对象

表 3-4 列举的方法的应用示例如代码 3-46 所示。

代码 3-46　可变集合常用操作

```
>>> myset4={'red','green','blue','yellow'}
>>> myset4_copy=myset4.copy()               # 创建一个集合副本对象
>>> others={'black','white'}
# 可变集合增添元素
```

```
>>> myset4.add('orange')                          # 使用集合方法 add 函数增添元素
>>> myset4.update(others)                         # 使用集合方法 update 函数合并两个集合
>>> myset4
{'black', 'green', 'yellow', 'orange', 'white', 'blue', 'red'}
# 删除可变集合元素
>>> myset4.pop()                                  # 使用 pop 函数从集合中抽离出一个元素
'black'
>>> myset4                                        # 查看抽离元素后的集合内容
{'green', 'yellow', 'orange', 'white', 'blue', 'red'}
>>> myset4.remove('yellow')                       # 使用 remove 函数删除指定元素
>>> myset4_copy.clear()                           # 使用 clear 函数将副本集合内容清空
>>> myset4_copy
set()
>>> 'green' in myset4                             # 使用 in 查看集合是否包含指定元素
True
>>> len(myset4)                                   # 使用 len 函数获取集合元素个数
5
```

通过本小节内容的学习，读者体验了使用 Python 来处理数学集合是非常方便的，只需要熟练掌握前面介绍的集合运算和常用集合方法及函数即可，向集合当中存储数据和挖掘集合数据中的某些信息将会是一件简单轻松的事情。

3.5.5　任务实现

根据任务分析，本任务的具体实现过程可以参考如下操作。

（1）使用方括号创建列表[‘apple’,‘pear’,‘watermelon’,‘peach’]，并赋值给变量 set1。

（2）使用 list 函数创建列表[‘pear’,‘banana’,‘orange’,‘peach’,‘grape’]，并赋值给 set2。

（3）使用 set 函数将所创建的各个列表对象分别转换为可变集合类型。

（4）使用 type 函数查看对象转换后的数据类型。

（5）使用符号“|”求出两个集合的并集。

（6）使用符号“&”求出两个集合的交集。

（7）使用符号“–”求出两集合的差集 set1–set2。

（8）使用 difference 函数求出差集 set2–set1。

参考代码如任务实现 3-4 所示。

任务实现 3-4

```
# -*-coding:utf-8-*-

set1=['apple','pear','watermelon','peach']
set2=list(('pear','banana','orange','peach','grape'))
set1=set(set1)                                    # 转换列表对象为集合类型
```

```
set2=set(set2)
type(set1);type(set2)                    # 查看转换后的数据类型
set1|set2                                # 求出并集
set1&set2                                # 求出交集
set1-set2                                # 求出差集 set1-set2
set2.difference(set1)                    # 求出差集 set2-set1
```

小结

Python 中的数据结构是存储和组织数据的基本容器，是实现 Python 各种强大功能所不可或缺的重要部分。本章介绍了 Python 中的列表、元组、字典、集合这几种基本而且重要的数据结构，并将这 4 种数据结构归结为序列、映射、集合 3 种 Python 基础数据结构类型，同时也根据是否可变的性质进行分类。从这两个角度出发，对数据结构的特性、常用处理方法和函数等进行了讨论。下面总结一下本章涉及的具体知识点。

（1）Python 主要包含序列、映射、集合 3 种基本数据结构类型，其中序列包括列表、元组等数据结构；映射只包含字典类型的数据结构，包含的重要组成部分是映射关系；集合类型与数学中的集合有密切联系。

（2）Python 中的列表是有序、可变的数据结构，其归属于序列类型，可以进行元素索引和切片操作，根据可变性质，列表包含多种元素的增删改查等处理方法和函数，这体现出列表是 Python 当中非常灵活的内置数据结构。

（3）Python 中的元组是有序、不可变的数据结构，元组与列表唯一的差别就是它的不可变性。元组也具有元素索引和切片操作性能，由于其具有不可变性，因此元组所包含的处理方法和函数相对较少。

（4）Python 中的字典是无序、可变的数据结构，其最主要的组成部分是键值对。键值对是键和值之间的映射关系，字典中的键必须是不可变的，而值则可以为任意数据类型。字典对元素的操作主要针对对应的键，使用字典包含的方法和函数可以轻松地进行元素的增删改查等处理。

（5）Python 中的集合是无序的数据结构，具体包括可变集合和不可变集合两种类型。Python 中的集合如同数学中的集合，可以进行集合的基本运算，包括并集、交集、差集等。对于可变集合，还可以进元素的增添、删除、查询等操作。

实训

实训 1　计算出斐波那契数列前两项给定长度的数列，并删除重复项和追加数列各项之和为新项

1. 训练要点

（1）掌握斐波那契数列的特点，斐波那契数列的前两项为 0 和 1，之后各项都等于其前两项之和，例如数列第 n 项为 $F(n)=F(n-2)+F(n-1)$（$n\geqslant 2$，$n\in\mathbf{N}^{+}$）

（2）掌握列表的创建以及元素的增添、删除和修改操作方法。

（3）掌握函数 sum 的使用。

2. 需求说明

给出列表的前两项 0 和 1，对于指定的数列长度，这里指定长度为 5，利用斐波那契数列计算公式算出指定长度的数列，同时删除数列当中的重复项，计算此时数列各项之和并追加到数列尾部。

3. 实训思路及步骤

（1）创建初始数列列表，通过计算公式计算出各项的数值。

（2）使用列表方法 append 函数增添新项。

（3）根据各项计算公式和数列第一项可知，数列当中的第 3 项数值是唯一的重复值，使用列表方法 pop 函数删除第 3 项。

（4）利用 Python 中的函数 sum 获取序列中各数值元素之和，结合 append 函数来实现追加新项。

实训 2　用户自定义查询菜单，输出查询结果

1. 训练要点

（1）掌握 input 函数的使用。

（2）掌握字符型转换为数值型的方法。

（3）掌握元组元素索引方法的使用。

2. 需求说明

将菜单信息进行存储，通过字符串“请选择需要进行操作的对应数字，查询汉堡类菜单请输入 1，查询小食类菜单请输入 2，查询饮料类菜单请输入 3，若不查询请输入 0：”提示用户输入操作数字，并在用户输入数字后输出相应类型的详细食物菜单，若输入 0，则返回字符串“感谢您的使用”。

各类食物详细菜单如表 3-5 所示。

表 3-5　各类食物详细菜单

汉　堡　类	小　食　类	饮　料　类
香辣鸡腿堡	薯条	可口可乐
劲脆鸡腿堡	黄金鸡块	九珍果汁
新奥尔良烤鸡腿堡	香甜粟米棒	经典咖啡
半鸡半虾堡		

3. 实训思路及步骤

（1）创建存储食物菜单以及操作提示语字符串的元组。

（2）利用 input 函数提示用户进行输入操作。

（3）将用户输入的数字操作转换为数值类型，用于菜单元组元素索引。

实训 3　简单的好友通讯录管理程序

1. 训练要点

（1）掌握字典数据结构的创建方法。

（2）掌握字典元素的增添、删除、修改、查询等常用操作。

（3）掌握字符串方法 split 函数的使用。

（4）初步掌握 Python 中 if-elif-else 控制流语句的使用。

2. 需求说明

保存已有好友通讯录信息，并通过字符串提示用户对好友通讯录信息进行增删改查操作，输入数字 1 进行好友添加，输入数字 2 删除好友，输入数字 3 和 4 分别进行好友信息修改和查询，接着根据用户选择的处理方式来进行针对性的好友信息管理。

已有好友通讯录信息如表 3-6 所示。

表 3-6　好友通讯录信息

姓　名	电　话	地　址
小明	001	广州
小红	002	深圳
小王	003	北京

3. 实训思路及步骤

（1）创建字典对象来保存已有的好友通讯录信息。

（2）使用 input 函数提示用户选择操作方式。

（3）由于 input 函数得到的是一个将用户输入信息集合起来的字符串，可以使用字符串方法 split 函数将字符串按传入的分隔符进行拆分，得到被拆分的子字符串组成的列表。

（4）使用 Python 中的 if-elif-else 语句进行条件语句处理，当 if 或 elif 后面的条件满足时，程序就会执行其后面的语句，否则就执行 else 后面的语句。使用 if-elif-else 语句可以对好友信息字典元素进行增删改查等选择性处理操作。

实训 4　对两个给定的数进行最大公约数、最小公倍数的分析

1. 训练要点

（1）掌握最大公约数和最小公倍数的概念，最大公约数就是两个数各自的约数集合的交集中的最大元素，最小公倍数则是两个数各自的倍数集合的交集中的最小元素。

（2）掌握 max 函数和 min 函数的使用。

（3）掌握交集、差集等集合运算的 Python 实现。

2. 需求说明

获取两个给定数的最大公约数和最小公倍数，并分别提取出它们各自独有的约数和倍数集合，这里给定的两个数为 24 和 36。

3. 实训思路及步骤

（1）24 的约数集合为{1,2,3,4,6,8,12,24}，而 36 的约数集合为{1,2,3,4,6,9,12,18,36}，各自的倍数集合分别只取到 5 倍。

（2）利用 max 和 min 函数分别获取序列当中的最大值和最小值。

（3）利用 Python 中的交集集合运算求出 24 和 36 的最大公约数与最小公倍数。

（4）利用差集运算分别获取 24 和 36 各自独有的约数集合。

课后习题

1．选择题

（1）Pyhton 中的序列类型数据结构元素的切片操作非常灵活且功能强大，对于列表 Letter=[‘a’,’b’,’c’,’d’,’e’]，下述操作会正常输出结果的是（　　）。

A．Letter[-4:-1:-1]　B．Letter(:3:2)　C．Letter[1:3:0]　D．Letter[‘a’:’d’:2]

（2）列表类型数据结构拥有很多方法和函数，可以实现对列表对象的常用处理，对于列表对象 names=[‘Lucy’,’Lily’,’Tom’,’Mike’,’David’]，下述列表方法和函数使用正确的是（　　）。

A．names.append(‘Helen’,’Mary’)　B．names.remove(1)

C．names.index(‘Jack’)　D．names[2]=’Jack’

（3）下述操作不改变对象本身的是（　　）。

A．List.insert(2,’A’)　B．Tuple.copy()　C．del Dict[‘key1’]　D．Set.add(‘A’)

（4）下述对元组的操作合法的是（　　）。

A．Tuple.extend(otherTuple)　B．Tuple[0]=’A’

C．Tuple.sort()　D．Tuple1+Tuple2

（5）Python 中的数据结构可分为可变类型与不可变类型，下面属于不可变类型的是（　　）。

A．字典　B．列表　C．字典中的键　D．集合（set 类型）

（6）字典类型主要是根据键来提取对应值，通过赋值操作可以实现字典元素的增添和修改，若对于字典 Dict={2:‘two’,3:‘three’,1:‘one’}进行操作 Dict[1]=‘One’，此时字典 Dict 将会变为（　　）（注意，下面选项不考虑顺序性）。

A．{2:’two’,3:’One’,1:’one’}　B．{2:’two’,3:’three’,1:’One’}

C．{2:’two’,3:’three’,1:’one’,1:’One’}　D．{2:’One’,3:’three’,1:’one’}

（7）利用字典方法 keys 函数与 values 函数分别可以获取字典中的键和值，通过 list 函数可将结果转换为列表，其排列顺序保持着键与值的对应关系。对于上题的字典 Dict，若 list(Dict.values())[0]为‘two’，则 list(Dict.keys())[0]的结果是（　　）。

A．3　B．2　C．1　D．无法确定

（8）若要获取两个集合 A 和 B 的并集，在 Python 中应该使用（　　）。

A．B　B．A+B　C．A|B　D．A^B

（9）在 Python 中对两个集合对象实行操作 A&B，得到的结果是（　　）。

A．并集　B．交集　C．差集　D．异或集

（10）数据结构 frozenset 可以归类为（　　）。

A．序列　B．映射　C．可变类型　D．不可变类型

2. 操作题

（1）利用 Python 中的方法和函数提取出给定列表[5,8,–7,4,6,2,–3,0]中的最大元素，并删除最小元素，同时将负数的负号去除。

（2）给定有关生日信息的字典{‘小明’:‘4 月 1 日’,‘小红’:‘1 月 2 日’,‘老王’:‘4 月 1 日’,‘小强’:‘9 月 10 日’}，查询出小明的生日并修改为“5 月 1 日”，同时将老王的生日信息删除，增加小王的生日信息为“10 月 1 日”。

第 4 章 程序流程控制语句

控制语句是程序语言的基础，也是程序编写的重点。掌握 Python 的流程控制语句的应用，可以实现机器算法的自编及面向对象编程等进阶操作。本章主要介绍 Python 的条件分支结构 if 语句及两种主要循环结构 while 语句和 for 语句，并详细讲解 Python 循环结构中一些函数的用法。

学习目标

（1）掌握 if、else 和 elif 语句的基本结构与语法。
（2）掌握 for 与 while 循环语句的基本结构与用法。
（3）掌握循环语句中常用的 range 函数，以及 break、continue、pass 语句。
（4）掌握嵌套循环，以及条件与循环的组合。
（5）了解多变量迭代。
（6）掌握列表解析的创建。

任务 4.1 实现考试成绩等级划分

任务描述

运用 Python 流程控制语句的 if 语句和 else 语句编写程序，实现对考试成绩进行等级划分：分数≥90，等级为 A；80≤分数<90，等级为 B；70≤分数<80，等级为 C；60≤分数<70，等级为 D；分数<60，等级为 E。

任务分析

通过如下步骤可以实现上述任务。
（1）创建一个变量，将成绩赋予它。
（2）设置条件分支判断成绩属于哪个等级。
（3）打印结果。

4.1.1 掌握 if 语句的基本结构

首先输入成绩，如果成绩在一个等级范围内（例如等级 A 范围是 90 以上），则输出这次的考试成绩属于的等级。

如果想通过 Python 实现上述过程，就必须借助 if 语句实现条件分支，当然还需要用到

布尔表达式，格式如下。

```
if 布尔表达式 1：
    分支
```

注意，每个条件后面都要使用冒号（:）来表示接下来满足条件时要执行的语句块。使用缩进来划分语句块，相同缩进的语句组成一个语句块。

第 2 章中已经介绍过操作运算符，使用布尔表达式可以返回一个布尔值（或称为真值）的表达式。False、None、0、“”、()、[]、{}值作为布尔表达式的时候，会直接返回假（False）。就是说，标准值 False 和 None 以及数字 0 和所有空序列都为 False，其余单个对象都为 True。

这里将使用逻辑表达式实现判断，逻辑表达式是布尔表达式的一种，指的是带逻辑操作符或比较操作符（如>、==）的表达式，返回值是 False 或者 True，如代码 4-1 所示。

代码 4-1　if 语句示例

```
>>> score = 91
>>> score >= 90 and score <= 100
True
>>> score = 91
>>> if score >= 90 and score <= 100:
…    print('本次考试，成绩等级为：A')
本次考试，成绩等级为：A
```

从代码 4-1 可以看出，程序只对成绩进行了一次判断，条件满足时返回真，这里打印的结果就是“本次考试，成绩等级为：A”。

4.1.2　实现多路分支（else、elif）

4.1.1 小节中介绍了 if 语句的分支，if 语句能够设置多路分支，有且只有一条分支会被执行，这和日常语言中的“如果”是一样的。程序都是一条条语句顺序执行的，通过 else 与 elif 语句，程序可以选择执行。使用 if 语句设置多路分支的一般格式如下。

```
if 布尔表达式 1：
    分支一
elif 布尔表达式 2：
    分支二
else：
    分支三
```

程序会先计算第 1 个布尔表达式，如果结果为真，则执行第 1 个分支中的所有语句；如果为假，则计算第 2 个布尔表达式，如果第 2 个布尔表达式的结果为真，则执行第 2 个分支中的所有语句；如果结果仍然为假，则执行第 3 个分支中的所有语句。如果只有两个分支，那么不需要 elif，直接写 else 即可。如果有更多的分支，则需要添加更多的 elif 语句。Python 中没有 switch 和 case 语句，多路分支只能通过 if-elif-else 控制流语句来实现。注意，整个分支结构中是有严格的退格缩进要求的，两个分支的示例如代码 4-2 所示。

代码 4-2　两个分支的示例

```
>>> score = 59
>>> if score < 60:
...    print('考试不及格')
>>> else:
...    print('考试及格')
考试不及格
```

4.1.3　任务实现

根据任务分析，本任务的具体实现过程可以参考如下操作。

（1）创建 score 变量来存放成绩数据。

（2）设置 if 语句分支。

（3）通过 else 与 elif 语句添加分支。

（4）打印结果。

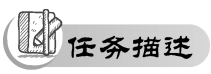
参考代码如任务实现 4-1 所示。

任务实现 4-1

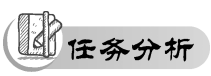
```
# -*- coding: utf-8 -*-

score = 78
if score >= 90:
    print('本次考试，成绩等级为：A')
elif score >= 80 and score < 90:
    print('本次考试，成绩等级为：B')
elif score >= 70 and score < 80:
    print('本次考试，成绩等级为：C')
elif score >= 60 and score < 70:
    print('本次考试，成绩等级为：D')
else:
    print('本次考试，成绩等级为：E')
```

任务 4.2　实现一组数的连加与连乘



一般情况下，程序都是一条一条语句顺序执行的，如果要让程序重复地做一件事情，就只能重复地写相同的代码，操作会比较烦琐。为应对此问题，一个重要的方法——循环应运而生，可以轻易实现对一组数据进行连续加法和连续乘法。



通过如下步骤可以实现上述任务。

（1）创建一个包含 1 ~ 10 的数字的列表对象。
（2）创建变量来存放计算结果。
（3）编写循环语句。
（4）编写连加与连乘公式。
（5）打印结果。

4.2.1 编写 for 循环语句

for 循环在 Python 中是一个通用的序列迭代器，可以遍历任何有序的序列，如字符串、列表、元组等。

Python 中的 for 语句接收可迭代对象，如序列和迭代器作为其参数，每次循环可以调取其中一个元素。Python 的 for 循环看上去像伪代码，非常简洁。代码 4-3 给出了 for 循环对字符串和列表元素的遍历。

代码 4-3 使用 for 循环对字符串和列表元素进行遍历

```
>>> for a in ['e','f','g']:
... print(a)
e f g
>>> for a in 'string':
... print(a)
s t r i n g
```

4.2.2 编写 while 循环语句

while 循环也是最常用的循环之一，它的格式如下。

```
while 布尔表达式:
    程序段
```

只要布尔表达式为真，程序段就会被执行。执行完毕后再次计算布尔表达式，如果结果仍然为真，则再次执行程序段，直至布尔表达式为假，如图 4-1 所示。

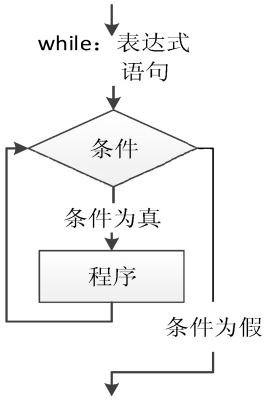


图 4-1 while 循环示意图

一个循环累加一定次数的编程示例如代码 4-4 所示。

代码 4-4　while 循环计数

```
>>> s = 0
>>> while(s <= 1):
... print('计数: ', s)
... s = s + 1
计数:  0
计数:  1
```

代码 4-4 的意思是，当 s 的值小于等于 1 的时候打印出 s，这里的结果循环到 1，一共打印了两次计数。

当条件判断语句即布尔表达式一直为真时，就会进行无限次循环，如代码 4-5 所示。

代码 4-5　无限次循环

```
>>> s = 1
>>> while(s <= 1):
... print('无限次循环')
无限次循环
无限次循环
...
```

代码 4-5 中的无限次循环可以使用“Ctrl+C”组合键来中断执行。还有另外两个重要的语句 continue、break，可以用来跳出循环，continue 用于跳过该次循环，break 则是用于退出循环。这些语句的相关知识将在 4.2.4 小节进行讲解。

4.2.3　range 函数

如果希望 Python 能像 C 语言程序的格式那样进行循环，那么实际上需要的是一个数字序列，使用 range 函数能够快速构造一个数字序列，例如，range(5)或 range(0,5)即构造了序列 0,1,2,3,4。注意，这里包括 0，但不包括 5。

在 Python 中，for i in range(5)的执行效果和 C 中 for(i=0;i<5;i++)的执行效果是一样的。range(a,b)能够返回列表[a,a+1,⋯,b–1]（注意不包含 b），这样 for 循环就可以从任意起点开始，在任意终点结束。range 函数经常和 len 函数一起用于遍历整个序列。len 函数能够返回一个序列的长度，for i in range(len(L))能够迭代整个列表 L 的元素索引。虽然直接使用 for 循环似乎也可以实现这个效果，但是直接使用 for 循环难以对序列进行修改，因为每次迭代调取的元素并不是序列元素的引用。而通过 range 函数和 len 函数可以快速通过索引访问序列并对其进行修改，如代码 4-6 所示。

代码 4-6　range 函数的使用

```
>>> for i in range(0,5):
... print(i)
#result:0,1,2,3,4
>>> for i in range(0,6,2):
... print(i)
```

```
0,2,4                                    #相邻元素的间隔为 3
#直接使用 for 循环难以改变序列元素
>>> L = [1,2,3]
>>> for a in L:
... a+=1                                 #a 不是引用，L 中对应的元素没有发生改变
>>> print(L)
[1, 2, 3]
# 结合 range 与 len 函数来遍历序列并修改元素
>>> for i in range(len(L)):
... L[i]+=1                              #通过索引访问
>>> print(L)
[2, 3, 4]
```

