高等院校计算机应用系列教材

Python 程序设计基础

主　编　马亚丽

副主编　叶燕文　李　焱

王志强　任　洁

清華大學出版社

北　京

内 容 简 介

本书旨在讲述 Python 程序设计的基础知识。全书共 10 章，内容包括 Python 基础、编程基础、程序控制结构、组合数据类型、函数、文件、异常处理、常见第三方库、数据分析入门和 Python 实例。最后一章的每个实例都是经典的实际问题，让读者在学习相关章节后，运用所学知识来解决实际问题，助力读者提升实战技能。本书语言表述通俗易懂，案例习题配套丰富，可以让读者将所学的理论知识落地，帮助读者更好地掌握相关技术，可使读者随时随地开展自学，掌握 Python 程序设计相关知识与方法。

本书可作为高等院校计算机程序设计课程的教材，也可供渴望用编程解决实际问题但对编程缺乏基础的读者使用。

图书在版编目 (CIP) 数据

Python 程序设计基础 / 马亚丽主编 . -- 北京 :
清华大学出版社，2025. 8. -- ISBN 978-7-302-69944-6

Ⅰ. TP312.8

中国国家版本馆 CIP 数据核字第 2025EG9122 号

责任编辑：王　定
封面设计：周晓亮
版式设计：思创景点
责任校对：成凤进
责任印制：丛怀宇

出版发行：清华大学出版社
网　　址：https://www.tup.com.cn，https://www.wqxuetang.com
地　　址：北京清华大学学研大厦A座　　**邮　　编：**100084
社 总 机：010-83470000　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：大厂回族自治县彩虹印刷有限公司
经　　销：全国新华书店
开　　本：203mm×260mm　　**印　　张：**18　　**字　　数：**480千字
版　　次：2025年8月第1版　　**印　　次：**2025年8月第1次印刷
定　　价：69.80元

产品编号：112400-01

前言 PREFACE

党的二十大报告指出：“教育、科技、人才是全面建设社会主义现代化国家的基础性、战略性支撑。”“我们要坚持教育优先发展、科技自立自强、人才引领驱动，加快建设教育强国、科技强国、人才强国，坚持为党育人、为国育才，全面提高人才自主培养质量，着力造就拔尖创新人才，聚天下英才而用之。”

在当今数字化时代，无论是数据分析、人工智能、Web 开发，还是自动化运维、科学计算，Python 都能提供高效的解决方案。Python 以其简洁、易读、功能强大的特点，深受广大用户的青睐。对于初学者而言，Python 更是进入编程世界的理想选择——它降低了编程的门槛，却又不失其专业性和扩展性。

《Python 程序设计基础》旨在为编程零基础的读者提供一条清晰、系统的学习路径。本书不仅关注 Python 语法的基础知识，更注重培养读者的计算思维和实际编程能力。

本书共 10 章，内容包括 Python 基础、编程基础、程序控制结构、组合数据类型、函数、文件、异常处理、常见第三方库、数据分析入门和 Python 实例。本书的编写遵循以下原则。

(1) 循序渐进，由浅入深。本书从最基本的变量、数据类型、运算符讲起，逐步过渡到流程控制、函数、文件操作，最后到异常处理、数据分析等高级主题，确保读者能够扎实掌握 Python 的核心概念。

(2) 案例驱动，注重实践。每个知识点都配有典型示例代码，并结合实际应用场景进行讲解。书中还设计了丰富的练习题和实验，帮助读者巩固所学知识，提升动手能力。

(3) 结合现代 Python 特性。本书基于 Python 3.10 版本编写，涵盖 f- 字符串、上下文管理器等现代 Python 特性，确保读者学习的是当前主流技术。

(4) 培养编程思维，而非单纯记忆语法。编程不仅仅是写代码，更是解决问题的过程。本书在讲解知识点的同时，引导读者思考如何分解问题、设计算法、优化代码，从而使读者真正掌握编程的核心方法。

(5) 拓展应用场景，激发学习兴趣。在掌握基础语法后，本书最后给出经典问题的实例代码，帮助读者提升实战技能，提高编程兴趣。

编程是一门实践性极强的技能，只有不断练习和思考，才能真正掌握。希望本书能成为您 Python 学习之旅的得力助手，帮助您顺利迈入编程世界的大门。

本书由兰州财经大学的马亚丽担任主编，由叶燕文、李焱、王志强、任洁担任副主编。其中，第 1 章和第 7 章由马亚丽编写；第 2 章、第 5 章和第 8 章由王志强编写；第 3 章和第 9 章由任洁编写；第 4 章和第 6 章由叶燕文编写；第 10 章由李焱编写。全书由马亚丽策划、统稿并审定。

经过深入思考和反复讨论、修订，本书终于落地。但限于编者的能力和水平，书中难免存在不妥之处，殷切希望广大读者批评指正。

本书提供教学大纲、电子教案、教学课件、例题源代码、习题与实验参考答案、模拟试卷，读者可扫下列二维码获取。另外，书中还附有拓展阅读、Python 实例源代码，读者可扫相应章节的二维码学习。

教学大纲　电子教案　教学课件　例题源代码　习题与实验参考答案　模拟试卷

编　者

2025 年 4 月

目录 CONTENTS

第 1 章 Python 基础

己欲立而立人，己欲达而达人。

——《论语·雍也》

人与计算机打交道时，需要通过一种“语言”控制计算机，让计算机为我们做事，这种语言就叫编程语言。全世界有 6000 多种编程语言，其中流行的编程语言有 20 多种。Python 是当今非常受欢迎的编程语言之一，其以简洁优雅的语法和强大丰富的功能，为我们打开了一扇通往无限可能的大门。本章首先介绍 Python 的起源、发展、特点以及应用领域，然后讲解三种 Python 集成开发环境的安装方法以及代码的编写和运行步骤。

接下来，就让我们一同走进 Python 的基础世界，探索它如何帮助我们“立”与“达”。在学习过程中，愿我们携手共进，为彼此的编程之路增添光彩。

学习目标

(1) 了解 Python 语言的特点及应用领域。
(2) 掌握标准 Python 的安装与使用。
(3) 掌握 Pycharm 的安装与使用。
(4) 掌握 Anaconda 的安装与使用。

思维导图

1.1 Python 语言概述

Python 是一门跨平台的编程语言，最初用于编写脚本程序及科学计算。随着版本的更新，Python 多被用于数据分析、网站开发、人工智能等领域。本节将介绍 Python 的发展、特点以及应用领域。

1.1.1 Python 的发展

Python 语言的创始人是荷兰国家数学与计算机科学研究中心的吉多·范罗苏姆 (Guido van Rossum)。1989 年，吉多为了打发圣诞节假期，决定开发一个新的基于互联网社区的脚本解释程序，作为 ABC 语言的一种继承。ABC 语言是由吉多参加设计的一种教学语言，吉多认为，ABC 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，吉多认为是其非开放性造成的。吉多决心在新的语言中避免这一错误。同时，他想实现在 ABC 语言中闪现过但未曾实现的东西。

吉多特别喜爱电视喜剧 *Monty Python's Flying Circus*，所以将 Python (意为“大蟒蛇”) 作为新的编程语言的名字。于是，Python 语言诞生了。1991 年，Python 的第一个解释器发布，它由 C 语言实现，也有很多来自 ABC 语言的语法。2000 年 10 月 16 日，Python 2.0 发布，该版本增加了内存管理、对 Unicode 编码的支持等功能。2008 年 12 月 3 日，Python 3.0 版本发布，该版本比之前的 Python 2.0 版本有了较大的改进。由于 Python 3.0 版本不向下兼容，数以万计的函数库从 2.0 版本更新到 3.0 版本花费了大量的时间。

Python 经历了很多版本，具体信息如表 1.1 所示。

表 1.1 Python 版本

版本	发布时间	版本	发布时间
Python 0.9	1991.02	Python 3.3	2012.09
Python 1.0	1994.01	Python 3.4	2014.03
Python 1.6	1997.12	Python 3.5	2015.09
Python 2.0	2000.10	Python 3.6	2016.12
Python 2.3	2003.07	Python 3.7	2018.06
Python 2.4	2004.11	Python 3.8	2019.10
Python 2.5	2006.09	Python 3.9	2020.10
Python 2.6	2008.10	Python 3.10	2021.10
Python 2.7	2010.07	Python 3.11	2022.10
Python 3.0	2008.12	Python 3.12	2023.10
Python 3.1	2009.06	Python 3.13	2024.10
Python 3.2	2011.02	Python 3.14	2025.10

1.1.2 Python 的特点

Python 广泛应用于各个领域，成为全球主流的编程语言之一，其有以下主要特点。

(1) 易学易用。Python 的关键字较少，且有极其简单的说明文档，初学者容易上手。

(2) 简洁易读。Python 的语法简洁清晰，代码易读易写，采用强制缩进的方式使代码具有较好的可读性。

(3) 免费、开源。Python 是完全开放源码的，用户不但可以从 Internet 上免费获取并使用，还可以修改和分发。

(4) 可移植性。Python 可以被移植到不同操作系统，如 Windows、Linux、macOS、Android 等。在某个操作系统中编写的 Python 程序几乎可以不加修改就可以运行在其他操作系统中。

(5) 解释性。Python 是一种解释型高级语言，其程序可以直接执行。这使 Python 程序更加易于移植。

(6) 可嵌入性。用户可以将 Python 嵌入 C、C++ 程序，从而为程序提供脚本功能。

(7) 面向对象。Python 是一种面向对象的编程语言，支持类和对象的概念，方便代码的组织和重用。

(8) 功能丰富。Python 提供庞大的标准库和第三方库，包括但不限于数据处理、网络编程、图形图像处理、科学计算、机器学习、人工智能等。这些库涵盖各种领域的功能需求，利用 Python 开发时，许多功能直接使用库即可。

(9) 动态类型。Python 是动态类型语言，变量类型在运行时确定，增强了编程的灵活性。

(10) 社区支持。Python 拥有活跃的开发者社区，提供丰富的资源和支持。

1.1.3 Python 的应用领域

(1) 科学计算与数据分析。Python 提供了 NumPy(Numerical Python)、Pandas、SciPy(Scientific Python)、Matplotlib、Traits、VTK(Visualization toolkit)、OpenCV 等大量的库，这些库可以实现数值计算、符号计算、二维图表、数据可视化、图像处理以及界面设计等。

(2) Web 开发。Python 经常用于 Web 开发。Flask 和 Django 是两个非常流行的开发框架，它们可以快速构建高效、安全和可扩展的 Web 应用程序。许多大型网站如 YouTube、Google、豆瓣等都是用 Python 开发的。