4.2.4 运用 break、continue、pass 语句

1. break

break 语句用在 while 和 for 循环中，用来终止循环语句，即使循环条件没有 False 条件或者序列还没被完全递归完，也会停止执行循环语句。如果使用在嵌套循环中，break 语句可以停止执行最深层的循环，并开始执行下一行代码。

在 while 和 for 循环中使用 break 语句的示例如代码 4-7 所示。

代码 4-7　break 语句的使用

```
>>> s = 0
>>> while True:
...   s+=1
...   if s == 6:                         #满足 s 等于 6 的时候跳出循环
...     break
>>> s
6
>>> for i in range(0,10):
... print(i)
... if i == 1:                           #当 i 等于 1 的时候跳出循环
...     break
0
1
```

从代码 4-7 中可以看到，break 语句是直接跳出整个循环。在 while 循环中，当 s 等于 6 的时候，整个循环就结束了。在 for 循环中，i 等于 1 的时候，整个循环也结束了。

2. continue

continue 语句的作用是跳出本次循环，不像 break 那样是跳出整个循环。continue 语句用来告诉 Python 跳过当前循环的剩余语句，继续进行下一轮循环。continue 语句也是用在 while 和 for 循环中，运用示例如代码 4-8 所示。

代码 4-8 continue 语句的使用

```
>>> s = 3
>>> while s > 0:
... s = s - 1
... if s == 1:                          #当 s 等于 1 时跳出本次循环
...     continue
... print(s)
2
0
>>> for i in range(0,3):
... if i == 1:                          #当 i 等于 1 时跳出本次循环
...     continue
... print(i)
0
2
```

代码 4-8 中，while 循环在 s 等于 1 时直接跳过本次循环，继续进行。for 循环也是，从结果可以看出。

3. pass

pass 是空语句，作用是保持程序结构的完整性。pass 不做任何事情，一般用作占位语句。pass 语句应用示例如代码 4-9 所示。

代码 4-9 pass 语句的使用

```
>>> for i in range(0,3):
... if i == 1:
...     pass
...     print('pass 块')
... print(i)
0
pass 块
1
2
```

可以看到，pass 语句用于在输出结果 0 ~ 1 之间占位，不做任何事情。

4.2.5 任务实现

根据任务分析，本任务的具体实现过程可以参考如下操作。

（1）创建一个列表 vec。

（2）连加时，创建一个赋值为 0 的变量 m；连乘时，创建一个赋值为 1 的变量 n。

（3）编写 for 循环语句。

（4）编写连加或连乘公式。

（5）打印结果。

参考代码如任务实现 4-2 所示。

任务实现 4-2

```
# -*- coding: utf-8 -*-

# 连加
vec = [1,2,3,4,5,6,7,8,9,10]
m = 0
for i in vec:
    m = m + i
print(m)

# 连乘
n = 1
for i in vec:
    n =n * i
print(n)
```

任务 4.3 使用冒泡排序法排序

任务描述

使用冒泡排序法对数据进行排序，是程序流程控制语句中嵌套循环、条件语句和循环语句组合应用的实例之一。

任务分析

通过如下步骤可实现上述任务。

（1）创建一个列表对象[1,8,2,6,3,9,4,12,0,56,45]。

（2）编写嵌套循环。外循环 i 的取值为 0 到列表对象的长度，内循环 j 的取值为 i+1。

（3）当遍历的列表对象的前一个元素比后一个元素小时，两个元素的位置互换。

（4）打印结果。

4.3.1 掌握嵌套循环

嵌套循环，顾名思义，就是在一个循环中嵌入另一个循环。而 Python 语言是允许在一个循环体里面嵌入另一个循环的。例如可以在 for 循环中再嵌入另一个 for 循环，也可以在 for 循环中嵌入 while 循环，还可以在 while 循环中嵌入 for 循环，当然也可以在 while 循环中嵌入 while 循环。

for 循环与 for 循环的嵌套示例如代码 4-10 所示。

代码 4-10 for 循环与 for 循环的嵌套

```
>>> num = zeros(shape=(3,3))
```

```
>>> num
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> for i in range(0,3):
...     for j in range(0,3):
...         num[i,j] = i * j
>>> num
array([[ 0.,  0.,  0.],
       [ 0.,  1.,  2.],
       [ 0.,  2.,  4.]])
```

从代码 4-10 可以看出，利用嵌套循环对数组 num 里面的值进行了修改，重新赋值为 i*j。

while 循环与 for 循环的嵌套示例如代码 4-11 所示。

代码 4-11　while 循环与 for 循环的嵌套

```
>>> for i in range(0,11):
...     while(i > 8):
...         print(i*10)
...         break
90
100
```

执行代码 4-11，可打印出当 i>8 时 i 乘以 10 的值。

4.3.2　组合条件与循环

在循环中放入条件语句，才可以使得循环能够做更多的事情。for 循环与条件语句的组合应用示例如代码 4-12 所示。

代码 4-12　for 循环与条件语句的组合

```
>>> for x in range(10,15):                  # 迭代 10 ~ 15 的数字
...     for i in range(2,x):                # 根据因子迭代
...         if x%i == 0:                    # 确定第 1 个因子
...             j=x/i                       # 计算第 2 个因子
...             print('%d 等于 %d * %d' % (x,i,j))
...             break                       # 跳出当前循环
...         else:                           # 循环的 else 部分
...             print(x, '是一个质数')
...             break                       # 跳出当前循环
10 等于 2 * 5
11 是一个质数
12 等于 2 * 6
13 是一个质数
14 等于 2 * 7
```

代码 4-12 使用 for 循环和 if 条件语句对数据进行判断，判断是否为质数并打印结果。if 语句后面表达式的意思就是判断 x 求余是否为 0，当求余为 0 时，x 就不是质数，否则为质数。

while 循环与条件语句的组合应用示例如代码 4-13 所示。

代码 4-13　while 循环与条件语句的组合

```
>>> count = 0
>>> while count < 5:
...    if count > 3:
...       print(count**2)
...    else:
...       print(count)
...    count = count + 1
0
1
2
3
16
```

代码 4-13 在 while 循环中设置条件语句，当 count 大于 3 的时候打印两倍的计数。

4.3.3　任务实现

根据任务分析，本任务的具体实现过程可以参考如下操作。

（1）创建列表 mppx。

（2）编写 for 循环与 for 循环的嵌套循环，外循环 i 的取值为 range(len(mppx))，内循环 j 的取值为 range(i+1)。

（3）设置条件语句，当列表中的后一个元素比前一个元素大时，将它们的位置互换。

（4）打印结果。

参考代码如任务实现 4-3 所示。

任务实现 4-3

```
# -*- coding: utf-8 -*-

###冒泡排序
mppx = [1,8,2,6,3,9,4,12,0,56,45]            #定义列表
for i in range(len(mppx)):
    for j in range(i+1):
        if mppx[i] < mppx[j]:
            mppx[i],mppx[j] = mppx[j],mppx[i] # 实现两个变量位置的互换
print(mppx)
```

任务 4.4 输出数字金字塔

任务描述

前面学习了条件语句和循环语句的编程实现，本任务将运用嵌套循环和多变量迭代实现输出数字金字塔，达到输入一个数字便可以自动输出数字金字塔的效果。

任务分析

通过如下步骤可实现上述任务。

（1）设置输入语句，输入数字。

（2）创建变量来存放金字塔层数。

（3）编写嵌套循环，控制变量存放每一层的长度。

（4）设置条件来打印每一行输出的数字。

（5）输出打印结果。

4.4.1 多变量迭代

如果给定一个 list 或 tuple，通过 for 循环可以遍历这个 list 或 tuple，这种遍历称为迭代（Iteration）。在 Python 中，迭代是通过 for in 语句来完成的。Python 的 for 循环不仅可以用在 list 或 tuple 上，还可以作用在其他可迭代的对象上。list 这种数据类型有下标，但很多其他数据类型是没有下标的，只要是可迭代对象，无论有无下标，都可以迭代。字典 dict 的迭代示例如代码 4-14 所示。

代码 4-14 dict 的迭代

```
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> for key in d:
...     print(key)
a
c
b
```

代码 4-14 中，因为 dict 的存储不是按照 list 的方式顺序排列的，所以迭代出的结果顺序很可能不一样。

在 Python 中，for 循环同时引用两个变量也很常见，示例如代码 4-15 所示。

代码 4-15 同时引用两个变量

```
>>> for x, y in [(1, 1), (2, 4), (3, 9)]:
>>> print(x, y)
1 1
2 4
3 9
```

3 个变量的迭代示例如代码 4-16 所示。

代码 4-16　3 个变量的迭代

```
>>> for x,y,z in [(1,2,3),(4,5,6),(7,8,9)]:
>>> print(x,y,z)
1 2 3
4 5 6
7 8 9
```

4.4.2　创建列表解析

列表解析是一种高效创建新列表的方式，可以用来动态地创建列表。列表解析是 Python 迭代机制的一种应用，它常用于实现创建新的列表，因此用在[ ]中。

列表解析也可以称为列表推导式。列表解析示例如代码 4-17 所示。

代码 4-17　列表解析示例

```
>>> map(lambda x: x**3, range(6))                    #计算 x 的 3 次幂
[0,1,8,27,64,125]
>>> [x**3 for x in range(6)]
[0, 1, 8, 27, 64, 125]
>>> seq = [1,2,3,4,5,6,7,8]                          #当 x%2 为 1 时取值
>>> filter(lambda x: x % 2, seq)
[1,3,5,7]
>>> [x for x in seq if x % 2]
[1, 3, 5, 7]
```

如代码 4-17 所示，列表解析式完全可以替换 Python 内建的 map 函数以及 lambda 函数，而且效率更高。

用列表解析式实现嵌套循环语句的示例如代码 4-18 所示。

代码 4-18　列表解析式嵌套循环示例

```
>>> [(i,j) for i in range(0,3) for j in range(0,3)]
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
>>> [(i,j) for i in range(0,3) if i < 1 for j in range(0,3) if j > 1]
[(0, 2)]
```

如代码 4-18 所示，列表解析式不仅可以运用到嵌套循环中，还可以在其中增加条件判断语句。列表解析式实现效率更高，且代码更加简洁。

4.4.3　任务实现

根据任务分析，本任务的具体实现过程可以参考如下操作。

（1）利用 eval 函数与 input 函数设置输入语句，输入数字。

（2）创建变量 level 来存放金字塔层数。

（3）编写嵌套循环，创建变量 kk 来存放每一层长度，设置 t 等于金字塔层数，设置 length 存放 2*t-1。内循环中划分 k 等于 1 时与 k 不等于 1 时的情况。

（4）设置公式来打印每一行输出的数字，这里可以利用 format 函数。

（5）打印结果。

参考代码如任务实现 4-4 所示。

任务实现 4-4

```
# -*- coding: utf-8 -*-

#完整金字塔
num = eval(input("输入一个整数:"))
print("数字金字塔显示如下:")
level = 1#金字塔的高度即层数
while level <= num:
    kk = 1#每一层长度计数
    t = level
    length = 2*t - 1
    while kk <= length:
        if kk == 1:
            if kk == length:
                print(format(t,str(2*num - 1)+"d"),"\n")
                break
#要形成金字塔型，13d 是 1 的距离，15d 是 7 的距离，然后进行测试
#只要之后的距离比 1 的距离多 2，再减去 2 倍的层数即可以得到金字塔形状
            else:
                print(format(t,str(2*num + 1 - 2*level)+"d"),"",end="")
                t -= 1
        else:
            if kk == length:
                print(t,"\n")
                break
            elif kk <= length/2:
                print(t,"",end="")
                t -= 1
            else:
                print(t,"",end="")
                t += 1
        kk += 1
    level += 1
```

小结

编程语言可以模拟人类生活的方方面面，使用编程语言中特殊的关键字可以控制程序的执行过程，这些关键字组成的就是流程控制语句。本章介绍的流程控制语句主要分分支

语句、循环语句。下面总结一下本章涉及的知识点。

（1）条件分支语句通过一条或多条语句（判断条件）的执行结果（True/False）来决定执行哪个分支的代码块。Python 中提供的分支语句为 if-else 语句，没有提供 switch、case 语句。本章介绍了单分支（if）、双分支（if-else）、多分支（if-elif-else）。

（2）break 语句可以跳出整个循环，使用 continue 语句可以跳出当前循环剩余语句，继续下一轮循环。pass 语句不做任何事情，一般用作占位语句。

（3）Python 的嵌套循环，包含 for 循环的嵌套和 while 循环的嵌套。

（4）Python 的多变量迭代，内容包括迭代、单变量迭代、两个及多个变量迭代。列表解析式也就是列表推导式，利用列表解析式可以实现 for 循环、while 循环、嵌套循环。

实训

实训 1　猜数字游戏

1. 训练要点

（1）掌握使用一个函数 randint 产生一个随机数字的方法，掌握变量赋值运算的使用。

（2）掌握 if-elif-else 条件语句的应用。

（3）掌握 while 循环，并且掌握条件语句与循环语句的组合应用。

2. 需求说明

加载第三方库，创建一个变量来存放产生的随机数字。将猜一猜的计数变量赋值为 0，输入一个数字，利用 while 循环和条件语句的组合去判断输入的数字是否与随机产生的数字一致。

3. 实训思路及步骤

（1）加载 random 库，创建变量 number 来存放随机数字，创建变量 guess 并赋值为 0，guess 将用来计猜的次数。

（2）编写 while 循环，利用 input 函数获取输入的数字。

（3）在循环中，检查输入的是否为数字，当输入的数字大了或者小了时做出提示。

（4）当判断正确后，输出正确的信号并输出猜了多少次。

实训 2　统计字符串内元素类型的个数

1. 训练要点

（1）掌握变量的创建方法。

（2）掌握输入语句的用法。

（3）掌握 for 循环与条件语句的组合应用。

（4）掌握 int 型判断函数和 str 型判断函数的用法。

（5）掌握%字符和 d 的含义，打印输出转换的字符。

2. 需求说明

给出记录 int 元素个数的变量、记录 str 元素个数的变量、记录其他元素个数的变量，并赋初始值为 0；使用新函数 is.digit、is.alpha；使用 for 循环与条件语句的组合。

3. 实训思路及步骤

（1）创建 3 个变量 intCount、strCount、otherCount 并赋值为 0，用来存放 int 元素、str 元素、other 元素的个数。

（2）编写输入语句，输入字符串。

（3）编写 for 循环，设置条件语句，用 is.digit 函数和 is.alpha 函数判断字符串元素是 int 型、str 型，还是其他类型，并使用相应的 3 个变量 intCount、strCount、otherCount 计数。

（4）打印结果。

课后习题

1. 选择题

（1）在 if 语句中进行判断，产生（　　）时会输出相应的结果。

A．0　　B．1　　C．布尔值　　D．以上均不正确

（2）在 Python 中实现多个条件判断需要用到（　　）语句与 if 语句的组合。

A．else　　B．elif　　C．pass　　D．以上均不正确

（3）循环中可以用（　　）语句来跳出深度循环。

A．pass　　B．continue　　C．break　　D．以上均可以

（4）可以使用（　　）语句跳出当前循环的剩余语句，继续进行下一轮循环。

A．pass　　B．continue　　C．break　　D．以上均可以

（5）在 for i in range(6)语句中，i 的取值是（　　）。

A．[1,2,3,4,5,6]　　B．[1,2,3,4,5]　　C．[0,1,2,3,4]　　D．[0,1,2,3,4,5]

（6）列表解析是 Python 迭代机制的一种应用，常用于实现创建新的列表，因此用在（　　）中。

A．()　　B．[]　　C．{}　　D．以上都可以

（7）while 循环语句和 for 循环语句使用 else 的区别是（　　）（多选）。

A．else 语句和 while 循环语句一起使用，则当条件变为 False 时，执行 else 语句

B．else 语句和 while 循环语句一起使用，则当条件变为 True 时，执行 else 语句

C．else 语句和 for 循环语句一起使用，else 语句块只在 for 循环正常终止时执行

D．else 语句和 for 循环语句一起使用，else 语句块只在 for 循环不正常终止时执行

（8）列表解析式[i+6 for i in range(0,3)]返回的结果是（　　）。

A．[1,2,3]　　B．[0,1,2]　　C．[6,7,8]　　D．[7,8,9]

（9）有一个列表 L=[4,6,8,10,12,5,7,9]，列表解析式[x for x in L if x%2==0]返回的结果是（　　）。

A．[4,8,12,7]　　B．[6,10,5,9]　　C．[4,6,8,10,12]　　D．[5,7,9]

（10）如下代码中可以正确运行出结果的是（　　）。

A．[n for i in range(0,3)]　　B．[n = I for I in range(0,3)]

C．[j for j in range(0,3)]　　D．以上均可以

2. 操作题

（1）使用嵌套循环实现 99 乘法法则。

（2）编写代码，打印图 4-2 所示的图形。

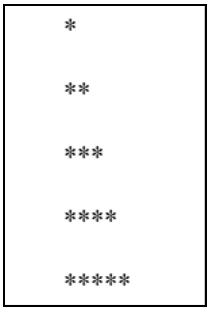


图 4-2　图形

第5章 函数

函数是 Python 为了使代码效率最大化，减少冗余而提供的最基本的程序结构。第 4 章介绍了众多流程控制的语句，在大中型程序中，同一段代码可能会被重复使用，但如果程序由一段冗余的流程控制语句组成，则程序的可读性会变差。使用函数封装这些重复使用的程序段，并加以注释，下次使用的时候直接调用，可以使代码更清晰。

本书前 4 章的内容虽然没有介绍函数封装的概念，但在第 3 章已经有所接触。例如列表操作的各种方法都是函数。如果不封装，很多时候没有必要去探究数据结构源码具体是如何编写的，每种数据结构都会提供众多的函数和相应的说明文档，程序开发人员仅需知道函数的输入和输出就可以使用数据结构编程了。

学习目标

（1）认识自定义函数，了解自定义函数的调用。
（2）掌握函数的参数设置及返回函数（return 函数）。
（3）掌握嵌套函数。
（4）掌握局部变量和全局变量的区别。
（5）掌握匿名函数和其他高阶函数的使用方法。
（6）掌握存储并导入函数模块的方法。

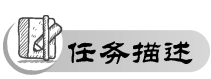
函数和方法

函数是一段代码，能将一些数据（参数）传递进程序进行处理，然后返回一些数据（返回值），也可以没有返回值，但它跟一个对象相关联。

方法和函数大致上是相同的，都是通过名字来进行调用的，但有两个主要的不同之处：一是方法中的数据是隐式传递的，函数的数据都是显式传递的；二是方法可以操作类内部的数据，而函数不行。

简单来说，就是方法和对象相关，函数和对象无关。Java 中只有方法，C 语言和 R 语言中只有函数，而 C++和 Python 的方法和函数取决于是否在面向对象编程的类中使用。

任务 5.1 自定义函数实现方差输出



任务描述

函数实现了对整段程序逻辑的封装，是程序逻辑的结构化或者过程化的一种编程方法。

使用函数，可以将实现某个功能的整块代码从代码中隔离开来，可以避免程序中出现大段重复代码。同时，维护时也只需要对函数内部进行修改即可，而不用去修改大量代码的副本。现在使用 def 关键字定义一个求列表方差的自定义函数。

通过如下步骤可实现上述任务。

（1）自定义列表的求和函数及求平方和函数。

（2）依据求和函数求列表的均值。

（3）依据求平方和函数求列表的方差。

5.1.1 认识自定义函数

跟其他编程语言一样，Python 也提供了自定义函数的功能。使用关键字 def 可定义函数，其后紧接函数名，括号内包含将要在函数体中使用的形式参数（简称形参），定义语句以冒号结束。函数体编写另起一行，函数体的缩进为 4 个空格或者一个制表符，函数定义示例如代码 5-1 所示。

代码 5-1 函数定义

```
>>>def my_function(parameter):
...   '''打印任何传入的字符串'''
...   print (parameter)                    # print 与 return 没有关系，也不会相互影响
...   return 'parameter is '+parameter
```

这个函数的名称为 my_function，输入参数是 parameter，输出参数是 parameter，返回加上“parameter is”的字符串。Python 的简洁性可以从函数中体现出来，Python 的参数不需要声明数据类型，但这也有一定的弊端，程序人员可能会因不清楚参数的数据类型而输入错误的参数，例如，若执行 my_function(1)，就会报错。所以一般在函数的开头注明函数的用途、输入和输出。

5.1.2 设置函数参数

Python 中的函数参数主要有 4 种。

（1）位置参数，调用函数时根据函数定义的位置参数来传递参数。

（2）关键字参数，通过“键-值”形式加以指定，可以让函数更加清晰，容易使用，同时也清除了参数的顺序要求。

（3）默认参数，定义函数时为参数提供的默认值，调用函数时，默认参数的值可传可不传。注意：所有的位置参数必须出现在默认参数前，包括函数定义和调用。

（4）可变参数，定义函数时，有时候不确定调用时会传递多少个参数（不传参数也可以）。此时，可用定义任意位置参数或者关键字参数的方法来进行参数传递，会显得非常方便。

1. 默认参数

在调用内建函数的时候，往往会发现很多函数提供了默认的参数。默认参数为程序人员提供了极大的便利，特别对于初次接触该函数的人来说更是意义重大。默认参数为设置

函数的参数值提供了参考。

例如代码 5-2，定义了一个计算利息的函数，其中，天数的默认值为 1，年化利率的默认值为 0.05，即 5%。

代码 5-2 默认参数

```
>>>def interest(money,day = 1,interest_rate= 0.05):
...    income = 0
...    income = money*interest_rate*day/365
...    print(income)
```

当仅需计算单日利息时，只要输入本金的数量即可，如代码 5-3 所示。

代码 5-3 默认参数使用

```
>>>interest(5000)            # 本金为 5000，年化利率为默认值 0.05 时的单日利息
0.684931506849315
>>>interest(10000)           # 本金为 10000，年化利率为默认值 0.05 时的单日利息
1.36986301369863
```

对于开发者而言，设置默认参数能让他们更好地控制软件。如果提供了默认参数，那么开发者可以设置他们期望的“最好”的默认值；而对于用户而言，也能避免初次使用便遇到一大堆参数设置的窘境。

2. 任意数量的位置可变参数

定义函数时需要定义函数的参数个数，通常情况下，这个参数个数表示了函数可调用的参数个数的上限。但是也有在定义函数时无法得知参数个数的情况，在 Python 中使用*args 和*kwargs 可以定义可变参数，在可变参数之前可以定义 0 到任意多个参数。注意，可变参数永远放在参数的最后面。

在定义任意数量的位置参数时需要一个星号前缀（*）来表示，在传递参数的时候，可以在原有的参数后面添加 0 个或多个参数，这些参数将会被放在元组内并传入函数。任意数量的位置参数必须定义在位置参数或关键字参数之后，如代码 5-4 所示。

代码 5-4 任意数量的位置可变参数

```
>>>def exp(x,y,*args):
...    print('x:',x)
...    print('y:',y)
...    print('args:',args)
>>>exp(1,5,66,55,'abc')
x: 1
y: 5
args: (66, 55, 'abc')
```

代码 5-4 定义了两个参数 x 和 y，之后定义了可变参数*args。*args 参数传入函数后存储在一个元组中。

3. 任意数量的关键字可变参数

在定义任意数量的关键字可变参数时，参数名称前面需要有两个星号（**）作为前缀。在传递的时候，可以在原有的参数后面添加任意数量的关键字可变参数，这些参数会被放到字典内并传入函数中。带两个星号前缀的参数必须在所有带默认值的参数之后，顺序不可以调转，如代码 5-5 所示。

代码 5-5　任意数量的关键字可变参数

```
>>>def exp(x,y,*args,**kwargs):
...    print('x:',x)
...    print('y:',y)
...    print('args:',args)
...    print('kwargs:',kwargs)
>>>exp(1,2,2,4,6,a='c',b=1)
```

代码 5-5 中，函数传入了“1,2,2,4,6,a=’c’,b=1”，总共 7 个参数。其中，1 和 2 被函数识别为 x 和 y，“2,4,6”被识别为*args 并存储在元组[2,4,6]中，“a=’c’,b=1”被识别为**kwargs 并存储在带有关键字的字典中。

5.1.3　返回函数值

函数可以处理一些数据，并返回一个或一组值。函数返回的值称为返回值。前文代码 5-2 中定义的函数执行了 print 操作但无返回值，如果想要保存或者调用函数的返回值，需要用到 return 函数，如代码 5-6 所示。

代码 5-6　return 函数

```
>>>def interest_r(money,day = 1,interest_rate= 0.05):
...    income = 0
...    income = money*interest_rate*day/365
...    return income
```

print 函数仅仅打印对象，打印出来的对象无法保存或者调用，而 return 函数返回的运行结果可以保存为一个对象供其他函数调用，如代码 5-7 所示。

代码 5-7　print 和 return 的区别

```
>>>x = interest(1000)
0.136986301369863
>>>y = interest_r(1000)
>>>y
0.136986301369863
```

Python 对函数返回值的数据类型没有限制，包括列表和字典等复杂的数据结构。当程序执行到函数中的 return 语句时，就会将指定的值返回并结束函数，后面的语句不会被执行。

5.1.4　调用自定义函数

Python 中使用“函数名()”的格式对函数进行调用，根据参数传入方式的不同，总共

有 3 种函数调用方式，分别为位置参数调用、关键字参数调用、可变参数调用。

1. 位置参数调用

位置参数调用是函数调用最常用的方式，函数的参数严格按照函数定义时的位置传入，顺序不可以调换，否则会影响输出结果或者直接报错。如 range 函数，定义的 3 个参数 start、stop、step 需按照顺序传入，如代码 5-8 所示。

代码 5-8 传入位置参数

```
>>>list(range(0,10,2))          # 按 start=0,stop=10,step=2 的顺序传入
[0, 2, 4, 6, 8]
>>>list(range(10,0,2))          # 调转 strart 和 stop 的顺序后传入
[]
>>>list(range(10,2,0))          # 调转全部参数的顺序后传入
Traceback (most recent call last):
  File  "E:\python\lib\site-packages\IPython\core\interactiveshell.py",  line
2881, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-2-0b56cef3a2dc>", line 1, in <module>
    list(range(10,2,0))
ValueError: range() arg 3 must not be zero
```

当函数的参数有默认设置值时，可以不设置该函数，因为此时的函数会使用默认的参数，如代码 5-9 所示。

代码 5-9 调用位置参数

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

2. 关键字参数调用

除了可以使用位置参数对函数进行调用外，还可以使用关键字参数对函数进行调用。使用关键字参数时，可以不严格按照位置，因为解释器会自动按照关键字进行匹配。例如前文定义的 interest 函数，其参数 money、day 和 interest_rate 即为关键字参数，如代码 5-10 所示。

代码 5-10 设置关键字参数

```
>>>interest(money=5000,day=7,interest_rate=0.06)
5.7534246575342465
>>>interest(day=7,money=5000,interest_rate=0.06)
5.7534246575342465
```

关键字参数也可以与位置参数混用，但关键字参数必须跟在位置参数后面，否则会报错，如代码 5-11 所示。

代码 5-11　调用关键字参数

```
>>>interest(10000,day=7,interest_rate=0.06)
11.506849315068493
>>>interest(10000,interest_rate=0.06,day=7)
11.506849315068493
>>>interest(interest_rate=0.06,7,money=10000)
# 位置参数必须在关键字参数前面，否则会报错
  File "<ipython-input-14-1fbe6ee16ee2>", line 1
    interest(interest_rate=0.06,7,money=10000)
                               ^
SyntaxError: positional argument follows keyword argument
```

3. 可变参数调用

使用*arg 可变参数列表可以直接将元组或者列表转换为参数，然后传入函数，如代码 5-12 所示。

代码 5-12　调用*arg 可变参数

```
>>>arg = [0,10,2]
>>>list(range(*arg))
[0, 2, 4, 6, 8]
```

使用**kwargs 关键字参数列表可以直接将字典转换为关键字参数，然后传入函数中，如代码 5-13 所示。

代码 5-13　调用**kwargs 可变参数

```
>>>def user(username,age,**kwargs):
...    print('username:',username,
...          'age:',age,
...          'other:',kwargs)
>>>user('john',27,city = 'guangzhou',job = 'Data Analyst')
username: john age: 27 other: {'city': 'guangzhou', 'job': 'Data Analyst'}
>>>kw={'age':27,'city':'guangzhou','job':'Data Analyst'}
>>>user('john',**kw)
username: john age: 27 other: {'city': 'guangzhou', 'job': 'Data Analyst'}
```

5.1.5　掌握嵌套函数

Python 允许在函数中定义另外一个函数，这就是通常所说的函数嵌套。定义在其他函数内部的函数称为内建函数，而包含有内建函数的函数称为外部函数。需要注意的是，内建函数中的局部变量独立于外部函数，如果想使用外部函数的变量，则需要声明该变量为全局变量。

假设需要定义一个求均值的函数，那么首先需要计算数值的和，所以要先在求均值函数的内部内建一个求和函数，如代码 5-14 所示。

代码 5-14 定义求均值函数

```
>>>def mean (*args):
...    m = 0
...    def sum(x):                              # 内建求和函数
...        sum1 = 0
...        for i in x:
...            sum1 += i
...        return sum1
...    m = sum(args)/len(args)
...    return m
```

Python 将函数也视为对象，因此允许外部函数在返回结果时直接调用内部函数的结果，所以可对求均值函数做简化，令其直接返回求和函数的结果，如代码 5-15 所示。

代码 5-15 调用求均值函数

```
>>>def means (*args):
...    def sum(x):
...        sum1 = 0
...        for i in x:
...            sum1 += i
...        return sum1
...    return sum(args)/len(args)              # 直接返回 sum 函数的结果
```

5.1.6 区分局部变量和全局变量

Python 创建、改变或查找变量名都是在命名空间中进行的，更准确地说，是在特定的作用域下进行的，所以需要使用某个变量名时，应清楚地知道其作用域。由于 Python 不能声明变量，所以变量第一次被赋值的时候即与一个特定作用域绑定了。定义在函数内部的变量拥有一个局部作用域，定义在函数外部的变量拥有全局作用域。

1. 局部变量

在定义函数时，往往需要在函数内部对变量进行定义和赋值，在函数体内定义的变量为局部变量。例如定义一个求和函数，示例如代码 5-16 所示。

代码 5-16 定义求和函数

```
>>>def sum(*arg):
...    sum1 = 0
...    for i in range(len(arg)):
...        sum1 +=arg[i]
...    return sum1
```

在代码 5-16 中，函数体内定义了一个局部变量 sum1，所有针对该变量的操作仅在函数体内有效，如代码 5-17 所示。

代码 5-17　局部变量

```
>>>sum(1,2,3,4,5)
15
>>>sum1
0
```

2. 全局变量

与局部变量相对应，定义在函数体外面的变量为全局变量，全局变量可以在函数体内被调用，如代码 5-18 所示。

代码 5-18　全局变量

```
>>>sum0 = 10
>>>def fun():
...    sum_global=sum0+100
...    return sum_global
>>>fun()
110
```

需要注意的是，全局变量不能在函数体内直接被赋值，否则会报错，如代码 5-19 所示。

代码 5-19　全局变量不能在函数体中被直接赋值

```
>>>sum1=0
>>>def sum(*arg):
...    for i in range(len(arg)):
...        sum1 +=arg[i]
...    return sum1
>>>sum(1,2,3,4)
File "<ipython-input-15-64d994261251>", line 6, in <module>
    sum(1,2,3,4)
  File "<ipython-input-15-64d994261251>", line 4, in sum
    sum1 +=arg[i]
UnboundLocalError: local variable 'sum1' referenced before assignment
```

若同时存在全局变量和局部变量，那么函数体会使用局部变量对全局变量进行覆盖，如代码 5-20 所示。

代码 5-20　局部变量覆盖全局变量

```
>>>sum1=10
>>>def sum(*arg):
...    sum1=0
...    for i in range(len(arg)):
...        sum1 +=arg[i]
...    return sum1
>>>sum(1,3,4,5)
```

```
13
```

显然，代码 5-20 中，函数使用的是函数体内部的局部变量 sum1=0。

如果想要在函数体内对全局变量赋值，那么需要使用关键字 global（嵌套函数中用 nonlocal，用法和 global 一样），则代码 5-19 可改为代码 5-21 所示的形式。

代码 5-21 使用关键字在函数体内对全局变量赋值

```
>>>sum1=0
>>>def sum(*arg):
...    global sum1
...    for i in range(len(arg)):
...        sum1 +=arg[i]
...    return sum1
>>>sum(1,3,5,7)
16
>>>sum1
16
>>>sum(1,3,5,7)
32
```

需要注意的是，虽然 global 似乎很好用，但建议程序中尽量少用，因为它会使代码变得混乱，使可读性变差。相反，局部变量会使代码更加抽象，封装性更好。

5.1.7 任务实现

方差计算公式的一个推导公式为平方的均值减去均值的平方，如式（5-1）所示。

$$S^2 = \frac{x_1^2 + x_2^2 + x_3^2 + \cdots + x_n^2}{n} - M^2 \qquad (5\text{-}1)$$

要实现一个计算方差的函数，可以按以下步骤进行操作。

（1）构建求和函数 sum 及求平方和函数。

（2）构建求均值函数 mean，需调用求和函数的结果。

（3）构建求方差函数 var，需调用求平方和函数及求均值函数的结果。

（4）在方差函数中内建求平方和函数及求均值函数，并在求均值函数中再内建一个求和函数。

参考代码如任务实现 5-1 所示。

任务实现 5-1

```
# -*-coding:utf-8-*-

#构建方差函数示例

def var (*args):                                    # 主体方差函数
    def mean(z):                                    # 内建求均值函数
```

```
    def sum(x):                                    # 内建求和函数
        sum1 = 0
        for i in x:
            sum1 += i
        return sum1
    return sum(args) / len(args)                   # 直接返回 sum 函数的结果
def sums(y):                                       # 内建求平方和函数
    sum2=0
    for i in y:
        sum2 +=i**2
    return sum2
return sums(args)/len(args)-mean(args)**2
# 返回 sums 函数及 mean 函数的结果并计算方差
```

任务 5.2 使用匿名函数添加列表元素

任务描述

Python 中有一些常用的高阶内置函数，如 lambda 函数、map 函数、fib 函数和 filter 函数等。要为一个空列表加入一组累加数据，除了可以用自定义函数的方式实现外，也可以用匿名函数和其他方式实现。

任务分析

通过如下步骤可实现上述任务。

（1）先自定义一个累加函数。

（2）创建一个空列表和一组数据。

（3）用循环结构把累加后的结果添加进列表。

（4）使用匿名函数代替累加函数，再进行后续操作。

（5）使用 map 函数快速实现以上操作。

5.2.1 创建并使用匿名函数

Python 允许使用 lambda 语句创建匿名函数，也就是说函数没有具体的名称。可能读者会产生疑惑，即函数没有名称应该不是好事。实际上，当需要定义一个功能简单但不经常使用的函数来执行脚本时，就可以使用 lambda 定义匿名函数，从而省去定义函数的过程。对于一些抽象的、不会在其他地方重复使用的函数，有时候给函数命名也很麻烦（需要避免函数重名），而使用 lambda 语句则不需要考虑函数命名的问题，同时可以避免重复使用的函数。

lambda 语句中，冒号前是函数参数，若有多个函数须使用逗号分隔；冒号后是返回值。def 语句也可以创建一个函数对象，只是使用 lambda 语句创建的函数对象没有名称。

使用 Lambda 语句创建函数的示例如代码 5-22 所示。

代码 5-22 使用 lambda 语句创建函数

```
>>>example = lambda x : x**3
>>>print(example)
<function <lambda> at 0x000001EA16DCA730>
>>>example(2)
8
```

lambda 为定义匿名函数时的关键字，arguments 为传入函数的参数，expression 为返回的结果。对于 lambda 语句，应该注意如下 4 点。

（1）lambda 定义的是单行函数，如果需要复杂的函数，应使用 def 语句。

（2）lambda 语句可以包含多个参数。

（3）lambda 语句有且只有一个返回值。

（4）lambda 语句中的表达式不能含有命令，且仅限一条表达式。这是为了避免匿名函数的滥用，过于复杂的匿名函数反而不易于解读。

Python 允许将 lambda 语句作为对象赋值给变量，然后使用变量名进行调用。举个例子，Python 的数学库中只有以自然底数 e 和 10 为底的对数函数，使用 lambda 语句即可创建指定某个数为底的对数函数，如代码 5-23 所示。

代码 5-23 使用 lambda 语句创建对数函数

```
>>>from math import log              # 引入 Python 数学库的对数函数
# 此函数用于返回一个以 base 为底的匿名对数函数
>>>def make_logarithmic_function(base):
...    return lambda x : log(x,base)
# 创建一个以 3 为底的匿名对数函数，并赋值
>>>my_log = make_logarithmic_function(3)
# 调用匿名函数 my_log，底数已经设置为 3，只需设置真数
# 如果用 log 函数，则需要同时设置真数和底数
>>>print(my_log(9))
2.0
```

5.2.2 掌握其他常用高阶函数

除了 lambda 函数外，Python 中还有其他常用的高阶内置函数，如 map 函数、fib 函数和 filter 函数。由于列表综合使用的引入，reduce 函数在 Python 3 中被移到了 functools 模块，apply 函数也在逐步被淘汰，这里不再介绍。

1. map 函数

前文中的代码 5-22 用 map 函数也能实现，如代码 5-24 所示。

代码 5-24 使用 map 函数实现代码 5-22

```
>>>def add(x):
...    x += 3
...    return x
```

```
>>>numbers = list(range(10))
>>>num1 = list(map(add,numbers))
>>>num2 = list(map(lambda x:x+3,numbers))          # 速度快，可读性强
```

map 函数是 Python 内置的高阶函数，它的基本样式为 map(func,list)。其中，func 是一个函数，list 是一个序列对象。在执行的时候，序列对象中的每个元素，按照从左到右的顺序通过把函数 func 依次作用在 list 的每个元素上，得到一个新的 list 并返回。

注意

map 函数不改变原有的 list，只是返回一个新的 list。

2. fib 函数

fib 函数是一个递归函数，最典型的递归示例之一就是斐波那契数列。根据斐波那契数列的定义，可以直接写成斐波那契递归函数，fib 函数示例如代码 5-25 所示。

代码 5-25　fib 函数示例

```
>>>def fib(n):
...    if n <=2 :
...       return 2
...    else:
...       return fib(n-1)+fib(n-2)
>>>f = fib(10)
>>>print(f)
110
```

fib(n–1)+fib(n–2)就是调用了这个函数自己实现递归的。为了明确递归的过程，下面介绍一下计算过程（令 n＝3）。

（1）n=3，fib(3)，判断为计算 fib(3–1)+fib(3–2)。

（2）先看 fib(3–1)，即 fib(2)，返回结果为 2。

（3）再看 fib(3–2)，即 fib(1)，返回结果也为 2。

（4）最后计算第（1）步，结果为 fib(n–1)+fib(n–2)=2+2=4，将结果返回。

从而得到 fib(3)的结果为 4。从这个计算过程就可以看出，每个递归的过程都是向着最初的已知条件方向得到结果，然后一层层向上反馈计算结果。

3. filter 函数

filter 函数是 Python 内置的另一个常用的高阶函数。filter 函数接收一个函数 func 和一个 list，这个函数 func 的作用是对每个元素进行判断，通过返回 True 或 False 来过滤掉不符合条件的元素，符合条件的元素组成的新 list，filter 函数示例如代码 5-26 所示。

代码 5-26　filter 函数示例

```
>>>list(filter(lambda x:x%2==1, [1, 4, 6, 7, 9, 12, 17]))
```

```
>>>s = list(filter(lambda c:c!='o','i love python and R!'))
>>>s = ''.join(s)                                    # 变为字符型
>>>print(s)
```

Python 不是也不大可能成为一种函数式编程语言，它支持许多有价值的函数式编程语言构建，表现得也像函数式编程机制，但从传统上不能被认为是函数式编程语言的构建。如上高阶函数虽然对于程序的性能提高并没有显著的效果，但是在代码简洁方面的作用还是很明显的，也体现出了 Python 优雅简洁的特点。

5.2.3 任务实现

根据任务分析，本任务的具体实现过程可以参考如下操作。

（1）先用 def 关键字定义一个累加函数 add。

（2）创建一个空列表和一组数据[0,9]。

（3）用循环结构对列表进行数据累加后的元素添加（list.append）。

（4）使用匿名函数代替累加函数，再进行后续操作。

（5）使用 map 函数快速实现以上操作。

参考代码如任务实现 5-2 所示。

任务实现 5-2

```
#-*-coding:utf-8-*-

# 正常方式
def add(x):
    x += 3
    return x
new_numbers = []
for i in range(10):
    new_numbers.append(add(i))      #调用 add 函数，并 append 到 list 中
new_numbers

# lambda 函数方式（匿名函数）
lam = lambda x:x+3
n2 = []
for i in range(10):
    n2.append(lam(i))
n2
# 或者
lam = lambda x:x+3
n1 = []
[n1.append(lam(i)) for i in range(10)]
n1
```

```
# map 函数方式
numbers = list(range(10))
map(add,numbers)
aa = list(map(lambda x:x+3,numbers))
[aaa**2 for aaa in aa]   # 速度快，可读性强
```

任务 5.3　存储并导入函数模块

本章内容的开始提到了封装这个概念，本任务将实现简单的封装方法，将自定义函数封装为函数模块，然后导入模块，再调用里面的函数。

通过如下步骤可实现上述任务。

（1）将方差函数封装并命名。

（2）导入封装好的函数模块。

（3）导入模块中特定的函数。

（4）给函数指定别名。

（5）给函数模块指定别名。

（6）导入模块中的所有函数。

本任务通过引入模块调用写好的函数，这也是 Python 这门开源语言的一个特点。模块是最高级别的程序组织单元，它能够将程序代码和数据封装以便重用。模块往往对应了 Python 的脚本文件（.py），包含了所有该模块定义的函数和变量。模块可以被别的程序导入，以使用该模块的函数等功能，这也是使用 Python 标准库的方法。导入模块后，在该模块文件中定义的所有变量名都会以被导入模块对象的成员的形式被调用。换言之，模块文件的全局作用域变成了模块对象的局部作用域。