(3) 网络爬虫。Python 提供了大量的网络爬虫库，如 urllib、requests、Scrapy 等。它们使 Python 在网络爬虫开发方面具有很高的效率。通过使用这些库，可以方便地抓取网页数据、进行数据分析或数据挖掘等操作。

(4) 图形界面开发。Python 也广泛应用于图形界面开发。Tkinter、wxPython、PyQt 等库使得跨平台的桌面软件开发变得简单。

(5) 人工智能与机器学习。TensorFlow、PyTorch、Keras 等深度学习框架都是基于 Python 开发的，这些框架为构建神经网络，实现自然语言处理、图像识别等功能提供了极大的便利。

(6) 操作系统管理。Python 可用于操作系统管理，如文件系统操作、进程管理、网络配置等。

(7) 嵌入式硬件编程。Python 在嵌入式硬件编程中有应用，支持硬件控制、传感器数据处理等功能。

(8) 系统运维。在很多操作系统中，Python 是标准的系统组件，可以直接在终端运行 Python 脚本。用 Python 编写的系统管理脚本在可读性、性能、代码重用度和扩展性方面都优于普通的 shell 脚本，常用于系统监控、资源管理等运维工作。

(9) 游戏开发。Python 可以用于游戏逻辑的开发，也有一些游戏开发库可供使用，可用来创建简单的 2D 或小型 3D 游戏，以及为游戏开发提供辅助工具。

(10) 云计算。在云计算领域，Python 可用于云基础设施建设和管理。一些云计算平台和服务的开发会使用 Python，而且 Python 的简洁性和灵活性使其适用于构建在云上运行的应用程序。

(11) 物联网。可以用 Python 编写脚本与传感器和设备进行交互，实现物联网应用的开发和控制，如智能家居系统、工业自动化等。

(12) 其他领域。Python 还可以应用于多媒体处理、桌面应用开发、教育领域、金融领域等。

1.2 Python 集成开发环境

在学习使用 Python 语言之前，首先要搭建好 Python 开发环境。本节将介绍三款常用 Python 集成开发环境的安装和使用。Python 集成开发环境可以安装在不同的操作系统中，本书以 Windows 为例。

1.2.1 Python 自带的集成开发环境

IDLE 是 Python 自带的开发环境，是开发 Python 程序的基本集成开发环境 (IDE)，具备基本的 IDE 功能，能够快速验证各种指令和简单程序，是非商业 Python 开发的不错选择。Python 安装好以后，IDLE 就自动安装好了。

本书以 Python 3.10 版本为基础进行代码和程序的编写与运行。

1. 下载与安装

(1) 在浏览器地址栏中输入 Python 官网地址 (https://www.python.org)，按回车键，打开 Python 官网，如图 1.1 所示。

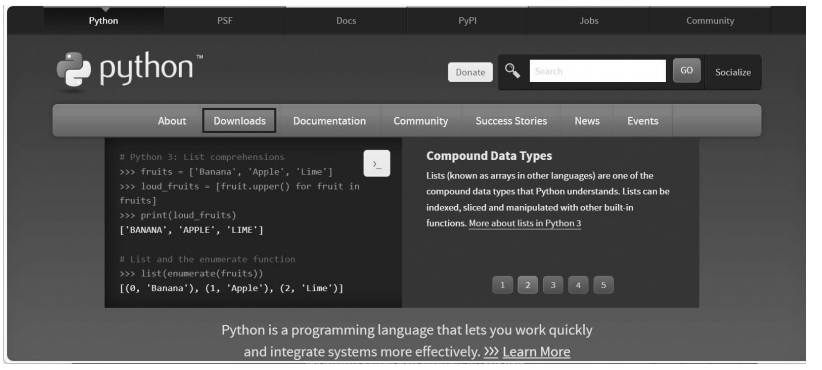

图 1.1 Python 官网主页

(2) 单击图 1.1 中的 Downloads 菜单，会显示 Python 的下载选项，如图 1.2 所示。在这里几乎可以找到 Python 的所有版本。一般情况下，网站会自动根据计算机的操作系统类型选择对应的最新的 Python 版本。

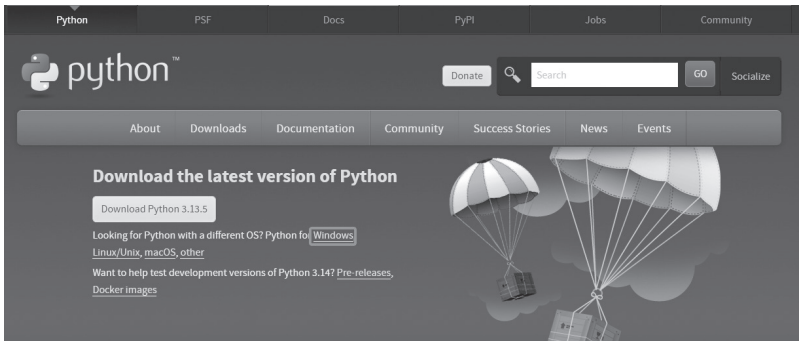

图 1.2　Python 下载页面 1

(3) 单击图 1.2 中的 Windows，会显示 Windows 平台上的所有版本，如图 1.3 所示。单击所需选项，即可下载安装程序。

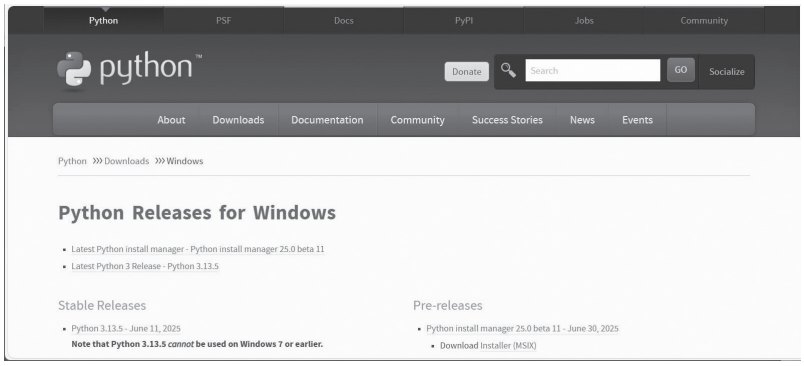

图 1.3　Python 下载页面 2

(4) 本书下载的是 64 位的 Python 3.10.11 版本，安装程序图标如图 1.4 所示。双击安装程序图标，打开安装向导对话框，选中 Add python.exe to PATH 选项，如图 1.5 所示。选中 Add python.exe to PATH 选项可以将可执行文件路径添加到 Windows 操作系统的环境变量 PATH 中，以便在后续的开发中启动各种工具。

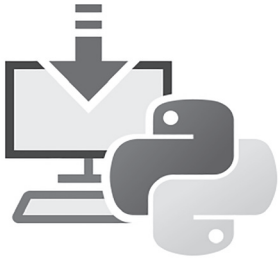

图 1.4　Python 安装程序图标

图 1.5　Python 安装向导

(5) 单击图 1.5 中的 Install Now (默认安装方式) 选项开始安装，接下来按照提示操作即可完成安装；或者使用 Customize installation (自定义安装方式) 安装，单击后进入可选特性界面，如图 1.6 所示。

(6) 单击图 1.6 中的 Next 按钮，进入高级选项界面，如图 1.7 所示。单击 Browse 按钮可选择安装路径，本书选择路径为 C:\Users\Python310。最后单击 Install 按钮开始安装，按照提示操作即可完成安装。

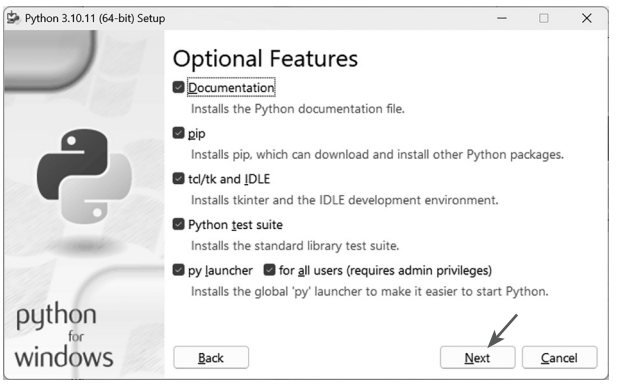

图 1.6　可选特性界面

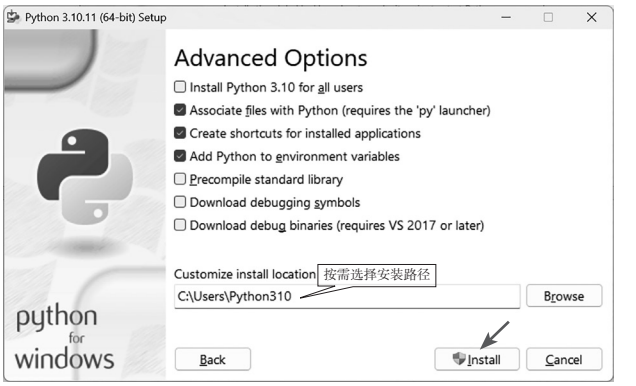

图 1.7　高级选项界面

2. 检测和使用

(1) 安装完成后，需要检测 Python 是否成功安装在 Windows 系统中。按键盘上的 Win+R 键打开运行对话框，输入 cmd，如图 1.8 所示。

(2) 单击图 1.8 中的“确定”按钮，打开命令提示符窗口，输入 Python 按回车键，显示 Python 的版本信息，并出现“>>>”提示符，说明安装成功，且此时正处于 Python 的交互模式中。

在交互模式下，可以直接输入代码，按回车键执行，立即就能看到输出结果。比如，输入 print("Hello Python!") 命令按回车键后，在下一行立刻显示“Hello Python！”，如图 1.9 所示。