5.3.1　存储并导入整个模块

如果要导入模块的函数，需要先创建一个模块。创建一个包含 make_steak 函数的模块，然后将文件中的其他代码删除，如代码 5-27 所示。

代码 5-27　创建模块

```
>>>def make_steak(d,*other):
...    '''做一份牛排'''
...    print('Make a steak well done in %d ' % d + 'with the other:')
...    for o in other:
...        print('- '+o)
```

接下来，将该代码块在所在目录中保存为 steak.py。导入这个模块，并且调用里面的 make_steak 函数两次，如代码 5-28 所示。

代码 5-28 调用模块

```
>>>import steak
>>>steak.make_steak(9,'salad')
Make a steak well done in 9 with the other:
- salad
>>>steak.make_steak(8,'red wine','salad','coffee')
Make a steak well done in 8 with the other:
- red wine
- salad
- coffee
```

这是一种导入方法，用 import 语句导入指定的模块名，可在程序中使用该模块中的所有函数，但是需要有模块名的前缀。

5.3.2 导入函数

1. 导入指定函数

在 Python 中可以导入模块中的指定函数，也可以是多个指定函数。以 steak.py 为例，只导入需要使用的函数示例如代码 5-29 所示。

代码 5-29 导入指定模块

```
>>>from steak import make_steak
>>>make_steak(9,'salad')
Make a steak well done in 9 with the other:
- salad
>>>make_steak(8,'red wine','salad','coffee')
Make a steak well done in 8 with the other:
- red wine
- salad
- coffee
```

若使用这种方法，调用函数时就不需要加模块的前缀，直接调用它的函数名称即可，但如果模块中的函数较多，则这种方法比较烦琐。

2. 导入所有函数

如果模块中的函数较多，并需要导入所有函数，使用星号（*）运算符，如代码 5-30 所示。

代码 5-30 导入所有函数

```
>>>from steak import *
>>>make_steak(9,'salad')
Make a steak well done in 9 with the other:
- salad
>>>make_steak(8,'red wine','salad','coffee')
```

```
Make a steak well done in 8 with the other:
- red wine
- salad
- coffee
```

import 语句中星号的作用是将 steak 模块中的所有函数都导入当前程序中。采用没有模块前缀的方法可以调用模块中的所有函数。在编写大型程序时，最好不要采用这种导入方式，因为如果模块中的函数名称和项目程序中的名称相同，就会导致混乱或者程序出错等众多问题。

5.3.3 指定别名

1. 指定函数别名

如果导入的函数名称可能与程序中现有的名称冲突，或者名称太长，可用 as 语句在导入时给函数指定别名。给 make_steak 函数指定别名为 ms 函数的示例如代码 5-31 所示。

代码 5-31 指定函数别名

```
>>>from steak import as ms
>>>ms(9,'salad')
Make a steak well done in 9 with the other:
- salad
>>>ms(8,'red wine','salad','coffee')
Make a steak well done in 8 with the other:
- red wine
- salad
- coffee
```

代码 5-31 中的 import 语句将 make_steak 函数重命名为 ms 函数；每当需要调用 make_steak 函数时，都可以简写为 ms 函数，这样可以避免与程序中名称相同的函数产生混淆。

2. 指定模块别名

在 Python 中，不仅可以给函数指定别名，还可以给模块指定别名。通过给模块指定简短的别名（例如，为 steak 模块指定别名为 S），能够轻松地调用模块中的函数，相比 steak.make_steak 函数也更为简洁，如代码 5-32 所示。

代码 5-32 指定模块别名

```
>>>import steak as S
>>>S.make_steak(9,'salad')
Make a steak well done in 9 with the other:
- salad
>>>S.make_steak(8,'red wine','salad','coffee')
Make a steak well done in 8 with the other:
- red wine
- salad
```

```
- coffee
```

上述 import 语句给模块 steak 指定了别名 S，该模块中的所有函数名称都没变。调用 make_steak 函数时，编写 S.make_steak，而不是 steak.make_steak，不仅能使代码简洁，还可以不再关注与描述模块名。

关于 Python 的导入方法，最佳的是，只导入所需使用的函数，或者导入整个模块，并用前缀的方式表示。这能让代码更清晰，更容易阅读和理解。

5.3.4 任务实现

根据任务分析，本任务的具体实现过程可以参考如下操作。

（1）将前文任务 5.1 中的方差函数封装并命名为 var.py。

（2）导入封装好的函数模块。

（3）导入模块中特定的函数 var.var。

（4）给函数指定别名 fangcha。

（5）给函数模块指定别名 V。

（6）导入模块中的所有函数。

参考代码如任务实现 5-3 所示。

任务实现 5-3

```
# -*-coding:utf-8-*-

import var                                  # 导入封装好的函数模块 var
var1 = var.var(1,3,5,7,9,11,13)
print(var1)
from var import var                         # 导入模块中的函数 var.var
var2 = var(5,6,7,8,9)
print(var2)
from var import var as fangcha              # 给函数指定别名 fangcha
var3 = fangcha(1,2,3,4,5,6)
print(var3)
import var as V                             # 给函数模块指定别名 V
var4 = V.var(8,9,10,11)
print(var4)
from var import *                           # 导入模块中的所有函数
```

小结

Python 相对于其他语言而言，构建函数的风格更为简练。本章主要介绍了 Python 中命名函数的定义方法，从函数定义的格式上可以体现出简练的编程风格，其多种多样的参数设置方式提供了更加灵活的函数定义及调用方法。本章的主要知识点如下。

（1）Python 中定义函数时由关键字 def 声明，其后紧跟函数名和参数，参数存放在括

号中，之后紧跟冒号，函数体的缩进为 4 个空格，代码需严格按照缩进编写。

（2）Python 除设置了默认参数外，还提供了可变参数的方法，使得定义函数和调用函数更为灵活。*args 参数传入时存储在元组中，**kwargs 参数传入时存储在字典内。

（3）在函数内部可以内建函数，函数体内的局部变量仅在该层函数体内有效。变量转换为全局变量后，才可在全局使用，但是需要注意全局变量值的改变。

（4）使用 lambda 表达式可以创建匿名函数。匿名函数适用于定义不需要多次使用的简短函数。使用过于复杂的匿名函数，会影响函数的可读性。

（5）使用在 Python 中导入模块的方法，可以让代码更简洁，也更易于阅读和理解。

实训

实训 1　构建一个计算列表中位数的函数

1. 训练要点

（1）掌握自定义函数的定义方法。

（2）掌握自定义函数的构建格式。

（3）掌握可变参数*args 的使用方法。

2. 需求说明

中位数为常见的统计量之一，可将一个数集划分为相等的上下两部分。对于元素个数不同的列表而言，中位数的计算方式分为如下两种。

（1）若列表中元素的个数为奇数，则中位数为排序后列表中间位置的那个数。

（2）若列表中元素的个数为偶数，则中位数为排序后列表中间位置的两个数的均值。

3. 实训思路及步骤

（1）sorted 函数可对列表中的元素进行排序。

（2）使用下标可获取列表中对应位置的元素。

（3）列表中元素的个数为奇数时，中位数为列表正中间位置的那个数。

（4）列表中元素的个数为偶数时，中位数为列表正中间位置的两个数的均值。

实训 2　使用 lambda 表达式实现对列表中的元素求平方

1. 训练要点

（1）掌握 lambda 表达式的使用方法。

（2）掌握 lambda 表达式的应用场景。

2. 需求说明

平方为一个数乘以自身的乘积，本实训将对列表中的每一个元素求自身的平方。

3. 实训思路及步骤

（1）元素乘以自身即为自身的平方。

（2）map 函数可对列表中的每个元素分别应用求平方表达式。

课后习题

1. 选择题

（1）定义函数时，函数体的正确缩进为（　　）。

A．一个空格　　B．两个制表符　C．4 个空格　　D．4 个制表符

（2）可变参数*args 传入函数时的存储方式为（　　）。

A．元组　　B．列表　　C．字典　　D．数据框

（3）可变参数**kwargs 传入函数时的存储方式为（　　）。

A．元组　　B．字典　　C．列表　　D．数据框

（4）以下对自定义函数 def interest(money,day=1,interest_rate=0.05)调用错误的是（　　）。

A．interest(3000)　　B．interest(3000,3,0.1)

C．interest(day=2,3000,0.05)　　D．interest(3000,interest_rate=0.1,day=7)

（5）以下关于全局变量及局部变量描述错误的是（　　）。

A．全局变量可以被任意位置调用　　B．局部变量可以在外部被赋值

C．全局变量可以在任意位置被赋值　D．局部变量可以在外部被调用

（6）以下关于 lambda 表达式的描述错误的是（　　）。

A．lambda 表达式不允许多行　　B．lambda 表达式创建函数不需要命名

C．lambda 表达式解释性良好　　D．lambda 表达式可视为对象

（7）以下可改变原始变量，而不产生新变量的是（　　）。

A．map 函数　　B．filter 函数　C．sort 函数　　D．sorted 函数

（8）以下导入方式不需要加模块前缀的是（　　）。

A．import Numpy　　B．from Numpy import *

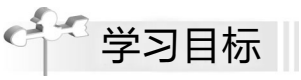
C．import Numpy as np　　D．from Numpy import matrix and array

2. 操作题

小说 *Walden* 的中文译名为《瓦尔登湖》，是美国作家梭罗独居瓦尔登湖畔时对生活的记录，描绘了他两年多时间里的所见、所闻和所思。该书崇尚简朴生活，热爱大自然的风光，内容丰富，意义深远，语言生动。请用 Python 统计小说 *Walden* 中各单词出现的频次，并按频次由高到低排序。

第6章 面向对象编程

本书前面几章介绍了 Python 中数据类型、数据结构、控制语句和函数的使用，要用 Python 尽心地开发工作，仅靠这些是不够的，还需要类和对象的概念及应用。Python 不只是解释性语言，也是一门面向对象的编程语言，因此自定义对象是 Python 语言的核心之一。本章将先介绍面向对象编程，再逐步讲解类和对象的定义、属性及方法。类使得程序设计更加抽象，通过类的继承及其他常用方法，可以让程序语言更接近人类的语言。



（1）认识面向对象编程的发展、实例、优点。
（2）了解使用面向对象编程的情形。
（3）掌握类的定义、使用和专有方法。
（4）掌握 self 参数的使用。
（5）掌握对象的创建（实例化）、删除。
（6）掌握对象的属性、方法引用和私有化方法。
（7）掌握迭代器和生成器。
（8）掌握类的继承、重载、封装等其他方法。

任务 6.1 认识面向对象编程

理解面向对象编程有助于理解类的意义，培养解决问题的逻辑思维。现在的面向对象编程增强了结构化，融合了数据和动作。数据层和逻辑层可以由简单抽象来描述。本任务中还可了解面向对象的发展，以及认识面向对象编程为什么是最有效的软件编写方法之一。

（1）了解面向对象的发展历程。
（2）体会面向对象中一个实例的分析思路。
（3）理解面向对象编程的优点。
（4）掌握何时使用面向对象编程。

6.1.1　了解面向对象编程及相关内容

1．面向对象编程

面向对象编程（Object Oriented Programming，OOP）即面向对象程序设计。类和对象是 OOP 中的两个关键内容。在面向对象编程中，以类来构造现实世界中的事物情景，再基于类创建对象来帮助进一步认识、理解、刻画。根据类来创建的对象，每个对象都会自动带有类的属性和特点，还可以按照实际需要赋予每个对象特有的属性，这个过程称为类的实例化。

抽象的直接表现形式通常为类。抽象指对现实世界的事物、行为和特征建模，建立一个相关的数据集用于描绘程序结构，从而实现这个模型。抽象不仅包括这种模型的数据属性，还定义了这些数据的接口。从面向对象设计（Object Oriented Design，OOD）角度去看，如果类是从现实对象抽象而来的，那么抽象类就是基于类抽象而来的，可以进行相似编码，或者编入与对象交互的对象中。从实现角度来看，抽象类与普通类的不同之处在于：抽象类中只能有抽象方法（没有实现功能），该类不能被实例化，只能被继承，且子类必须实现抽象方法。

2．面向对象方法

面向对象方法（Object Oriented Method），简称 OO 方法，是在软件开发过程中以“对象”为中心，用面向对象的思想来指导开发活动的系统方法。正如研究 OO 方法的专家和学者所说，近十多年来，OO 方法正在像 20 世纪 70 年代的结构化方法对计算机技术应用所产生的巨大影响和促进那样，一直在强烈地影响和促进一系列高技术的发展和多学科的综合。

面向对象方法起源于面向对象的编程语言。20 世纪 50 年代后期，在编写大型程序时，常出现相同变量名在程序的不同部分发生冲突的问题。对于这个问题，ALGOL 语言的设计者在 ALGOL60 中用“Begin…End”为标志，形成局部变量，避免它们与程序中其他同名变量相冲突。这是编程语言中首次进行封装的尝试，后来此结构广泛用于高级语言（如 Pascal、Ada、C）之中。

1986 年，首届“面向对象编程、系统、语言和应用（OOPSLA'86）”国际会议在美国举行，使面向对象的概念受到世人瞩目，其后每年一届，这标志着面向对象方法的研究已普及全世界。面向对象方法已被广泛应用于程序设计语言、数据库、设计方法学、人机接口、操作系统、分布式系统、人工智能、实时系统、计算机体系结构以及并发工程、综合集成工程等众多领域，而且都得到了很大的发展。例如，现代的面向对象程序设计方法使得设计模式的用途、契约式设计和建模语言（如 UML）技术也得到了一定提升。

3．面向对象编程语言

20 世纪 60 年代中后期，在 ALGOL 的基础上研制出现了 Simula 语言，提出了对象的概念，并使用了类，也支持类继承，面向对象程序设计的雏形得以形成。20 世纪 70 年代，经典的 Smalltalk 语言诞生，它以 Simula 的类为核心概念，以 Lisp 语言为主要内容。由于 Smalltalk 持续不断改进，引入了对象、对象类、方法、实例等概念和术语，采用动态联编和单继承机制，以至于现在都将这一语言视为面向对象的基础。

正是通过 Smalltalk 不断的改进与推广应用，人们才发现面向对象方法具有模块化、信息封装与隐蔽、抽象性、继承性、多态性等独特之处，为研制大型软件及提高软件可靠性、可重用性、可扩充性和可维护性都提供了有效的手段和途径。例如，分解和模块化可以给不同组件设定不同的功能，把一个问题分解成多个小的、独立的、互相作用的组件，来处理复杂、大型的软件。

从 20 世纪 80 年代起，面向对象程序设计成了一种主导思想，但一直没有专门面向对象程序设计的语言。人们将以前提出的有关信息封装与隐蔽、抽象数据类型等概念以及 BASIC、Ada 和 Smalltalk 和 Modula-2 等语言进行了糅合，却常常出现兼容性和维护性问题。后来因为客观需求的推动，大量理论研究和实践探索的进行，以及不同类型的面向对象语言（如 Eiffel、C++、Java、Object-Pascal 等）的产生及发展，逐步解决了这些问题。

在过去几年中，Java 语言成了广为应用的语言，除与 C 语言的语法近似外，还有面向对象编程的强大一面，即 Java 的可移植性。在近几年的计算机语言发展中，一些既支持面向过程程序设计（该怎么做）又支持面向对象程序设计（对象该怎么做）的语言开始崭露头角，如 Python、Ruby 等。

6.1.2 体会面向对象实例

面向对象出现以前，结构化程序设计是程序设计的主流。结构化程序设计又称为面向过程的程序设计。面向过程是分析出解决问题所需要的步骤，然后用函数一步一步实现这些步骤，使用的时候一个一个依次调用即可。而面向对象是把构成问题的事物分解成各个对象，建立对象的目的不是为了完成一个步骤，而是为了描述某个事物在整个解决问题的步骤中的行为。

例如五子棋，面向过程的设计思路就是首先分析问题的步骤，即开始游戏→黑子先走→绘制画面→判断输赢→轮到白子→绘制画面→判断输赢→返回步骤 2→输出最后结果，把每个步骤分别用函数来实现，问题就解决了；而面对对象的设计则是从另一种思路来解决问题，它将其分为 3 个对象：一是黑白双方，双方的行为是一模一样的；二是棋盘系统，负责绘制画面；三是规则系统，负责判断诸如犯规、输赢等。第一类对象负责接收用户输入信息，并告知第 2 类对象棋子布局的变化，棋盘对象接收到棋子的输入就要负责在画面上显示出这种变化，同时利用第 3 类对象来对棋局进行判定。

可以明显地看出，面向对象是以功能来划分问题的，而不是循环步骤。同样是绘制棋局，在面向过程的设计中，需要多个步骤执行该任务。但这样很可能导致不同步骤的绘制棋局的程序不同，因为设计人员会根据实际情况对绘制棋局的程序进行简化。而面向对象的设计中，绘图只可能在棋盘对象中出现，可以保证绘制棋局的统一。

6.1.3 了解面向对象的优点

在面向过程的程序设计中，问题被看作一系列需要完成的任务，解决问题的焦点集中于函数。其中的函数是面向过程的，即它关注如何根据规定的条件完成指定的任务。在多函数程序中，许多重要的数据被放置在全局数据区，这样它们可以被所有函数访问。但每个函数都只具有自己的局部数据。这样的程序结构很容易造成全局数据在无意中被其他函数改动，从而导致程序的不准确性。

面向对象程序设计的出发点之一就是弥补面向过程程序设计中的一些缺点，对象是程序的基本元素，它将数据和操作紧密地连接在一起，并保护数据不会被外界的函数意外地改变。因此面向对象有如下优点。

（1）基于数据抽象的概念可以在保持外部接口不变的情况下对内部进行修改，从而减少甚至避免对外界的干扰。

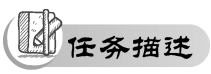
（2）通过继承可以大幅减少冗余代码，并可以方便地拓展现有代码，提高编码效率，也降低了出错概率，降低了软件维护难度。

（3）结合面向对象分析、面向对象设计，允许将问题中的对象直接映射到程序中，减少了软件开发过程中中间环节的转换过程。

6.1.4　何时使用面向对象编程

面向对象的程序与人类对事物的抽象理解密切相关。举一个例子，虽然不知道精灵宝可梦这款游戏（又名口袋妖怪）的具体代码，但可以确定的是，它的程序是根据面向对象的思路编写的。游戏中的每种精灵被看作一个类，具体的某只精灵就是其中一个类的一个实例对象，所以每种精灵的程序具有一定的独立性。程序人员可以同时编写多只精灵的程序，它们之间不会相互影响。为什么这里不能使用面向过程编程？读者可以思考一下，如果程序员要开发新的精灵，那么必须对之前的程序做大规模的修改，以使程序的各个函数能够正常工作，因为以前的函数没有新精灵的数据，工作量会大很多。现在的程序和软件开发都是使用面向对象编程的，最重要的原因还是其具有良好的抽象性。但对于小型程序和算法来说，面向对象的程序一般会比面向过程的程序慢，所以编写程序需要掌握两种思想，发挥它的长处。

任务 6.2　创建 Car 类



创建一个 Car 类，为其赋予车轮数（4）、颜色（red）的属性，并定义函数来输出“汽车有 4 个车轮，颜色是红色。”及“车行驶在学习的大道上。”，再调用类的方法（函数）。

通过如下步骤可实现上述任务。

（1）创建 Car 类，添加车轮数和颜色两个属性。

（2）定义第 1 个函数，并增加参数 name，输出“name 有 4 个车轮，颜色是红色。”。

（3）定义第 2 个函数，输出“车行驶在学习的大道上。”。

（4）调用 Car 类，赋值于新变量。

（5）对新变量调用自己定义的两个函数。

6.2.1　定义和使用类

1. 类的定义

在面向对象的程序设计中，类是创建对象的基础，描述了所创建对象共有的属性和方

法。它同时也有接口和结构，接口可以通过方法与类或对象进行互操作，而结构表现出一个对象中有什么样的属性。这些都为面向对象编程的 3 个最重要的特性（封装性、继承性、多态性）提供了实现手段。

类的定义就像函数定义，只是用 class 语句替代了 def 语句，同样是在执行 class 的整段代码后这个类才会生效。进入类定义部分后，会创建出一个新的局部作用域，后面定义的类的数据属性和函数方法都是属于此作用域的局部变量。

2. 类的使用

定义一个类，格式如下。

```
class 类名:
    属性列表
    方法列表
```

在用 class 语句创建类时，只要把所需的属性列表和方法列表列出即可，如代码 6-1 所示。

代码 6-1　创建类

```
>>>class Cat():
...     """一次模拟猫咪的简单尝试"""
...     name = 'tesila'                          # 属性
...     age = 3                                  # 方法
...     def sleep(self):
...         """模拟猫咪被命令睡觉"""
...         print('%d 岁的%s 正在沙发上睡懒觉。'%(self.age, self.name))
...     def eat(self,food):
...         """模拟猫咪被命令吃东西"""
...         self.food = food
...         print('%d 岁的%s 在吃%s'%(self.age, self.name,self.food))
```

代码 6-1 创建的类很简单，尽管只有一些简单的方法，但是需要注意的地方比较多。首先根据约定，在 Python 中，首字母大写的名称指的是类（Car），这个类定义的括号中是空的，因为是从空白来创建这个类（Python 2.0 中没有括号）；如果名称是两个单词，那么两个单词的首字母都要大写，例如 class HotDog，这种被形象地称为“驼峰式命名”；函数和方法没有区别，每个人的习惯不同而已，函数名称一般用小写字母或者下划线符号连接，如“new_car”。