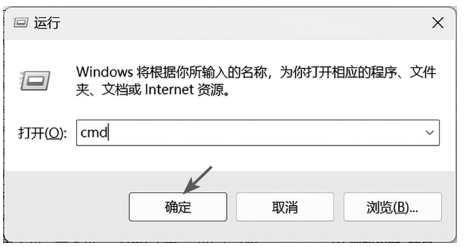

图 1.8　“运行”对话框

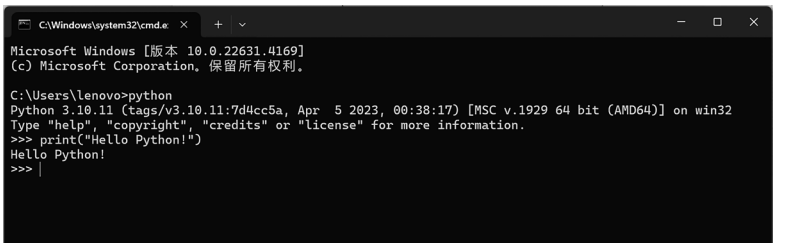

图 1.9　Python 交互模式窗口

(3) 在“开始”菜单中找到 Python 3.10，单击展开后可以看到图 1.10 所示的内容。单击 Python 3.10(64-bit) 命令可以打开图 1.9 所示的交互模式窗口；单击 IDLE(Python3.1064-bit) 命令可以打开 Python 自带的集成开发环境 IDLE 窗口，如图 1.11 所示，在该窗口中既可以将代码输入提示符“>>>”后面按回车键执行，也可以单击 File|New File 命令创建程序文件，如图 1.12 所示。

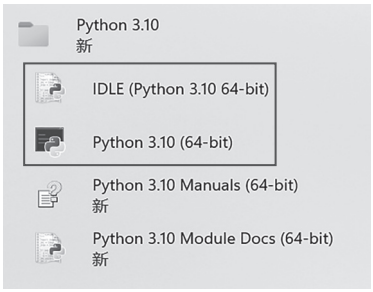

图 1.10　Python 菜单命令选项

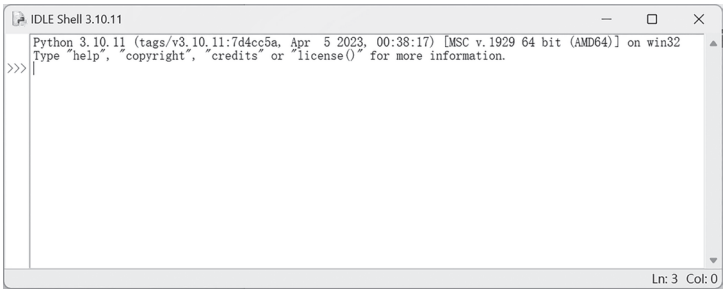

图 1.11　Python 的 IDLE 窗口

补充：集成开发环境 (integrated development environment，IDE) 是用于提供程序开发环境的应用程序，通常包括代码编辑器、编译器、调试器和图形用户界面等工具。它集成了代码编写功能、分析功能、编译功能和调试等功能，是一体化的开发软件服务套件。换言之，具备这一特性的软件或软件套 (组) 装都可以称为集成开发环境。

(4) 在图 1.12 所示的程序文件创建窗口中输入程序代码，单击 Run|Run Module 命令或直接按 F5 键执行程序，结果会输出在 IDLE 窗口中。

Python 安装成功后，在交互命令窗口中，通常会有提示符“>>>”，可以将需要执行的代码输入到提示符后面，并按回车键执行，即可在下一行看到代码的执行结果。例如：

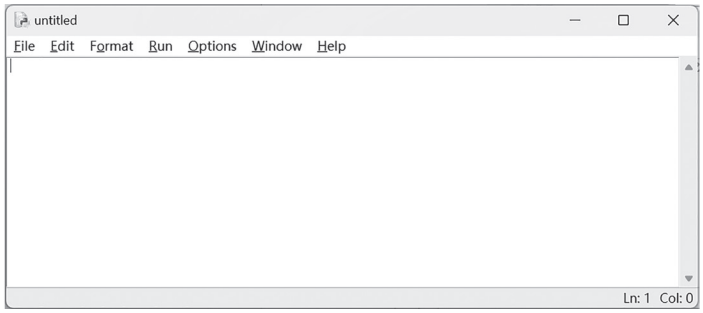

图 1.12　Python 程序文件编辑窗口

```
>>> ' 我要学 ' + 'Python'
' 我要学 Python'
```

1.2.2　PyCharm 集成开发环境

PyCharm 是由 JetBrains 打造的一款 PythonIDE，带有一整套可以帮助用户在使用 Python 语言开发时提高其效率的工具。PyCharm 具有一般 IDE 具备的功能，如调试、语法高亮、项目管理、代码跳转、智能提示、自动完成、单元测试、版本控制等。

与其他集成开发环境相比，PyCharm 有可视化的界面，输入代码、调试程序都更加方便，有利于调试程序以及大型项目的开发。

1. 下载与安装

(1) 在浏览器地址栏中输入 PyCharm 的官方下载地址，按回车键，打开 PyCharm 下载页面 (https://www.jetbrains.com/pycharm/download)。Pycharm 有付费版 Professional 和免费版 Community 两种版本，这里选择免费版，如图 1.13 所示。

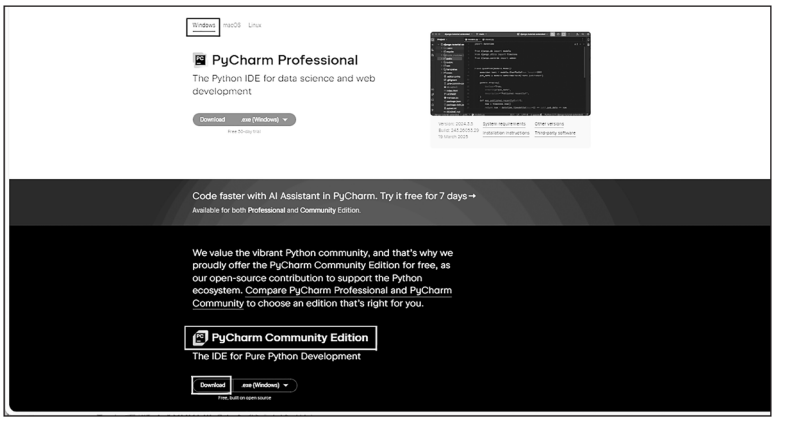

图 1.13　PyCharm 下载页面

(2) 单击 Windows，然后单击 Pycharm Community Edition 下面的 Download 进行下载，下载的文件名为 pycharm-community-2024.3.1.1.exe。双击该文件图标，进入 Pycharm 安装界面，如图 1.14 所示。

(3) 单击“下一步”按钮，进入选择安装路径界面，本书选择 D:\PyCharm，如图 1.15 所示。

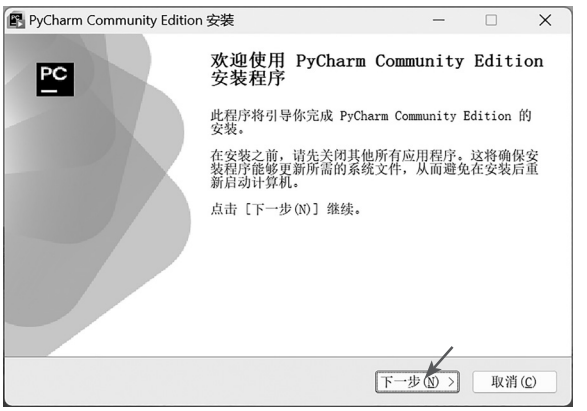

图 1.14　安装界面

图 1.15　选择安装路径界面

(4) 单击“下一步”按钮，进入配置界面，勾选全部选项，如图 1.16 所示。

(5) 单击“下一步”按钮，进入选择文件夹界面，如图 1.17 所示。

图 1.16　配置界面

图 1.17　选择文件夹界面

(6) 单击“安装”按钮，进入安装过程界面，如图 1.18 所示。

(7) 安装完成后，界面如图 1.19 所示，选择“否，我会在之后重新启动”，最后单击“完成”按钮。

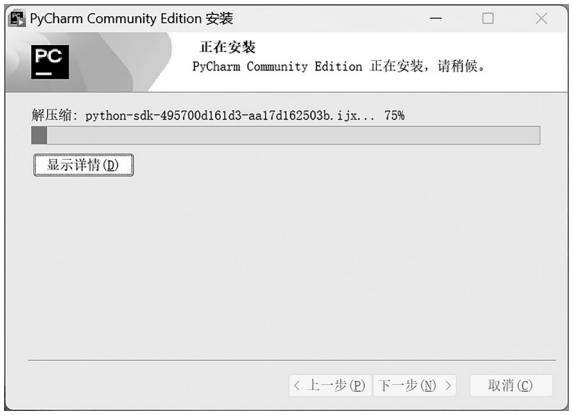

图 1.18　安装过程界面

图 1.19　安装完成界面

2. 配置与使用

(1) 启动 PyCharm，打开 Welcome to Pycharm 界面，如图 1.20 所示。

(2) 单击 New Project，修改 Location(项目目录路径)，指定名称，本书指定为 D:\PythonProject；选择解释器，本书选择 Python 3.10，如图 1.21 所示。

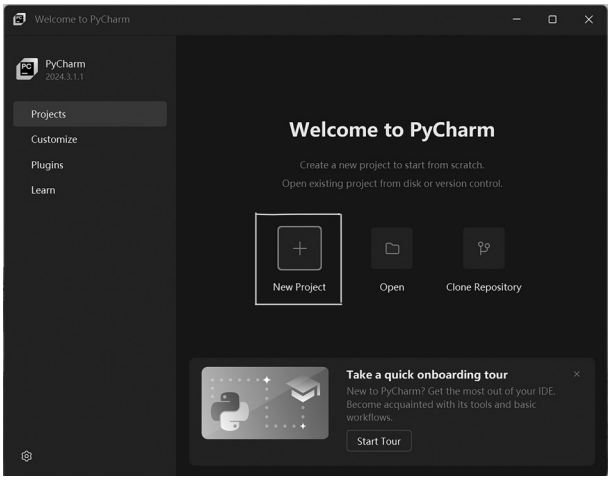

图 1.20　欢迎界面

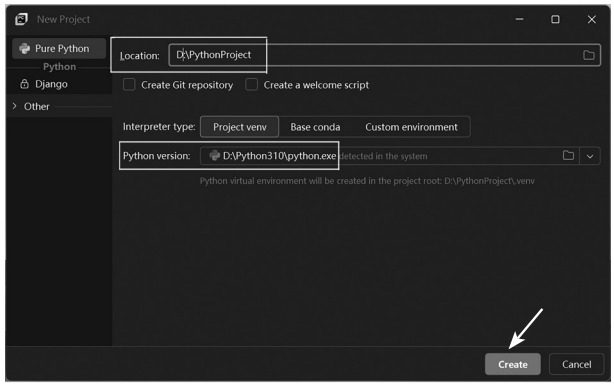

图 1.21　修改 Location

(3) 单击 Create 按钮，打开 PyCharm 工作界面，如图 1.22 所示。

(4) 单击图 1.22 中箭头所指的 Main Menu 按钮，打开 PyCharm 系统主菜单栏，如图 1.23 所示。

(5) 单击主菜单栏中的 File|New Project 可以创建新项目。单击 File|Settings 可以对 PyCharm 进行设置 (本书不再详述)。