最后可能发现，类的函数（方法）的参数都有一个 self 参数，并默认为第 1 个参数，这也是在编程时需要注意的地方。

6.2.2　绑定 self

Python 的类的方法和普通的函数有一个很明显的区别，就是类的方法必须有个额外的参数（self），并且在调用这个方法的时候不必为这个参数赋值。Python 类方法的这个特别参数指代的是对象本身，而按照 Python 惯例，它用 self 来表示。

对代码 6-1 创建的类稍加修改，查看效果，如代码 6-2 所示。

代码 6-2　self 参数

```
>>>class Cat():
...    def sleep(self):
...        print(self)
>>>new_cat = Cat()
>>>print(new_cat.sleep())
<__main__.Cat object at 0x00000000098656D8>
```

self 代表当前对象的地址，能避免非限定调用时找不到访问对象或变量。当调用 sleep 等函数时，会自动把该对象的地址作为第 1 个参数传入；如果不传入地址，程序将不知道该访问哪个对象。

self 名称也不是必需的，在 Python 中，self 不是关键字，可以定义成 a、b 或其他名字。例如代码 6-3 利用 my_address 代替 self，一样不会出现错误。

代码 6-3　self 可以修改名称

```
>>>class Test:
...    def prt(my_address):
...        print(my_address)
...        print(my_address.__class__)
>>>t = Test()
>>>t.prt()
<__main__.Test object at 0x000000000980AF60>
<class '__main__.Test'>
```

简而言之，self 需要定义，但是在调用时会自动传入；self 的名字并不是规定死的，但最好还是按照约定使用 self。self 总是指调用时的类的实例。

6.2.3　掌握类的专有方法

无论任何类，都有类的专有方法，它们的特殊性由代码就能看出来，通常是用双下划线“__”开头和结尾。

访问类或者对象（实例）的属性和方法，要通过点号操作来实现，即 object.attribute，当然也可以实现对属性的修改和增加。展示类的属性及方法示例如代码 6-4 所示。

代码 6-4　查看类的属性及方法

```
>>>class Example():
...    pass
>>>example = Example()
>>>dir(example)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__']
```

```
>>>dir(Example)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__']
```

从代码 6-4 的结果可以看到，用 dir 函数可以查看类的属性和方法。由于在定义类的函数里面只有 pass 语句，所以列出的结果中都以双下划线“__”开头和结尾。

常用的类的专有方法如表 6-1 所示。

表 6-1　类的专有方法

类的专有方法	功　能	类的专有方法	功　能
__init__	构造函数，在生成对象时调用	__call__	函数调用
__del__	析构函数，释放对象时使用	__add__	加运算
__repr__	打印，转换	__sub__	减运算
__setitem__	按照索引赋值	__mul__	乘运算
__getitem__	按照索引获取值	__div__	除运算
__len__	获得长度	__mod__	求余运算
__cmp__	比较运算	__pow__	乘方

__getitem__和__setitem，像普通的方法 clear、keys 和 values 一样，它只是重定向到字典，返回字典的值。通常不用直接调用它，而使用相应的语法让 Python 来调用__setitem__。每个文件类型都可以拥有一个处理器类，这些类知道如何从一个特殊的类得到元数据。一旦知道了某些属性（如文件名和位置），处理器类就知道如何自动地得到其他属性。

__repr__是一个专有的方法，只有当调用 repr(instance)时被调用。repr 函数是一个内置函数，它用返回一个对象的字符串表示。

__cmp__在比较类实例时被调用，通常可以通过使用“==”比较任意两个 Python 对象，不只是类实例。

__len__在调用 len(instance)时被调用。len 是 Python 的内置函数，可以返回一个对象的长度。字符串对象返回的是字符个数；字典对象返回的是关键字的个数；列表或序列返回的是元素的个数。对于类和对象，定义__len__方法，可以自己定义长度的计算方式，然后调用 len(instance)，Python 则会将调用定义的__len__专用方法。

__del__在调用 del instance[key]时调用，它会从字典中删除单个元素。

__call__方法让一个类表现得像一个函数，可以直接调用一个类实例。

__setitem__方法可以让任何类像字典一样保存键-值对。

__getitem__方法可以让任何类表现得像一个序列。

任何定义了__cmp__方法的类都可以用==进行比较。在类的应用中，最常见的是将类实例化，再通过实例来执行类的专有方法。

6.2.4　任务实现

根据任务分析，本任务的具体实现过程可以参考如下操作。

（1）使用 class 语句创建 Car 类并命名，添加车轮数和颜色两个属性。

（2）使用 def 函数定义 getCarInfo 函数，增加参数 name，用 print 函数输出“name 有 4 个车轮，颜色是红色。”。

（3）使用 def 函数定义 run 函数，用 print 函数输出“车行驶在学习的大道上。”。

（4）调用 Car 类赋值于 new_car。

（5）对 new_car 调用 getCarInfo 函数和 run 函数。

参考代码如任务实现 6-1 所示。

任务实现 6-1

```
# -*-coding:utf-8-*-

class Car():                                          # 创建类
    """一次模拟汽车的简单尝试"""
    wheelNum = 4                                      # 增加属性
    color = 'red'
def getCarInfo(self,name):                            # 定义 getCarInfo 函数
        self.name = name
        print(self.name,'有%d 个车轮，颜色是%s。'%(self.wheelNum,self.color))
    def run(self):                                    # 定义 run 函数
        print('车行驶在学习的大道上。')
new_car = Car()                                       # 调用 Car 类
print(new_car.getCarInfo('Land Rover'))               # 调用 getCarInfo 函数
print(new_car.run())                                  # 调用 run 函数
```

任务 6.3　创建 Car 对象

任务描述

根据任务 6.2 介绍的方法创建 Car 类，先用构造器构造初始化类的实例对象，赋予属性 newWheelNum 和 newColor，并定义函数，实现输出“车在跑，目标:夏威夷。”，然后创建具体对象，访问对象的属性和方法（函数），最后用析构的方法删除对象。

任务分析

通过如下步骤可实现上述任务。

（1）先创建 Car 类。

（2）将 Car 类实例化，添加 newWheelNum 和 newColor 两个属性。

（3）定义一个函数，输出“车在跑，目标:夏威夷。”。

（4）定义析构方法，输出“---析构方法被调用---”。

（5）调用 Car 类，创建对象实例。

（6）访问对象属性并调用函数。

（7）用析构方法删除所创建的实例对象，并查看对象是否被删除。

6.3.1 创建对象

__init__是类的一个特殊的方法，每当根据类创建新实例时，Python 都会自动运行它。这就是一个初始化手段，Python 中的__init__方法用于初始化类的实例对象。__init__方法的作用在一定程度上与 C++的构造函数相似，但并不等于。使用 C++的构造函数，可创建一个类的实例对象，而 Python 执行__init__方法时，实例对象已被构造出来。__init__方法会在对象构造出来后自动执行，所以可以用于初始化所需要的数据属性。创建对象示例如代码 6-5 所示。

代码 6-5 创建对象

```
>>>class Cat():
...     """再次模拟猫咪的简单尝试"""
...     # 构造器方法
...     def __init__(self,name,age):
...         self.name = name                        # 属性
...         self.age = age
...     def sleep(self):
...         """模拟猫咪被命令睡觉"""
...         print('%d 岁的%s 正在沙发上睡懒觉。'%(self.age, self.name))
...     def eat(self,food):
...         """模拟猫咪被命令吃东西"""
...         self.food = food
...         print('%d 岁的%s 在吃%s'%(self.age, self.name,self.food))
```

代码 6-5 把属性 name 和 age 放入了__init__方法中并进行初始化，通过实参向 Cat 类传递名字和年龄。self 会自动传递，因此创建对象时只需给出后两个形参（name 和 age）的值即可。

6.3.2 删除对象

创建对象时，默认调用构造方法。当删除一个对象时，同样也会默认调用一个方法，这个方法为析构方法。__del__是类的另一个特殊的方法，当使用 del 语句删除对象时，会调用它本身的析构函数。另外，当对象在某个作用域中调用完毕，跳出其作用域的同时，析构函数也会被调用一次，释放内存空间。它的具体用法如代码 6-6 所示。

代码 6-6 删除对象

```
>>>class Animal():
...         # 构造方法
...     def __init__(self):
...         print( '---构造方法被调用---')
```

```
...        # 析构方法
...    def __del__(self):
...        print( '---析构方法被调用---')
>>>cat = Animal()
---构造方法被调用---
>>>print(cat)
<__main__.Animal object at 0x0000000009851400>
>>>del cat
---析构方法被调用---
>>>print(cat)
Traceback (most recent call last):
  File "<ipython-input-43-7c3364e51d98>", line 1, in <module>
    print(cat)
NameError: name 'cat' is not defined
```

6.3.3　掌握对象的属性和方法

学习了类的定义和方法后，可以尝试建立具体的对象来更深一层学习面向对象程序设计。以代码 6-5 构造的对象为例，创建实例对象示例如代码 6-7 所示。

代码 6-7　创建实例

```
>>>class Cat():
...    def __init__(self,name,age):
...        self.name = name
...        self.age = age
...    def sleep(self):
...        print('%d 岁的%s 正在沙发上睡懒觉。'%(self.age, self.name))
...    def eat(self,food):
...        self.food = food
...        print('%d 岁的%s 在吃%s。'%(self.age, self.name,self.food))
>>>cat1 = Cat('Tom', 3)                                  # 创建对象
>>>cat2 = Cat('Jack',4)
>>>print('Cat1 的名字为:',cat1.name)                      # 访问对象的属性
Cat1 的名字为: Tom
>>>print('Cat2 的名字为:',cat2.name)
Cat2 的名字为: Jack
>>>print(cat1.sleep())                                   # 访问对象的方法
3 岁的 Tom 正在沙发上睡懒觉。
>>>print(cat2.eat('fish'))
4 岁的 Jack 在吃 fish。
```

创建对象和调用一个函数很相似，可以使用类名作为关键字去创建一个类的对象。但是创建实例对象需要参数，实际上这是__init__函数的参数。__init__函数会自动将数据属

性进行初始化，然后调用相关函数返回需要的对象数据属性。

1. 对象的属性

对象的变量由类的每个实例对象拥有。因此每个对象有自己对这个域的一份备份，即它们不是共享的。在同一个类的不同实例中，即使对象的变量有相同的名称，也互不相关。通俗的说法就是，不同的对象调用该变量，其变量值改变后，对象之间互不影响。

对于类属性和对象属性，如果在类方法中引用某个属性，则该属性必定是类属性。而如果在实例对象方法中引用某个属性（不进行更改），并且存在同名的类属性，此时，若实例对象有该名称的对象属性，则对象属性会屏蔽类属性，即引用的是对象属性；若实例对象没有该名称的对象属性，则引用的是类属性。如果在实例对象方法中更改某个属性，并且存在同名的类属性，此时，若实例对象有该名称的对象属性，则修改的是对象属性；若实例对象没有该名称的实例属性，则会创建一个同名称的对象属性。要修改类属性，如果在类外，则可以通过类对象修改；如果在类里面，则只有在类方法中进行修改。

2. 对象的方法

（1）方法引用

对象的方法和类的方法是一样的。在定义类的方法时，程序没有为类的方法分配内存，而在创建具体实例对象时程序才会为对象的每个数据属性和方法分配内存。类的方法是由 def 定义的，具体定义格式与普通函数相似，只是类方法的第一个参数需要是 self 参数。用普通函数可以实现对对象函数的引用，如代码 6-8 所示。

代码 6-8　对象方法的引用

```
>>>cat1 = Cat('Tom', 3)
>>>sleep = cat1.sleep
>>>print(sleep())
3 岁的 Tom 正在沙发上睡懒觉。
>>>cat2 = Cat('Jack',4)
>>>eat = cat2.eat
>>>print(eat('fish'))
4 岁的 Jack 在吃 fish。
```

代码 6-8 虽然看上去调用了一个普通函数，但是 sleep 函数和 eat 函数是引用了 cat1.sleep()和 cat2.eat()的，这意味着程序还是隐性地加入了 self 参数。

（2）私有化

如果要获取对象的数据属性，那么并不需要通过 sleep、eat 等方法，直接在程序外部调用数据属性即可，示例如代码 6-9 所示。

代码 6-9　对象属性的私有化

```
>>>print(cat1.age)
3
>>>print(cat2.name)
Jack
```

尽管这似乎很方便，但是却违反了类的封装原则，因为对象的状态对于类外部应该是不可访问的。查看 Python 模块代码时会发现源码里面定义了很多类，模块中的算法通过使用类来实现是很常见的，如果使用算法时能够随意访问对象中的数据属性，那么很可能在不经意中修改了算法中已经设置的参数，这是很麻烦的。一般封装好的类都会有足够的函数接口供程序开发人员使用，所以程序开发人员没有必要访问对象的具体数据属性。

为防止程序开发人员无意中修改对象的状态，需要对类的数据属性和方法进行私有化。Python 不支持直接私有方式，但可以使用一些小技巧达到私有的目的。为了让方法的数据属性或方法变为私有，只需要在它的名字前面加上双下划线即可，修改前文创建的 Car 类代码示例如代码 6-10 所示。

代码 6-10　私有化方法

```
>>>class Cat():
...    def __init__(self,name,age):
...        self.__name = name
...        self.__age = age
...    def sleep(self):
...        """模拟猫咪被命令睡觉"""
...        print('%d 岁的%s 正在沙发上睡懒觉。'%(self.__age, self.__name))
...    def eat(self,food):
...        """模拟猫咪被命令吃东西"""
...        self.__food = food
...        print('%d 岁的%s 在吃%s。'%(self.__age, self.__name,self.__food))
...    def getAttribute(self):
...        return self.__name,self.__age
>>>cat1 = Cat('Tom', 3)                    # 创建对象
>>>cat2 = Cat('Jack',4)
>>>print('Cat1 的名字为:',cat1.name)         # 从外部访问对象的属性，会发现访问不了
Traceback (most recent call last):
  File "<ipython-input-57-c85f1f024f81>", line 1, in <module>
    print('Cat1 的名字为:',cat1.name)
AttributeError: 'Cat' object has no attribute 'name'
>>>print('Cat2 的名字为:',cat2.name)
Traceback (most recent call last):
  File "<ipython-input-58-526f8a44b064>", line 1, in <module>
    print('Cat2 的名字为:',cat2.name)
AttributeError: 'Cat' object has no attribute 'name'
>>>print(cat1.sleep())                     # 只能通过设置好的接口函数来访问对象
3 岁的 Tom 正在沙发上睡懒觉。
>>>print(cat2.eat('fish'))
4 岁的 Jack 在吃 fish。
>>>print(cat1.getAttribute())
```

```
('Tom', 3)
```

现在，在程序外部直接访问私有数据属性是不允许的，只能通过设定好的接口函数去调取对象的信息。不过，通过双下划线实现的私有化其实是“伪私有化”，实际上还要从外部访问这些私有数据属性，如代码 6-11 所示。

代码 6-11　访问私有化属性

```
>>>print(cat1._Cat__name)
Tom
>>>print(cat1._Cat__age)
3
```

Python 使用 name_mangling 技术将__membername 替换成_class_membername，在外部使用原来的私有成员时，会提示无法找到，而执行 cat1._Cat__name 则可以访问。简而言之，想让其他人无法访问对象的方法和数据属性是不可能的，程序开发人员也不应该随意使用从外部访问私有成员的 name_mangling 技术。

6.3.4　任务实现

根据任务分析，本任务的具体实现过程可以参考如下操作。

（1）使用 class 语句创建 Car 类。

（2）将 Car 类实例化，添加 newWheelNuw 和 newColor 两个属性。

（3）使用 def 函数定义 run 函数，用 print 函数输出“车在跑，目标:夏威夷。”。

（4）用 def__del__定义析构方法，用 print 函数输出“---析构方法被调用---”。

（5）调用 Car 类，创建对象并命名为 BMW。

（6）访问对象属性，调用 run 函数，并用 print 函数输出。

（7）用析构方法删除 BMW，并查看对象是否被删除。

参考代码如任务实现 6-2 所示。

任务实现 6-2

```
# -*-coding:utf-8-*-


class Car():
    # 构造器方法
    def __init__(self, newWheelNum, newColor):
        self.wheelNum = newWheelNum
        self.color = newColor
    # 方法（函数）
    def run(self):
        print ('车在跑，目标:夏威夷。')
    # 析构方法
    def __del__(self):
        print( '---析构方法被调用---')
```

```
BMW = Car(4, 'green')                          # 创建对象
print( '车的颜色为:',BMW.color)                  # 访问属性
print( '车轮子数量为:',BMW.wheelNum)
BMW.run                                        # 调用对象的 run 函数

del BMW                                        # 删除对象
print(BMW)                                     # 查看是否删除
```

任务 6.4　迭代 Car 对象

任务描述

流程控制语句中已经接触过迭代方式，这里说的是面向对象过程中对象的迭代。本任务将对 Car 类进行迭代，增加品牌（brand）和废气涡轮增压（T）两个属性，并依次输出所有属性。

任务分析

通过如下步骤可实现上述任务。

（1）在原有 Car 类上增加品牌（brand）和废气涡轮增压（T）两个属性。

（2）创建列表(brand,wheelnum,color,T)并赋值给变量。

（3）为迭代设置初始变量。

（4）分别定义返回对应属性值的函数。

（5）定义迭代函数，输出“品牌 车轮数 颜色 废气涡轮增压”，返回对象位置。

（6）定义迭代器的基本方法，用 if 语句进行判断，返回对应位置的属性。

（7）调用 Car 类，创建对象并命名。

（8）访问对象属性和调用迭代函数，并输出结果。

6.4.1　生成迭代器

迭代是 Python 最强大的功能之一，是访问集合元素的一种方式。之前接触到的 Python 容器对象都可以用 for 循环进行遍历，如代码 6-12 所示。

代码 6-12　for 循环

```
>>>for element in [1,2,3]:
...    print(element)
>>>for element in (1,2,3):
...    print(element)
>>>for key in {'one':1,'two':2}:
...    print(key)
>>>for char in '123':
...    print(char)
>>>for line in open('myfile.txt'):
...    print(line)
```

这种编程风格十分简洁。迭代器有两个基本的函数，即 iter 函数和 next 函数。假如 for 语句在容器对象上调用 iter 函数，则该函数会返回一个定义 next 函数的迭代对象，它会在容器中逐一访问元素。当容器遍历完毕，next 函数找不到后续元素时，会引发一个 StopIteration 异常，告知 for 循环终止，如代码 6-13 所示。

代码 6-13　iter 函数与 next 函数

```
>>>L = [1,2,3]
>>>it = iter(L)
>>>it
<list_iterator at 0xa9e0630>
>>>next(it)
1
>>>next(it)
2
>>>next(it)
3
>>>next(it)
Traceback (most recent call last):
  File "<ipython-input-9-2cdb14c0d4d6>", line 1, in <module>
    next(it)
StopIteration
```

迭代器（Iterator）是一个可以记住遍历的位置的对象，从第 1 个元素被访问开始，直到所有元素被访问完结束。要注意的是，迭代器只能往前，不会后退。

要将迭代器加入自己的类中，需要定义一个__iter__函数，它返回一个有 next 函数的对象。如果类定义了 next 函数，则__iter__函数可以只返回 self。仍以代码 6-5 创建的 Cat 类为例，通过迭代器能输出对象的全部信息，如代码 6-14 所示。

代码 6-14　iter 函数应用

```
>>>class Cat():
...    def __init__(self,name,age):
...        self.name = name
...        self.age = age
...        self.info = [self.name,self.age]
...        self.index = -1
...    def getName(self):
...        return self.name
...    def getAge(self):
...        return self.age
...    def __iter__(self):
...        print('名字 年龄')
...        return self
```

```
...    def next(self):
...        if self.index == len(self.info)-1:
...            raise StopIteration
...        self.index += 1
...        return self.info[self.index]

>>>newcat = Cat('coffe', 3)                          # 创建对象
>>>print(newcat.getName())                           # 访问对象的属性
coffe
>>>iterator = iter(newcat.next,1)                    # 调用迭代函数来输出对象的属性
>>>for info in iterator:
...    print(info)
coffe
3
```

6.4.2　返回迭代器

1. yield

在 Python 中，使用生成器（Generator）可以很方便地支持迭代器协议。生成器是一个返回迭代器的函数，它可以通过常规的 def 函数来定义，但是不用 return 语句返回，而是用 yield 语句一次返回一个结果。一般的函数生成值后会退出，但生成器函数在生成值后会自动挂起暂停执行状态并保存状态信息。这些信息在函数恢复时将再度生效，通过在每个结果之间挂起和继续它们的状态自动实现迭代协议。

这里用一个示例（yield 实现斐波那契数列）来区分有 yield 语句和没有 yield 语句的情况，对生成器进一步了解，如代码 6-15 和代码 6-16 所示。

代码 6-15　生成器函数——斐波那契数列

```
>>>import sys
>>>def fibonacci(n,w=0):
...    a, b, counter = 0, 1, 0
...    while True:
...        if (counter > n):
...            return
...        yield a
...        a, b = b, a + b
...        print('%d,%d' % (a,b))
...        counter += 1
>>>f = fibonacci(10,0)           # f 是一个迭代器，由生成器返回生成
>>>while True:
...    try:
...        print (next(f), end=" ")
```

```
...    except :
...        sys.exit()
0 1,1
1 1,2
1 2,3
2 3,5
3 5,8
5 8,13
8 13,21
13 21,34
21 34,55
34 55,89
55 89,144
```

代码 6-16　生成器函数——斐波那契数列

```
>>>import sys
>>>def fibonacci(n,w=0):
...    a, b, counter = 0, 1, 0
...    while True:
...        if (counter > n):
...            return
...        # yield a                    # 不执行 yield 语句
...        a, b = b, a + b
...        print('%d,%d' % (a,b))
...        counter += 1
>>>f = fibonacci(10,0)                  # f 是一个迭代器，由生成器返回生成
>>>while True:
...    try:
...        print (next(f), end=" ")
...    except :
...        sys.exit()
1,1
1,2
2,3
3,5
5,8
8,13
13,21
21,34
34,55
55,89
```

```
89,144
```

在调用生成器并运行的过程中，每次遇到 yield 时，函数都会暂停并保存当前所有运行信息，返回 yield 的值，并在下一次执行 next 函数时从当前位置继续运行。

简而言之，包含 yield 语句的函数会被特地编译成生成器，当函数被调用时，返回一个生成器对象，这个对象支持迭代器接口。

2. 生成器表达式

列表解析的一般形式如下。

```
expr for iter_var in iterable if cond_expr
```

迭代 iterable 里所有内容时，每一次迭代后，把 iterable 里面满足 cond_expr 条件的内容放到 iter_var 中，再在表达式 expr 中应用 iter_var 的内容，最后用表达式的计算值生成一个列表。

例如，生成一个 list 来保护 50 以内的所有奇数。

```
[i for i in range(50) if i%2]
```

当序列过长，而每次只需要获取一个元素时，应当考虑使用生成器表达式，而不是列表解析。生成器表达式的语法和列表解析一样，只不过生成器表达式是被圆括号()括起来的，而不是方括号[ ]。

```
(expr for iter_var in iterable if cond_expr)
```

6.4.3　任务实现

根据任务分析，本任务的具体实现过程可以参考如下操作。

（1）在原有 Car 类上增加品牌（brand）和废气涡轮增压（T）两个属性。

（2）使用方括号创建列表[brand,wheelnum,color,T]并赋值给变量（info）。

（3）为迭代设置初始变量（index）。

（4）使用 def 函数分别定义 getBrand、getNewheelnum、getNewcolor、getT 函数，用 return 语句返回对应的属性值。

（5）使用 def 函数定义__iter__函数，用 print 函数输出“品牌 车轮数 颜色 废气涡轮增压”，返回对象位置。

（6）使用 def 函数定义 next 函数，用 if 语句进行判断，返回对应位置的属性。

（7）调用 Car 类，创建对象并命名为 newcar。

（8）访问对象属性，调用 iter 函数，并用 print 函数输出结果。

参考代码如任务实现 6-3 所示。

任务实现 6-3

```
# -*-coding:utf-8-*-

class Car():
    def __init__(self,brand, newWheelNum, newColor,T):
        self.brand = brand
```

```
        self.wheelNum = newWheelNum
        self.color = newColor
        self.T = T                                  # T 为废气涡轮增压
        self.info = [self.brand,self.wheelNum,self.color,self.T]
        self.index = -1
    def getBrand(self):
        return self.brand
    def getNewheelnum(self):
        return self.wheelNum
    def getNewcolor(self):
        return self.color
    def getT(self):
        return self.T
    def __iter__(self):
        print('品牌 车轮数 颜色 废气涡轮增压')
        return self
    def next(self):
        if self.index == 3:
            raise StopIteration
        self.index += 1
        return self.info[self.index]

newcar = Car('BMW',4, 'green',2.4)                  # 创建对象
print(newcar.getNewcolor())                         # 访问属性
iterator = iter(newcar.next,1)                      # 调用迭代函数输出对象的属性
for info in iterator:
    print(info)
```

任务 6.5 产生 Land_Rover 对象（子类）

任务描述

本任务将在任务 6.4 创建的 Car 类上产生子类 Land_Rover，而且要使子类 Land_Rover 拥有两个父类属性（品牌、颜色）和两个自带属性（车轮数、废气涡轮增压），然后输出子类属性。

任务分析

通过如下步骤可实现上述任务。

（1）创建子类 Land_Rover。

（2）使用构造方法创建对象，设置品牌、颜色两个父类参数和两个自带参数。

（3）在子类中调用父类构造函数。

（4）调用子类，创建对象并命名。

（5）访问对象属性，调用迭代函数，并输出结果。

6.5.1　继承父类属性和方法

1．继承

面向对象编程带来的好处之一是代码的重用，实现这种重用的方法之一是继承机制。继承（Inheritance）是两个类或多个类之间的父子关系，子类继承了父类的所有公有数据属性和方法，并且可以通过编写子类的代码扩充子类的功能。继承实现了数据属性和方法的重用，减少了代码的冗余度。

在程序中，继承描述的是事物之间的所属关系，例如猫和狗都属于动物，程序中便可以描述为猫和狗继承自动物；同理，波斯猫和巴厘猫都继承自猫，而沙皮狗和斑点狗都继承自狗，如图 6-1 所示。

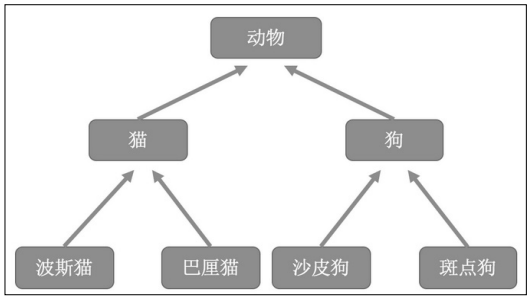


图 6-1　继承

特定狗种类继承狗类，狗类继承动物类，狗类编写了描述所有狗种共有的行为和方法，而特定狗种类则增加了该狗种特有的行为。不过继承也有一定的弊端，可能父类对于子类也有一定特殊的地方，如某种特定狗种不具有绝大部分狗种的行为，当程序员没有理清类间的关系时，可能会使得子类具有了不该有的方法。另外，如果继承链太长，任何一点小的变化都会引起一连串变化，所以使用的继承要注意控制继承链的规模。

在 Python 中，继承有以下一些特点。

（1）在继承中，基类初始化方法__init__不会被自动调用。如果希望子类调用基类的__init__方法，需要在子类的__init__方法中显示调用它。这与 C++和 C#区别很大。

（2）在调用基类的方法时，需要加上基类的类名前缀，且带上 self 参数变量。注意，在类中调用在该类中定义的方法是不需要 self 参数的。

（3）Python 总是首先查找对应类的方法，如果在子类中没有对应的方法，Python 才会在继承链的基类中按顺序查找。

（4）在 Python 继承中，子类不能访问基类的私有成员。

利用继承机制修改类 Cat 的代码，如代码 6-17 所示。

代码 6-17　添加继承方法

```
>>>class Cat():
...    def __init__(self):
```

```
...       self.name = '猫'
...       self.age = 4
...       self.info = [self.name,self.age]
...       self.index = -1
...    def run(self):
...       print( self.name,'--在跑')
...    def getName(self):
...       return self.name
...    def getAge(self):
...       return self.age
...    def __iter__(self):
...       print('名字 年龄')
...       return self
...    def next(self):
...       if self.index == len(self.info)-1:
...          raise StopIteration
...       self.index += 1
...       return self.info[self.index]
>>>class Bosi(Cat):
...   def setName(self, newName):
...       self.name = newName
...   def eat(self):
...       print( self.name,'--在吃')
...
>>>bs = Bosi()                                  # 创建对象
>>>print( 'bs 的名字为:',bs.name)                # 继承父类的属性和方法
bs 的名字为: 猫
>>>print( 'bs 的年龄为:',bs.age)
bs 的年龄为: 4
>>>print(bs.run())
猫 --在跑
>>>bs.setName('波斯猫')                         # 子类的属性和方法
>>>bs.eat()
波斯猫 --在吃
>>>iterator = iter(bs.next,1)                   # 迭代输出父类的属性
>>>for info in iterator:
...    print(info)
猫
4
```

代码 6-17 定义了 Bosi 类的父类 Cat，将猫共有的属性和方法都放到父类中，子类仅仅需要向父类传输数据属性。这样做可以很轻松地定义其他基于 Cat 类的子类。因为假如有数百只猫，使用继承的方法可以大大减少代码量，且当需要对全部猫整体修改时，仅修改 Cat 类即可。Bosi 类的__init__函数中显示调用了 Cat 类的__init__函数，并向父类传输数据，这里注意要加 self 参数。

在继承里面不能继承父类的私有属性，所以也不用担心父类和子类出现因继承造成的重名情况。为了能更清晰地说明这个问题，这里再举一个例子，如代码 6-18 所示。

代码 6-18　不能继承父类的私有属性

```
>>>class animal():
...    def __init__(self,age):
...        self.__age = age
...    def print2(self):
...        print(self.__age)
>>>class dog(animal):
...    def __init__(self,age):
...        animal.__init__(self,age)
...    def print2(self):
...        print(self.__age)

>>>a_animal = animal(10)
>>>a_animal.print2()
10
>>>a_dog = dog(10)
>>>a_dog.print2()                          # 程序报错
AttributeError: 'dog' object has no attribute '_dog__age'
```

2. 多继承

如果有多个父类，则父类名需要全部写在括号里，这种情况称为多继承，格式为 Class 子类名(父类名 1,父类名 2,…)，示例如代码 6-19 所示。

代码 6-19　多继承

```
>>>class A(object):                       # 定义一个父类
...    def __init__(self):
...        print ("   ->Input A")
...        print ("   <-Output A")
>>>class B(A):                            # 定义一个子类
...    def __init__(self):
...        print ("  -->Input B")
...        A.__init__(self)
```

```
...       print ("  <--Output B")
>>>class C(A):                              # 定义另一个子类
...    def __init__(self):
...       print (" --->Input C")
...       A.__init__(self)
...       print (" <---Output C")
>>>class D(B, C):                           # 定义一个子类
...    def __init__(self):
...       print ("---->Input D")
...       B.__init__(self)
...       C.__init__(self)
...       print ("<----Output D")

>>>d = D()                    # Python 中是可以多继承的，子类会继承父类中的方法、属性
---->Input D
  -->Input B
   ->Input A
   <-Output A
  <--Output B
 --->Input C
   ->Input A
   <-Output A
 <---Output C
<----Output D
>>>issubclass(C,B)           # 判断一个类是不是另一个类的子孙类
False
>>>issubclass(C,A)
True
```

实现继承之后，子类将继承父类的属性，也可以使用内建函数 issubclass 来判断一个类是不是另一个类的子孙类，前项参数为子类，后项参数为父类。

6.5.2 掌握其他方法

面向对象的三大特性是指重载、封装和多态。

1. 重载

所谓重载，就是子类中有一个和父类名字相同的方法，子类中的方法会覆盖父类中同名的方法，示例如代码 6-20 所示。

代码 6-20 重载

```
>>>class Cat:
...    def sayHello(self):
```

```
...       print("喵-----1")
>>>class Bosi(Cat):
...    def sayHello(self):
...       print("喵喵----2")

>>>bosi = Bosi()
>>>bosi.sayHello()                    # 子类中的方法会覆盖父类中同名的方法
喵喵----2
```

2. 封装

既然 Cat 实例本身就拥有这些数据，要访问这些数据，就没有必要从外部的函数去访问，可以直接在 Cat 类的内部定义访问数据的函数。这样，就把数据给“封装”起来了。

封装（Encapsulation）就是将抽象得到的数据和行为（或功能）相结合，形成一个有机的整体（即类）。封装的目的是增强安全性和简化编程，使用者不必了解具体的实现细节，而只是通过外部接口和特定的访问权限来使用类即可。简而言之，就是将内容封装到某个地方，以后再去调用被封装在某处的内容。

3. 多态

多态性（Polymorphism）是允许将父对象设置成和一个或多个它的子对象相等的技术，比如 Car=Land_Rover。多态性使得能够利用同一类（父类）类型的指针来引用不同类的对象，以及根据所引用对象的不同，以不同的方式执行相同的操作。

Python 是动态语言，可以调用实例方法，不检查类型，只需要方法存在，参数正确就可以调用，这就是与静态语言（例如 Java）最大的差别之一，表明了动态（运行时）绑定的存在。

6.5.3　任务实现

根据任务分析，本任务的具体实现过程可以参考如下操作。

（1）使用 class 语句创建子类 Land_Rover。

（2）使用构造方法创建对象，设置品牌、颜色两个父类参数和两个自带参数。

（3）在子类中调用父类构造函数 Car__init__。

（4）调用子类 Land_Rover，创建对象 Luxury_car。

（5）访问对象属性，调用 iter 函数，并用 print 函数输出结果。

参考代码如任务实现 6-4 所示。

任务实现 6-4

```
# -*-coding:utf-8-*-

class Car():
    def __init__(self,brand, newWheelNum, newColor,T):
        self.brand = brand
        self.wheelNum = newWheelNum
```

```
        self.color = newColor
        self.T = T                              # T 为废气涡轮增压
        self.info = [self.brand,self.wheelNum,self.color,self.T]
        self.index = -1
    def getBrand(self):
        return self.brand
    def getNewheelnum(self):
        return self.wheelNum
    def getNewcolor(self):
        return self.color
    def getT(self):
        return self.T
    def __iter__(self):
        print('品牌 车轮数 颜色 废气涡轮增压')
        return self
    def next(self):
        if self.index == 3:
            raise StopIteration
        self.index += 1
        return self.info[self.index]

class Land_Rover(Car):
    def __init__(self,brand, newColor):
        self.brand = brand
        self.wheelNum = 4
        self.color = newColor
        self.T = 3
        Car.__init__(self,self.brand,self.wheelNum,self.color,self.T)

Luxury_car = Car('BMW',4, 'green',2.4)              # 创建对象
print(Luxury_car.getNewcolor())                     # 访问属性
iterator = iter(Luxury_car.next,1)                  # 调用迭代函数
for info in iterator:
    print(info)
```

小结

Python 面向对象编程是一种编程方式，此编程方式的实现基于对类和对象的使用。类是一个模板，模板中包装了多个“函数”以供使用；对象是根据模板创建的实例（即对象），实例用于调用被包装在类中的函数。拥有基于标准库的大量工具，能够使用低级语言作为其他库接口的 Python 已成为一种强大的、应用于其他语言与工具之间的胶水语言。本章介

绍了 Python 面向对象程序设计的发展及其思想，介绍了何时及怎样运用面向对象编程；实现了面向对象的核心“类与对象”的创建和使用，并拓展了面向对象常用的功能和方法。

实训

实训 1　在精灵宝可梦游戏中创建小火龙角色，对给出的各属性进行迭代和私有化

1. 训练要点

（1）掌握游戏角色的数据属性的设置，以及相应类方法的创建。

（2）掌握类和对象的创建，以及对象属性的私有化方法。

（3）掌握迭代函数的使用，让角色在升级时提升各个属性。

2. 需求说明

游戏角色的数据属性：名字（name）、性别（gender）、等级（level）、能力（status、HP、攻击、防御、特攻、特防、速度）、属性（type）。在能力中，除了 HP 为 level*2+10 之外，其他能力均为 level+5。

类方法有 getName 函数（获取角色的名字，返回类型:str）、getGender 函数（获取角色的性别，返回类型:str）、getStatus 函数（返回角色的能力，返回类型:list）、getType 函数（返回角色的属性）。

迭代升级变化：level_up 函数，每升一级，HP 增加 2 点，其他属性增加 1 点。

迭代函数：__iter__函数，输出“名字 属性 性别 等级 能力”。

逐一访问属性：next 函数，当迭代到最后一个属性时，返回结果。

3. 实训思路及步骤

（1）创建类 Charmander，设置各个属性。

（2）定义类的方法返回属性值。

（3）根据等级进行迭代，改变相应的属性值。

（4）调用类 Charmander，创建对象并命名为 pokemon1，访问对象属性，调用 iter 函数，输出结果。

实训 2　对小火龙游戏角色采用继承机制

1. 训练要点

（1）掌握继承的使用。

（2）掌握继承中方法和属性的特点。

2. 需求说明

定义 Charmander 类的父类 pokemon，将精灵共有的行为放到父类中，然后子类向父类传输数据属性。如果在子类没有对应的方法，Python 才会在继承链的基类中按顺序查找。

3. 实训思路及步骤

（1）创建父类 pokemon，设置小火龙角色的属性。

（2）创建子类 Charmander。

（3）调用子类 Charmander，创建对象并命名为 pokemon1，访问对象属性，调用 iter 函数，输出结果。

课后习题

1．选择题

（1）在面向对象程序设计的发展中引入了对象、对象类、方法、实例等概念和术语，采用动态联编和单继承机制，以至于被视为面向对象的基础的语言是（　　）。

A．Simula　　B．Smautalk　　C．BASIC　　D．Java

（2）以下不是面向对象编程优点的是（　　）。

A．保持外部接口不变的情况下对内部进行修改

B．可以通过继承大幅减少冗余代码，并可以方便地拓展现有代码，提高编码效率，也降低出错概率，降低软件维护难度

C．允许将问题中的对象直接映射到程序中，减少软件开发的转换过程

D．作为一种建模技术，没有很好地定义自己的适用范围

（3）以下关于 self 的说法不正确的是（　　）。

A．self 可有可无，它的参数位置也不确定

B．self 是可以修改的

C．self 代表当前对象的地址

D．self 不是关键词，也不用赋值

（4）对于 Python 中类的专有方法表述错误的是（　　）。

A．__setitem__可以像字典一样　　B．__getitem__可以像序列一样

C．__cmp__可以进行比较　　D．__call__可以进行赋值

（5）以下关于私有化方法的说法错误是（　　）。

A．可以用双下划线的方法表示

B．私有化之后外部就不能访问

C．私有化后不能进行迭代

D．双下划线用于避免与子类中的属性命名冲突

（6）以下不是迭代器基本方法的是（　　）。

A．next()　　B．iter()　　C．close()　　D．open()

（7）生成器都是 Iterator 对象，但 list、dict、str 虽然都是 Iterable，却不是 Iterator。若要把 Iterable 变成 Iterator，在 Python 中应该使用（　　）。

A．capitalize 函数　B．Iterable 函数　C．Iterator 函数　　D．iter 函数

（8）以下是生成器表达式的是（　　）。

A．[i**2 for i in range(1,11)]

B．(file for file in os.listdir('/var/log') if file.endswith('.log'))

C．[(x, y) for x in range(3) for y in range(x)]

D．[x for x in (y.doSomething() for y in lst) if x>0]

（9）在类的继承中，子类不能从父类中继承的是（　　）。

A．__init__函数　　B．__getName 函数

C．name 属性　　D．iter 函数

（10）A 的子类有 B、C，而 B 的子类有 D、E，E 的子类有 F。下面不属于 F 的父类的是（　　）。

A．A　　B．B　　C．C　　D．E

2．操作题

定义一个学生类，类属性包括姓名（name）、年龄（age）、成绩（course、语文、数学、英语、每科成绩的类型为整数）。在类方法中，使用 get_name 函数获取学生的姓名，返回 str 类型；使用 get_age 函数获取学生的年龄，返回 int 类型；使用 get_course 函数返回 3 门科目中的最高分数，返回 int 类型。写好类以后用 zm=Student('zhangming',20,[69,88,100])测试，并输出结果。

第7章 文件基础

本章将介绍如何处理文件和保存数据，以便让程序使用得更容易，使用户能够读取 TXT 或 CSV 格式的文件，以及输出为 TXT 或 CSV 格式的文件；介绍如何用 Python 编程实现对计算机文件及文件夹的查询、移动、复制与删除等操作；介绍如何对文件进行压缩与解压等。

学习目标

（1）认识文件的概念与类型。

（2）掌握在 Python 中读取整个数据和逐行读取数据的方法。

（3）掌握工作路径的设置。

（4）掌握.txt 文件和.csv 文件的读取、修改及保存的方法。

（5）掌握 os 模块与 shutil 模块，并利用这两个模块对文件及文件夹进行查询、删除与移动等操作。

任务 7.1 认识文件

任务描述

在用计算机工作或娱乐的过程中，会接触到各种格式的文件，其中常见的有文档.doc、图片.jpg 和视频.mp4 等，但很多时候还会遇到一些特殊的文件类型，并且不知道用什么软件打开。在使用 Python 进行文件管理之前，需要先了解 Python 操作的文件的概念，掌握不同文件类型的判断及打开方式。

任务分析

（1）认识文件概念。

（2）了解常见文件类型。

（3）掌握常见文件类型的打开方式。

7.1.1 文件的概念及类型

1. 文件的概念

文件是指记录在存储介质上的一组相关信息的集合，存储介质可以是纸张、计算机磁盘、光盘或其他电子媒体，也可以是照片或标准样本，还可以是它们的组合。

在本章内容中，对于文件若无特殊说明，主要是指计算机文件，即以计算机磁盘为载

体存储在计算机上的信息集合。

2. 文件的类型

计算机中的文件包含文档文件、图片、程序、快捷方式、设备程序等。为区分不同文件及不同文件类型，需要给不同的文件指定不同的文件名称。在 Windows 操作系统下，文件名称由文件主名和扩展名组成，扩展名由小圆点和 1 ~ 3 个字符组成。