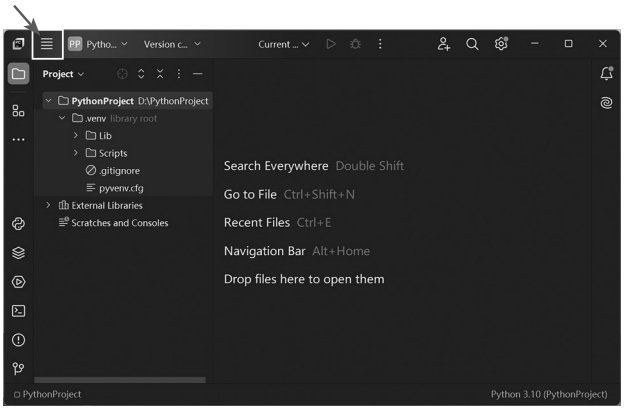

图 1.22　工作界面

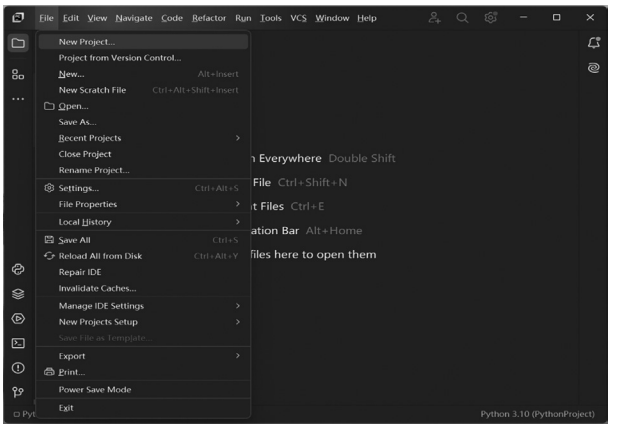

图 1.23　主菜单栏

1.2.3　Anaconda 集成开发环境

Anaconda 是一个开源的 Python 发行版本，可以看作 Python 的包管理工具，类似于 pip。其中包含了 Conda、Python 等 180 多个科学包及其依赖项。由于包含的科学包数量较多，Anaconda 所占存储空间较大。第 9 章数据分析所需的 Python 库都包含在 Anaconda 中。

Anaconda 中包含的 Jupyter Notebook 是基于网页用于交互计算的应用程序，可被应用于全过程计算，包括开发、文档编写、运行代码和展示结果。

1. 下载与安装

(1) 在浏览器地址栏中输入 Anaconda 下载地址 (https://www.anaconda.com) 或清华大学开源软件镜像站 (https://mirrors. tuna.tsinghua.edu.cn/anaconda/archive/)，按回车键，如图 1.24 所示。

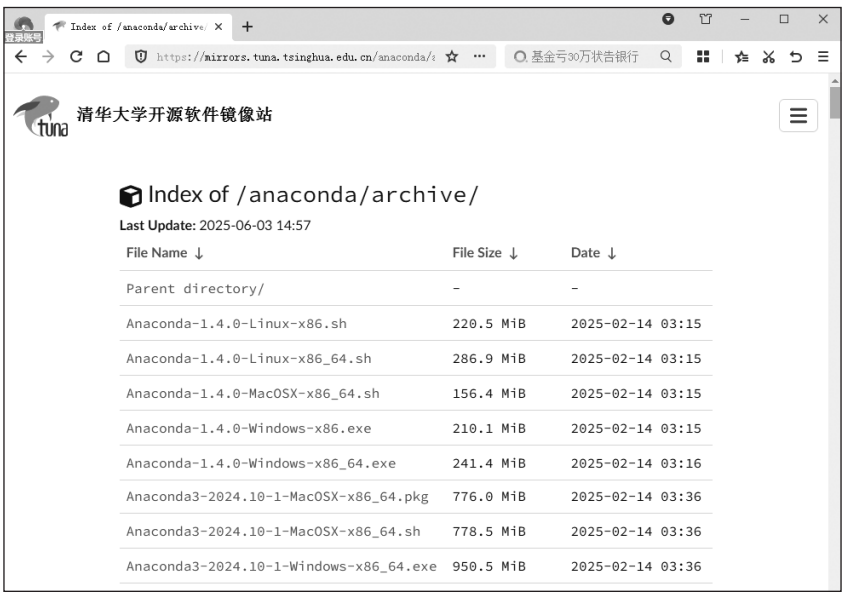

图 1.24　Anaconda 下载页面

(2) 根据所用计算机的配置情况，选择相应版本下载，本书下载的是 Anaconda3-2024.10-1-Windows-x86_64.exe。下载完成后，双击安装文件，打开安装窗口，如图 1.25 所示。