如 Readme.txt 作为文件名时，Readme 是文件主名，.txt 为扩展名，表示这个文件是纯文本文件，所有文字处理软件或编辑器都可将其打开。

常见的文件扩展名及文件对应打开方式如表 7-1 所示。

表 7-1　常见文件扩展名及相应的打开方式

文件类型	扩　展　名	打开方式
文档文件	.txt	可用所有的文字处理软件或编辑器打开
	.doc	可用 Microsoft Word 及 WPS 等软件打开
	.hlp	可用 Adobe Acrobat Reader 打开
	.rtf	可用 Microsoft Word 及 WPS 等软件打开
	.html	可用浏览器、写字板打开，可查看其源代码
	.pdf	可用各种电子阅读软件打开
压缩文件	.rar	可用 Win、Rar 打开
	.zip	可用 Win、Zip 打开
	.gz、.z	Lunix 的压缩文件，可用 Win、Zip 打开
图形文件	.bmp、.gif、.jpg、.pic、.png、.tif	可用常用图像处理软件打开
声音文件	.wav	可用媒体播放器打开
	.aif、.au	可用常用声音处理软件打开
	.mp3	可用 Winamp 播放
	.wma、.mmf、.amr、.aac、.flac	—
动画文件	.avi	可用常用动画处理软件播放
	.mov	可用 Active Movie 播放
	.swf	可用 Flash 自带的 Players 程序播放
系统文件	.int、.sys、.dll、.adt	—
可执行程序文件	.exe、.com	—
映像文件	.map	用 OziExplorer 3.95.4h 可打开这种模拟场景的图像数据调用文件
备份文件	.bak、.old、.wbk、.xlk、.ckr_	—
临时文件	.tmp、.syd、._mp、.gid、.gts	—
模板文件	.dot	通过 Word 文档程序可打开
可执行批处理文件	.bat	通过记事本可打开

7.1.2 文件命名

Windows 系统下的文件命名规则如下。

（1）文件名最长可以使用 255 个字符。

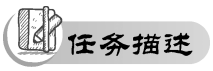
（2）可以使用扩展名，扩展名用来表示文件类型，也可以使用多间隔符的扩展名，其文件类型由最后一个扩展名决定。如 win.ini.txt 是一个合法的文件名。

（3）文件名中允许使用空格，但不允许使用英文输入法状态下的< > / \ | : " * ?。

（4）Windows 系统对文件名中大小写的字母在显示时会有不同，但在使用时不区分大小写。

需要注意的是，文件扩展名可以人为设定，扩展名为.txt 的文件也有可能是一张图片；同样，扩展名为.mp3 的文件，也可能是一个视频。但是人为修改文件扩展名可能会导致文件损坏。

任务 7.2 读取.txt 文件中的数据



用 Python 读取瓦尔登湖小说（Walden.txt，读者可通过书中配套数据查找相应文件）文件，并将文档保存到本章程序设置的文件夹里，然后用 Python 读取文件 Walden.txt 中的数据，并打印出来。

（1）掌握用 Python 读取整个文本文件的方式。

（2）掌握使用 with 语句读取文件的方法。

（3）掌握 Python 中相对路径与绝对路径之间的区别。

（4）掌握用 Python 逐行读取数据的方法。

（5）掌握文件数据读取成列表的方法。

7.2.1 读取整个文件

读写文件是最常用的 I/O（Input/Output）操作，Python 内置了读写文件的函数，用法是与 C 语言兼容的。

在读写文件之前，必须说明的是，在磁盘上面读取文件的功能是由操作系统提供的，现在的操作系统不容许普通的操作程序直接操作磁盘，所以读写文件就是请求操作系统打开一个文件对象（通常称为文件描述符），然后通过操作系统提供的接口从这个文件对象中读取数据（读文件），或者把数据写入这个文件对象（写文件），具体流程如图 7-1 所示。

要读取文件，需要先创建一个文件，它包含精确到小数点后 30 位的自然常数 e，且在小数点后的每 10 位处换行。

```
2.7182818284
5904523536
0287471352
```

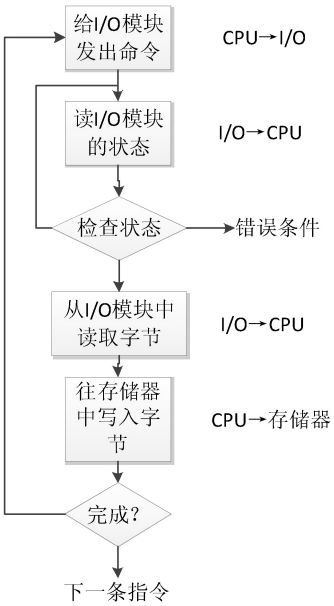


图 7-1　设备和内存之间的 I/O 控制

要以读文件的方式打开一个文件对象，可以使用 Python 内置函数中的 open 函数传入文件名称与标识符。其中，标识符可指定文件打开模式为读取模式（r）、写入模式（w）、附加模式（a）或读取和写入文件的模式（r+）。Python 默认以只读模式打开文件，如代码 7-1 所示。

代码 7-1　打开文件

```
>>>f = open('e_digits.txt', 'r')
```

如果读取的文件不存在，或者在当前工作路径下找不到要读取的文件，open 函数就会抛出一个 IOError 错误，并且给出错误码和详细的信息以说明文件不存在，如代码 7-2 所示。

代码 7-2　文件不存在

```
>>>f = open('not_exist.txt', 'r')
Traceback (most recent call last):
  File "file_read.py", line 2, in <module>
    f = open('not_exist.txt', 'r')
FileNotFoundError: [Errno 2] No such file or directory: 'not_exist.txt'
```

如果文件存在且程序可以正常打开文件，那么接下来就可以使用 read 函数一次性读取文件的全部内容，并将文件内容读入内存，然后用 print 函数将读取的文件内容打印出来，如代码 7-3 所示。

代码 7-3　读取文件

```
>>> f = open('e_point.txt', 'r')  # 打开 e_point.txt 文件并定义变量 f
>>>txt = f.read()  # 阅读文件 e_point.txt 的内容并赋值变量 txt
>>>print(txt)  # 输出文件 e_point.txt 的内容
```

```
2.7182818284
5904523536
0287471352
```

最后调用 close 函数关闭文件，如代码 7-4 所示。文件使用完毕后必须关闭，因为文件对象会占用操作系统的资源，并且操作系统同一时间能打开的文件数量也是有限的。

代码 7-4　关闭文件

```
>>>f .close()
```

7.2.2　使用 with 语句读取文件

在文件读取的过程中，一旦程序抛出 IOError 错误，后面的 close 函数将不会被调用。所以，在程序运行过程中，无论是否出错，都要确保能正常关闭文件，可以使用 try…finally…结构实现，如代码 7-5 所示。

代码 7-5　使用 try…finally…结构

```
>>>try:
...    f = open('e_point.txt', 'r')
...    print(f.read())
>>>finally:
...    if f:
...        f.close()
```

代码 7-5 虽然运行良好，但是每次都这样写实是太麻烦了，所以 Python 提供了更加优雅简短的语法。使用 with 语句可以很好地处理上下文环境产生的异常，会自动调用 close 函数，如代码 7-6 所示。

代码 7-6　使用 with 语句

```
>>>with open('e_point.txt', 'r') as f:
...    print(f.read())
```

这种 with 语句的使用效果与上文的 try…finally…结构的使用效果是一样的，但代码更为简洁，且不必调用 close 函数。

7.2.3　设置工作路径

在日常工作中，有时候需要打开不在程序文件所属目录下的文件，那么在程序里就需要提供文件所在路径，让 Python 到系统特定位置去查找并读取相应文件内容。

假如文件 e_point.txt 储存在文件夹 text_file 里面，而正在运行的 Python 程序储存在文件夹 Chapter7 里面。

1．相对文件路径

如果文件夹 text_file 是文件夹 Chapter7 的子文件夹，即文件夹 text_file 在文件夹 Chapter7 里面，那么需要提供相对文件路径让 Python 到指定位置查找文件，而该位置是相对于当前运行的程序所在的目录而言的，即相对文件路径，如代码 7-7 所示。

代码 7-7　相对文件路径

```
>>>with open('text_file\e_point.txt', 'r') as f:
...    print(f.read())
```

2. 绝对文件路径

如果将文件夹 text_file 放置到桌面，与文件夹 Chapter7 没有关系，那么需要提供完整准确的储存位置（即绝对文件路径）给程序，不需要考虑当前运行程序储存在什么位置，如代码 7-8 所示。

代码 7-8　绝对文件路径

```
>>>with open(r'C:\Users\45543\Desktop\ text_file\e_point.txt', 'r') as f:
...    print(f.read())
```

特别的，在绝对路径前面加了 r，这是因为在 Window 系统下，读取文件可以用反斜杠（\），但是在字符串中反斜杠被当作转义字符来使用，文件路径可能会被转义，所以需要在绝对文件路径前添加字符 r，显式声明字符串不用转义。

也可以采用双反斜杠（\\）的方式表示路径，此时不需要声明字符串，如代码 7-9 所示。

代码 7-9　双反斜杠方式

```
>>>with open('C:\\Users\\45543\\Desktop\\ text_file\\e_point.txt', 'r') as f:
...    print(f.read())
```

使用 Linux 路径表示方法是正斜杠（/），该方法也不需要声明字符串，在 Linux 及 Windows 操作系统下均可使用，如代码 7-10 所示。

代码 7-10　正斜杠方式

```
>>>with open('C:/Users/45543/Desktop/text_file/e_point.txt', 'r') as f:
...    print(f.read())
```

7.2.4　创建含有文件数据的列表

读取文件时，常常需要检查其中的每一行，可能需要在文件中查找特定的信息，或者需要以某种方式修改文件中的文本，此时可以对文件对象使用 for 循环，如代码 7-11 所示。

代码 7-11　for 循环来——读取

```
>>>file_name = 'e_point.txt'
>>>with open(file_name, 'r') as f:
...    for line_t in f:
...       print(line_t)
2.7182818284


5904523536


0287471352
```

在代码 7-11 中，将需要读取的文件名称赋值给 file_name 变量，这是为了方便修改文

件名称与路径，是使用文件时常见的做法。

此时，程序运行结果出现了很多空白行，空白行出现的原因是 e_point.txt 文档中每行末尾都有一个看不见的换行符，print 函数也给打印出来的数据加上了一个换行符。

如果需要消除换行符，可以使用 rstrip 函数删除 string 字符串末尾的指定字符（默认为空格）。与之关联的还有 lstrip 函数（删除字符前面的指定字符）和 strip 函数（删除字符串首尾两端的指定字符），如代码 7-12 所示。

代码 7-12　消除换行符

```
>>>file_name = 'e_point.txt'
>>>with open(file_name, 'r') as f:
...    for line_t in f:
...       print(line_t.rstrip())
2.7182818284
5904523536
0287471352
```

read 函数可以读取整个文件的内容，但是读取的内容将存储到一个字符串的变量中，如代码 7-13 所示。

代码 7-13　read 函数

```
>>>with open('e_point.txt') as f:
...    txts = f.read()
>>>print(type(txt))
<class 'str'>
>>>print(txt)
2.7182818284
5904523536
0287471352
```

如果需要将读取的文件存储到一个列表里面，可以使用 readlines 函数。该函数可以实现按行读取整个文件内容，然后将读取的内容存储到一个列表里面，如代码 7-14 所示。

代码 7-14　readlines 函数

```
>>>with open('e_point.txt') as f:
...    txts = f.readlines()
>>>print(type(txts))
<class 'list'>
>>>print(txts)
['2.7182818284\n', '5904523536\n', '0287471352\n']
```

为了让 readlines 函数存储的列表正常打印，可以做以下操作，如代码 7-15 所示。

代码 7-15　打印 readlines 函数存储的数据

```
>>> with open('e_point.txt') as f:
```

```
...    txts = f.readlines()
>>>for txt in txts:
...    print(txt.strip())
2.7182818284
5904523536
0287471352
```

此外，Python 还提供了 readline 函数，此函数可以实现每次读取文件的一行，通常也是将读取到的一行内容存储到一个字符串变量中，返回 str 类型，如代码 7-16 所示。

代码 7-16　readline 函数

```
>>>with open('e_point.txt') as f:
...    txt = f.readline()
>>>print(type(txt))
<class 'str'>
>>>print(txt)
2.7182818284
```

因为 readline 函数实现的是逐行读取，所以读取整个文件时，速度会比 readlines 函数慢，所以仅当没有足够内存读取整个文件时才会使用 readline 函数。

7.2.5　任务实现

实现打开并读取整个文件 Walden.txt，打印相关内容的代码如任务实现 7-1 所示。

任务实现 7-1

```
with open('Walden.txt ') as file_object:
    contects = file_object.readlines()
    print(contects)
```

任务 7.3　保存数据为 CSV 格式文件

任务描述

为实现数据写入文件，先使用 Python 创建列表数据，创建一个包含从 1 ~ 100 的平方的列表，如代码 7-17 所示；再将数据保存为 CSV 格式，并命名为 squares.csv。

代码 7-17　创建列表数据

```
>>>squares = [value**2 for value in range(1,101)]
>>>print(squares)
[4, 9, 16, 25, 36, 49, 64, 81, 100, …]
```

任务分析

（1）掌握数据写入 CSV 文件的方法。

（2）掌握数据写入 CSV 文件的格式设置方法。

（3）掌握读写 CSV 格式文件的方法。

7.3.1 写入.txt 文件

1. 数据写入文件

在 Python 的 open 函数中，标识符可指定文件打开模式，如果需要将数据写入文件，只需要将标识符设置为写入模式（w）即可。

如果要写入的文件不存在，那么 open 函数将自动创建文件。要注意的是，如果文件已经存在，那么以写入模式写入文件时会先清空该文件，如代码 7-18 所示。

代码 7-18 写入文件

```
>>>file_name = 'words.txt'
>>>f = open(file_name, 'w')
>>>f.write('Hello, world!')
>>>f.close()
```

代码 7-18 没有终端输出，但是可以在工作目录下打开 words.txt 文档来查看写入文档的内容，如表 7-1 所示。

表 7-1 word.txt

Hello, world!

需要注意的是，标识符 w 和 wb 表示写文本文件和写二进制文件（在 r 后面添加 b 表示要读二进制数据）。如果需要将数值型数据写入文本文件，必须先用 str 函数将数值型数据转换为字符串格式，如代码 7-19 所示。

代码 7-19 转换格式

```
>>> file_name = 'data.txt'
>>> f = open(file_name, 'w')
>>> data = list(range(1,11))
>>> f.write(data)
    Traceback (most recent call last):
      File "Ch7_read.py", line 12, in <module>
        f.write(data)
    TypeError: write() argument must be str, not list
>>>f.write(str(data))
>>>f.close()
```

写入内容后可查看写入的文档，如表 7-2 所示。

表 7-2 data.txt

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

需要注意的是，当写入多行数据的时候，write 函数不会自动添加换行符号，此时会出现几行数据挤在一起的情况，如代码 7-20 所示。

代码 7-20　write 函数

```
>>>file_name = 'words.txt'
>>>f = open(file_name, 'w')
>>>f.write('Hello, world!')
>>>f.write('I love Python!')
>>>f.close()
```

写入效果如表 7-3 所示，两行数据处于同一行。

表 7-3　words.txt

Hello, world! I love Python!

为了将行与行数据进行区分，需要在 write 语句内添加换行符号（\n），如代码 7-21 所示。

代码 7-21　添加换行符

```
>>>file_name = 'words.txt'
>>>f = open(file_name, 'w')
>>>f.write('Hello, world!\n')
>>>f.write('I love Python!\n')
>>>f.close()
```

添加换行符号后的写入效果如表 7-4 所示。

表 7-4　words.txt

Hello, world!
I love Python!

2. 使用 with 语句写入.txt 文件

在反复调用 write 来写入文件之后，务必要调用 close 函数来关闭文件。在写入文件的过程中，操作系统往往不会立刻把数据写入磁盘，而是放到内存中缓存起来，空闲的时候再慢慢写入。只有调用 close 函数时，操作系统才保证把没有写入的数据全部写入磁盘。忘记调用 close 函数的后果可能是数据只写了一部分到磁盘，剩下的丢失了。所以，还是用 with 语句来得保险，如代码 7-22 所示。

代码 7-22　写入文件

```
>>>file_name = 'words.txt'
>>>with open(file_name, 'w') as f:
...    f.write('Hello, world!\n')
...    f.write('I love Python!\n')
```

要写入特定编码的文本文件，需要给 open 函数传入 encoding 参数，将字符串自动转换成指定编码。open 函数默认 encoding 参数为 UTF-8。要读取非 UTF-8 编码的文本文件，例如读取 GBK 编码的文件，需要给 open 函数传入 encoding 参数，如代码 7-23 所示。

代码 7-23　加入编码

```
>>>file_name = 'words.txt'
>>>f = open(file_name, 'w', encoding = 'gbk')
>>>f.write('Hello, world!')
```

3. 对文件添加内容

在编写代码时，可能需要给文件添加内容，但不覆盖文件原内容，这时候需要以附加模式（a）打开文件，此时写入的内容会附加到文件末尾，而不会覆盖原内容，如代码 7-24 所示。

代码 7-24　添加文件

```
>>>file_name = 'words.txt'
>>>with open(file_name, 'a') as f:
...    f.write("What's your favourite language?\n")
...    f.write('My favourite language is Python too.\n')
```

代码 7-24 实现了将添加的两行字符串附加到文件的末尾，文件效果如表 7-5 所示。

表 7-5　words.txt

Hello, world!
I love Python!
What's your favourite language?
My favourite language is Python too.

7.3.2　读写 CSV 文件

逗号分隔值（Comma-Separated Values，CSV）有时也称为字符分隔值，是一种通用的、相对简单的文件格式，常应用于在程序之间转移表格数据。

CSV 文件由任意数目的记录组成，记录间以某种换行符分隔；每条记录由字段组成，字段间的分隔符是其他字符或字符串，最常见的分隔符是逗号或制表符。

在编写程序时，可能需要将数据转移到 CSV 文件里面，此时可以考虑使用 Python 的内置模块——csv 模块。在程序中，用命令 import csv 可直接调用 csv 模块进行 CSV 文件的读写。

1. 读取 CSV 数据

在读取 CSV 数据之前，先选择一个用 CSV 文件格式储存的数据作为演示的例子，这里选择 iris 数据集。

iris 数据集即鸢尾花卉数据集，是常用的分类实验数据集，由 Fisher 在 1936 年收集整理。数据集包含 150 个样本，分为 3 类（Setosa、Versicolour、Virginica），每类 50 个样本，每个数据包含 4 个属性——花萼长度、花萼宽度、花瓣长度、花瓣宽度，具体数据如表 7-6 所示。

读取 CSV 文件之前需要用 open 函数打开文件路径。

读取 CSV 文件的方法有两种。第一种是使用 csv.reader 函数，接收一个可迭代的对象（比如.csv 文件），能返回一个生成器，从其中解析出 CSV 的内容。

表 7-6　iris 数据集数据展示

编　　号	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5	3.4	1.5	0.2	setosa
9	4.4	2.9	1.4	0.2	setosa
10	4.9	3.1	1.5	0.1	setosa

以行为单位，利用 csv.reader 函数读取存储 iris 数据集的 iris.csv 文件的全部内容，并存储为列表，如代码 7-25 所示。

代码 7-25　读取 iris 数据

```
>>>import csv
file_name = 'C:\\Users\\45543\\Desktop\\iris.csv'
>>>with open(file_name, 'r') as f:
...    reader = csv.reader(f)
...    iris = [iris_item for iris_item in reader]
>>>print(iris)
[['', 'Sepal.Length', 'Sepal.Width', 'Petal.Length', 'Petal.Width', 'Species'],
['1', '5.1', '3.5', '1.4', '0.2', 'setosa'],
['2', '4.9', '3', '1.4', '0.2', 'setosa'],
['3', '4.7', '3.2', '1.3', '0.2', 'setosa'],
…]
```

第二种方法是使用 csv.DictReader 函数，该函数与 csv.reader 函数类似，接收一个可迭代的对象，能返回一个生成器，但是返回的每一个单元格都放在一个字典的值内，而字典的键则是这个单元格的标题（即列头）。

从代码 7-26 所示的代码中可以看出 csv.DictReader 函数读入的数据结构。

代码 7-26　csv.DictReader 函数读取数据

```
>>>import csv
file_name = 'C:\\Users\\45543\\Desktop\\iris.csv'
>>>with open(file_name, 'r') as f:
...    reader = csv.DictReader(f)
```

```
...    iris1 = [iris_item for iris_item in reader]
>>>print(iris1)
[OrderedDict([('', '1'), ('Sepal.Length', '5.1'), ('Sepal.Width', '3.5'),
('Petal.Length', '1.4'), ('Petal.Width', '0.2'), ('Species', 'setosa')]),
OrderedDict([('', '2'), ('Sepal.Length', '4.9'), ('Sepal.Width', '3'),
('Petal.Length', '1.4'), ('Petal.Width', '0.2'), ('Species', 'setosa')]),
OrderedDict([('', '3'), ('Sepal.Length', '4.7'), ('Sepal.Width', '3.2'),
('Petal.Length', '1.3'), ('Petal.Width', '0.2'), ('Species', 'setosa')]),…]
```

如果用 csv.DictReader 函数读取 CSV 文件的某一列，则可以用列的标题（Sepal.Length）查询，如代码 7-27 所示。

代码 7-27　读取 Name 列的内容

```
>>>with open(file_name, 'r') as f:
...    reader = csv.DictReader(f)
...    column = [iris_item['Sepal.Length'] for iris_item in reader]
>>>print(column)
['5.1', '4.9', '4.7', '4.6', '5', '5.4',…]
```

2. 写入.csv 文件

对于列表形式的数据，除了 csv.write 函数外，还需要用到 writerow 函数将数据逐行写入 CSV 文件。利用 csv.write 函数将数据写入 CSV 文件的示例如代码 7-28 所示。

代码 7-28　写入数据

```
>>>import csv
file_name = 'C:\\Users\\45543\\Desktop\\test.csv'
>>>with open(file_name, 'w', newline = '') as f:
...    write_csv = csv.writer(f)
...    write_csv.writerow(iris)
```

至于字典形式的数据，csv 模块提供了 csv.DictWriter 函数，除了提供 open 函数的参数外，还需要输入字典所有键的数据，然后通过 writeheader 函数在文件内添加标题，标题内容与键一致，最后使用 writerows 函数将字典内容写入文件，如代码 7-29 所示。

代码 7-29　写入字典内容

```
>>>import csv
file_name = 'C:\\Users\\45543\\Desktop\\test.csv'
>>>my_key = []                                    # 键的集合
>>>for i in iris1[0].keys():
...    my_key.append(i)
>>>with open(file_name, 'w', newline = '') as f:
...    write_csv = csv.DictWriter(f, my_key)
...    write_csv.writeheader()                    # 输入标题
...    write_csv.writerows(iris1)                 # 输入数据
```

7.3.3　任务实现

要让 1 ~ 100 的平方数列表以列的形式写入 squares.csv 文件中，需要先定义文件名称，并在写入的过程中用一个 for 循环逐行写入数据。要注意的是，默认在 open(file_name,'w') 模式下会有空行，在 open 后面添加参数 newline=‘’即可选择空行，如任务实现 7-2 所示。

任务实现 7-2

```
squares = [value**2 for value in range(1,101)]
import csv
file_name = 'C:\\Users\\45543\\Desktop\\squares.csv'
with open(file_name, 'w',newline='' ) as f:
    write_csv = csv.writer(f)
    for square in squares:
        write_csv.writerow([str(square)])
```

任务 7.4　认识 os 模块及 shutil 模块

任务描述

os 模块和 shutil 模块是常用的文件模块。通过编写 Python 程序，将代码 7-17 存储的 squares.csv 文件及任务实现 7-2 中的程序文件移动到当前工作目录下的 test_file 文件夹里，并压缩成 RAR 格式的压缩包。

任务分析

（1）认识 os 模块及 shutil 模块。

（2）掌握使用 os 模块对文件进行查询、创建、删除等操作。

（3）掌握使用 shutil 模块对文件夹或目录进行复制、移动、删除、压缩、解压等操作。

7.4.1　认识 os 模块

os 模块是 Python 标准库中的一个用于访问操作系统的模块，包含普遍的操作系统功能，如复制、创建、修改、删除文件及文件夹。os 模块提供了一个可移植的方法来使用操作系统的功能，使得程序能够跨平台使用，即它允许一个程序在编写后不需要任何改动，就可以在 Linux 和 Windows 等操作系统下都能运行，便于编写跨平台的应用。

1. 查询操作系统

在使用 os 模块的时候，如果需要获取当前操作系统，可以使用 os.name 函数获取操作系统的名称。如果是 Windows 系统，则返回 nt；如果是 Linux/UNIX 系统，则返回 posix。

使用 os.sep 函数可以查询相应操作系统下文件路径的分隔符。Windows 系统使用\\分隔路径，Linux 系统中的路径分隔符是/，而苹果 Mac OS 系统中的是:。

同样的，使用 os.linesep 函数可以查询当前系统使用的行终止符。Windows 系统使用\r\n，Linux 系统使用\n，而 Mac 系统使用\r，如代码 7-30 所示。

代码 7-30　查询操作系统

```
>>>import os
>>>os.name                          # 查询操作系统名称
'nt'
>>>os.sep                           # 查询文件路径的分隔符
'\\'
>>>os.linesep                       # 查询当前系统使用的行终止符
'\r\n'
```

2. 查询工作路径

如果不清楚 Python 的工作路径，可以使用 os 模块的 os.getcwd 函数进行查询，如代码 7-31 所示。

代码 7-31　查询工作路径

```
>>>path = os.getcwd()               # 查询当前工作目录，并赋值给 path
>>>print(path)
C:\\Users\\45543
```

3. 查询指定目录下的文件

使用 os 模块中提供的 os.listdir 函数可以查询指定目录下的所有文件和目录名，如代码 7-32 所示。

代码 7-32　查询指定路径下的文件

```
>>>os.listdir(path)                 # 查询当前工作目录下的文件
['.android',
 '.ipython',
 '.matplotlib',
 '.nuget',
 '.pylint.d',
…]
```

4. 删除文件

使用 os.remove 函数可以移除指定文件，如代码 7-33 所示。

代码 7-33　删除指定文件

```
>>>os.remove('C:\\Users\\45543\\Desktop\\test.csv')     # 删除指定文件
```

5. 创建与删除目录

创建与删除目录使用的是 os.mkdir 函数及 os.rmdir 函数。使用 os.rmdir 函数可以删除指定路径的文件夹，但是这个文件夹必须是空的，不包含任何文件或子文件夹，如代码 7-34 所示。

代码 7-34　创建与删除目录

```
>>> file_name = 'C:\\Users\\45543\\Desktop\\my_file'
>>> os.mkdir(file_name)                    # 创建文件夹
>>> os.rmdir(file_name)                    # 删除文件夹
```

6. 对文件路径的操作

os 模块里面含有 os.path 相关的函数，提供了相应的对文件路径的操作，具体函数及应用说明如表 7-7 所示。

表 7-7　os.path 相关的函数及应用说明

函数名称	函数应用
os.path.isdir(name)	判断 name 是不是目录，不是目录就返回 False
os.path.isfile(name)	判断 name 这个文件是否存在，不存在就返回 False
os.path.exists(name)	判断是否存在文件或目录 name
os.path.getsize(name)	获得文件大小，如果 name 是目录，则返回 0L
os.path.abspath(name)	获得绝对路径
os.path.isabs()	判断是否为绝对路径
os.path.normpath(path)	规范 path 字符串形式
os.path.split(name)	分隔文件名与目录（事实上，如果完全使用目录，它也会将最后一个目录作为文件名而使其分隔开，同时它不会判断文件或目录是否存在）
os.path.splitext()	分离文件名和扩展名
os.path.join(path,name)	连接目录与文件名或目录
os.path.basename(path)	返回文件名
os.path.dirname(path)	返回文件路径
os.path.split()	返回一个路径下的目录名和文件名
os.path.isfile() os.path.isdir()	分别检验给出的路径是一个目录还是文件
os.path.existe()	检验给出的路径是否真的存在

为了帮助读者更好地理解 os 模块，这里给出代码 7-35 所示的程序来展示 os 模块所能实现的功能。

代码 7-35　os 的功能实现

```
>>>import os
>>>system = os.name                                    # 获取操作系统名称
>>>if system == 'nt':
...    print('当前的操作系统是 Windows.')
>>>else:
```

```
...    print('当前的操作系统是 Linux.')
>>>print('本系统表示路径的分隔符是: ' + os.sep)
>>>print('本系统中使用的行终止符为: ' , [os.linesep])

>>>path=os.getcwd()  #获得当前目录
>>>print('运行本程序所在目录是:' + path)

>>>print('计算机的 Path 环境变量如下所示: \n' + os.getenv("Path"))

>>>os.mkdir("test")                                          # 创建空文件夹
>>>print('当前文件夹中的文件有: \n' , os.listdir(path))     # 获取文件夹中的所有文件
>>>if(os.path.exists("test")):                               # 判断文件是否存在
...    os.rmdir("test")                                      # 删除指定文件
...    print('删除成功夹')
>>>else:
...    print('文件夹不存在')
>>>print('删除后的结果: \n' , os.listdir(path))

>>>filepath1="Python7"
>>>if(os.path.isfile(filepath1)):    #判断是不是文件
...    print(filepath1 + "是一个文件")
>>>else:
...    print(filepath1 + "不是一个文件")

>>>name="Ch7.py"
>>>print("本程序的大小为" , os.path.getsize(name))          # 获取文件大小
>>>name=os.path.abspath(name)                               # 获取文件的绝对路径
>>>print("本程序的绝对路径是" + name)
>>>print("本程序的路径的文件名分别为: ", os.path.split(name))   #将文件名和路径分开
>>>files=os.path.splitext(name)                             # 将文件名和扩展名分开
>>>print("本程序的扩展为" + files[1])
>>>print("本程序的文件名为" + os.path.basename(name))       # 获取文件的名字
>>>print("本程序的路径为" + os.path.dirname(name))         # 获取文件的路径
```

7.4.2 认识 shutil 模块

os 模块不仅提供了新建文件、删除文件、查看文件属性的操作功能，还提供了对文件路径的操作功能。但是，对于移动、复制、打包、压缩、解压文件及文件夹等操作，os 模块没有提供相关的函数，此时需要用到 shutil 模块。

shutil 模块是对 os 模块中文件操作的补充，是 Python 自带的关于文件、文件夹、压缩文件的高层次的操作工具，类似于高级 API。

1. 移动文件或文件夹

使用 shutil.move 函数可以将指定的文件或文件夹移动到目标路径下，返回值是移动后的文件绝对路径字符串。

如果目标路径指向一个文件夹，那么指定文件将被移动到目标路径指向的文件夹中，并且保持其原有名字，如代码 7-36 所示。

代码 7-36　改变路径

```
>>> import shutil
>>>shutil.move('C:\\Users\\45543\\Desktop\\程序\\pi_digits.txt',
'C:\\Users\\45543\\Desktop')
'C:\\Users\\45543\\Desktop\\pi_digits.txt'
```

如果目标路径指向的文件夹中已经存在了同名文件，那么该文件将被重写；如果目标路径指向一个具体的文件，那么指定的文件在移动后将被重命名，如代码 7-37 所示。

代码 7-37　若存在，将被重写

```
>>> import shutil
>>>shutil.move('C:\\Users\\45543\\Desktop\\pi_digits.txt',
'C:\\Users\\45543\\ MyText.txt')
'C:\\Users\\45543\\ MyText.txt'
```

需要注意的是，目标路径下的文件夹必须是已经存在的，否则程序会返回错误。

2. 复制文件

shutil.copyfile(src,dst)可以从 src 文件复制内容（不包含元数据）到 dst 文件。dst 必须是完整的目标文件名。

返回值是复制后的文件绝对路径字符串，如代码 7-38 所示。

代码 7-38　复制文件

```
>>> import shutil
>>> shutil.copyfile('Ch7.py', 'Ch7.py.copy')
'Ch7.py.copy'
```

如果 src 和 dst 是同一文件，就会引发错误 shutil.Error。dst 文件必须是可写的，否则将引发异常 IOError。如果 dst 文件已经存在，则它会被替换。对于特殊文件，例如字符或块设备文件和管道不能使用此功能，因为 copyfile 会打开并读取文件。

shutil.copy(src,dst)可以复制文件 src 到文件或目录 dst。如果 dst 是目录，则会使用 src 相同的文件名创建（或覆盖），文件权限也会复制，返回值是复制后的文件绝对路径字符串，如代码 7-39 所示。

代码 7-39　跨路径复制文件

```
>>> import shutil
>>> shutil.copy('C:\\Users\\45543\\Desktop\\pi_digits.txt', 'C:\\Users\\45543\\
MyFile.txt')
'C:\\Users\\45543 \\MyFile.txt'
```

除了以上复制文件的函数，shutil 模块还提供了 shutil.copytree 函数用于进行目录的复制，如代码 7-40 所示。

代码 7-40　复制目录

```
>>> import shutil
>>> shutil.copytree('C:\\Users\\45543\\Desktop\\程序',
'C:\\Users\\45543\\test')
'C:\\Users\\45543\\test'
```

代码 7-40 中的函数返回结果为复制后目录的路径。“程序”文件夹下的目录将复制到 test 文件夹内。需要注意的是，test 文件夹必须事先不存在。

3．永久删除文件和文件夹

使用 os.unlink 函数会删除指定的文件。使用 os.rmdir 函数会删除路径指定的文件夹，但是这个文件夹必须是空的，不能包含任何文件或子文件夹。

使用 shutil.rmtree 函数可以删除路径指定的文件夹，并且这个文件夹里面的所有文件和子文件夹都会被删除。

因为涉及对文件与文件夹的永久删除，因此以上函数的使用必须要非常谨慎。代码 7-41 展示了在 Delect 文件夹含有子文件夹的情况下，使用上述删除函数的表现情况。

代码 7-41　删除文件和文件夹

```
>>> import shutil
>>>import os
>>>os.unlink('C:\\Users\\45543\\Desktop\\Delect')
Traceback (most recent call last):

  File "<ipython-input-20-3d33ebf24b19>", line 1, in <module>
    os.unlink('C:\\Users\\45543\\Desktop\\Delect')

PermissionError: [WinError 5] 拒绝访问。: 'C:\\Users\\45543\\Desktop\\Delect'

>>>os.rmdir('C:\\Users\\45543\\Desktop\\Delect')
Traceback (most recent call last):
  File "<ipython-input-21-841196aeb378>", line 1, in <module>
    os.rmdir('C:\\Users\\45543\\Desktop\\Delect')
OSError: [WinError 145] 目录不是空的。: 'C:\\Users\\45543\\Desktop\\Delect'

>>>shutil.rmtree('C:\\Users\\45543\\Desktop\\Delect')
```

4．压缩与解压文件

Python 2.7 版本之后就提供了使用 shutil 模块实现文件压缩与解压的功能。shutil.make_archive 函数的相关参数设置如表 7-8 所示。

表 7-8　shutil.make_archive 函数的相关参数

参　数	说　明
base_name	压缩包的文件名，也可以是压缩包的路径，是文件名时，则保存至当前目录，否则保存至指定路径
format	压缩包种类，可以是 zip、tar、bztar、gztar
root_dir	要压缩的文件夹路径（默认当前目录）
owner	用户，默认当前用户
group	组，默认当前组
logger	用于记录日志，通常是 logging.Logger 对象

将文件夹（C:\\Users\\45543\\Desktop 路径下的“程序”文件夹）压缩到指定路径（C:\\Users\\45543\\Desktop）下的示例如代码 7-42 所示。

代码 7-42　压缩文件

```
>>> import shutil
>>>  shutil.make_archive("C:\\Users\\45543\\Desktop\\test", 'zip', root_dir=
'C:\\Users\\45543\\Desktop\\程序')
```

对压缩文件进行解压处理，需要用到 shutil.unpack_archive 函数，示例如代码 7-43 所示。

代码 7-43　解压文件

```
>>> import shutil
>>> shutil.unpack_archive('C:\\Users\\45543\\Desktop\\test.zip', 'C:\\Users\\
45543\\Desktop\\test')
```

代码 7-43 中的 shutil.make_archive 函数将 test.zip 压缩包中的文件解压到了指定路径（C:\\Users\\45543\\Desktop）下的 test 文件夹里。

7.4.3　任务实现

任务 7.3 中将 squares.csv 文件储存在了路径“C:\\Users\\45543\\Desktop”下，本小节任务要求将程序文件存储在路径“\第 7 章 Python 文件基础\01-任务程序\code\任务实现 7.4.py”下。此时需要先把两个文件从不同路径下复制到当前工作目录下的 test_file 文件夹里，然后直接调用函数将 test_file 文件夹压缩为 RAR 格式的压缩包。

参考代码如任务实现 7-3 所示。

任务实现 7-3

```
import os
import shutil                                                    # 加载模块
file_name1 = 'C:\\Users\\45543\\Desktop\\squares.csv'            # 文件路径
file_name2 = 'C: \\Users\\45543\\\第 7 章 Python 文件基础\01-任务程序\code\任务实现
7.4.py '
out_file = os.getcwd() + '\\test'           # 目标路径（当前工作目录 + test 文件夹）
```

```
os.mkdir(out_file)                                          # 创建文件夹
shutil.copy(file_name1, out_file)
shutil.copy(file_name2, out_file)                           # 复制文件
shutil.make_archive('test7.4', 'zip', root_dir = out_file)  # 压缩文件
```

小结

本章首先阐述了 Python 读写.txt 文件的方法，然后介绍了如何使用内置 csv 模块进行 csv 格式文件的读写，详细介绍了 Python 读写.txt 文本文件及.csv 数据文件的函数及相应的使用方法。紧接着介绍了内置 os 模块及 shutil 模块，并阐述了如何使用 os 模块对文件内容进行查询、删除等操作，以及如何使用 shutil 模块对文件夹进行移动、复制、压缩、解压等操作。

实训

实训 1　计算 iris 数据集的均值

1. 训练要点

（1）掌握数据均值的计算方法。

（2）掌握将数据重新存储为.csv 文件的方法。

2. 需求说明

使用 iris 数据集（iris.csv）。iris 数据集中包含了 3 种鸢尾花的 4 个属性情况——花萼长度（Sepal.Length）、花萼宽度（Sepal.Width）、花瓣长度（Petal.Length）、花瓣宽度（Petal.Width），现需要计算 4 个属性的均值，然后将均值情况及原始数据存储到 test.csv 文件中。

3. 实现思路及步骤

（1）读取 iris.csv 文件，并存储为字典形式的数据。

（2）计算每一个属性的均值，并赋值给相应字典的相应键。

（3）把上述处理后的数据写入新建文件 my_iris.csv，确保数据与属性保持一致。

实训 2　编程实现文件在当前工作路径下的查找

1. 训练要点

（1）掌握 os 模块的使用。

（2）回顾自定义函数的使用。

（3）回顾打印结果的使用。

2. 需求说明

在查找文件的时候，通常使用系统自带的查找文件工具，通过关键字检索到当前目录下符合要求的文件。这里要求尝试通过编写 Python 程序，实现在当前目录以及当前目录的所有子目录下查找文件名包含指定字符串的文件的功能，并打印出相对路径。

3. 实现思路及步骤

（1）确定查找的对象与工作路径。

（2）自定义函数来判断文件是否符合要求。

（3）进一步查看工作路径下的子文件夹内的文件是否符合要求。

（4）打印查找到的相关信息。

课后习题

1. 选择题

（1）Python 描述路径时常见的 3 种方式不包含（　　）。

A. \\　　B. \　　C. /　　D. //

（2）以下不属于临时文件的是（　　）。

A. tmp　　B. syd　　C. _mp　　D. dot

（3）以下不属于 open 函数标识符可输入的参数是（　　）。

A. r　　B. rb　　C. w　　D. a+

（4）open 函数的默认 encoding 参数为（　　）。

A. utf-8　　B. utf-7　　C. gbk　　D. url

（5）以下函数不能读取文件的是（　　）。

A. read　　B. read　　C. readlines　　D. readline

（6）写入文本文件的数据类型必须是（　　）。

A. 字符型　　B. 数值型　　C. 浮点型　　D. 逻辑型

（7）.csv 文件默认的分隔符是（　　）。

A. 逗号　　B. 制表符　　C. 分号　　D. 顿号

（8）利用 csv.reader 函数读取的数据存储类型是（　　）。

A. 列表　　B. 向量　　C. 字典　　D. 元组

（9）os 模块不能进行的操作是（　　）。

A. 查询工作路径　B. 删除空文件夹C. 复制文件　　D. 删除文件

（10）shutil 模块不能进行的操作是（　　）。

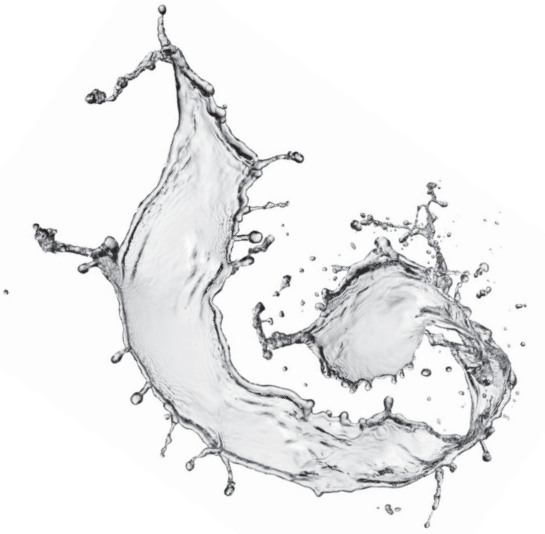
A. 移动文件夹　B. 创建文件夹　C. 压缩文件　　D. 删除非空文件夹

2. 操作题

（1）查找出工作目录下的所有 Python 程序文件（以.py 结尾的文件），然后将所有 Python 程序复制到新建文件夹 my_python 下，最后把 my_python 文件夹进行压缩，将压缩后的文件命名为 all python，并存储在计算机桌面。

（2）将 iris 数据集的数据进行 min-max 标准化（Min-max normalization），对原始数据进行线性变换，使结果落到[0,1]区间，转换函数为 x=(x-min)/(max-min)。其中，max 为样本数据的最大值，min 为样本数据的最小值。然后将得到的数据保存为 standard_iris.csv，并存储在计算机桌面。

大数据人才培养规划教材



大数据数学基础

数据库基础

大数据技术基础

数据采集

数据存储

数据分析

数据挖掘

数据可视化

免/费/提/供
PPT等教学相关资料

人邮教育
www.ryjiaoyu.com

教材服务热线：010-81055256
反馈/投稿/推荐信箱：315@ptpress.com.cn
人民邮电出版社教育服务与资源下载社区：www.ryjiaoyu.com

ISBN 978-7-115-47449-0

封面设计：董志桢