(3) 单击图 1.25 中的 Next 按钮，进入图 1.26 所示窗口。

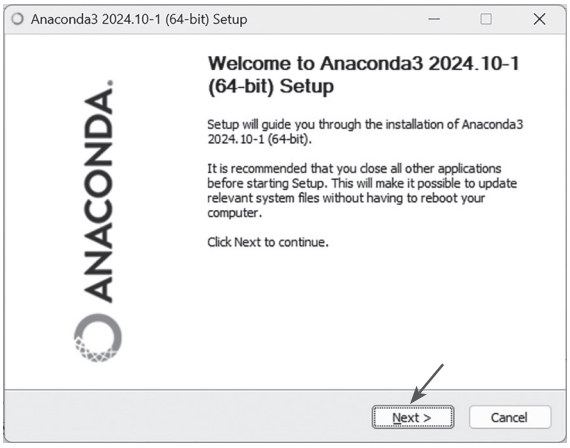

图 1.25　Anaconda 安装窗口 1

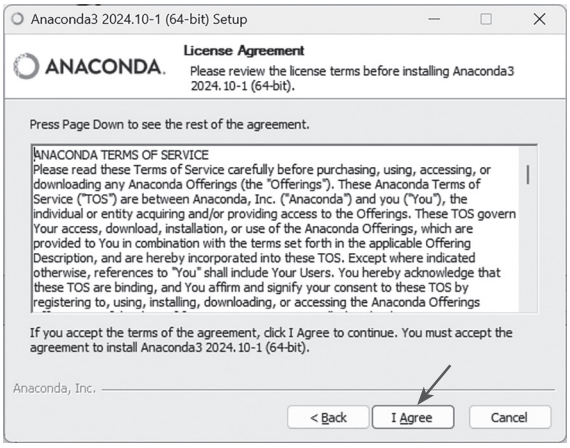

图 1.26　Anaconda 安装窗口 2

(4) 单击图 1.26 中的 I Agree 按钮，进入图 1.27 所示窗口。

(5) 选中图 1.27 中的 All Users (requires admin privileges) 选项，单击 Next 按钮，弹出图 1.28 所示窗口。

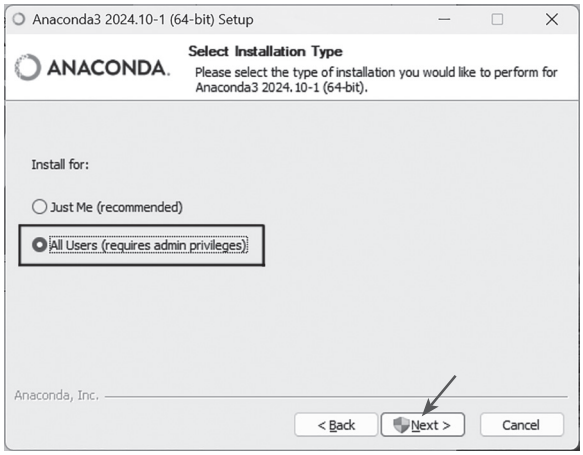

图 1.27　Anaconda 安装窗口 3

图 1.28　用户账户控制

(6) 单击图 1.28 中的“是”按钮，弹出图 1.29 所示窗口。

(7) 在图 1.29 中选择安装位置，本书选择 D:\anaconda，然后单击 Next 按钮，弹出图 1.30 所示窗口。

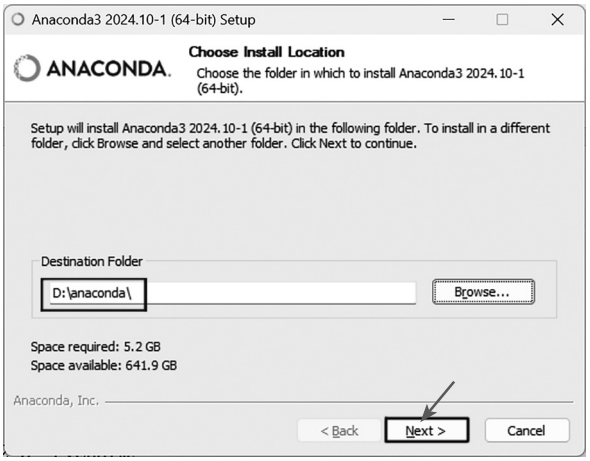

图 1.29　Anaconda 安装窗口 4

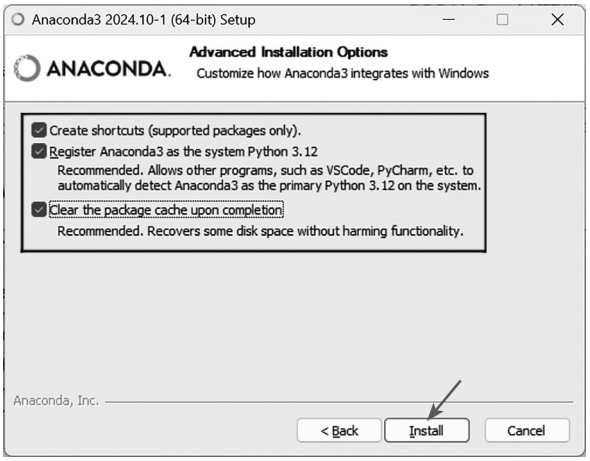

图 1.30　Anaconda 安装窗口 5

(8) 选中图 1.30 中的三个选项，单击 Install 按钮，等待安装，随后弹出图 1.31 所示的窗口。

(9) 单击图 1.31 中的 Next 按钮，弹出图 1.32 所示的窗口。

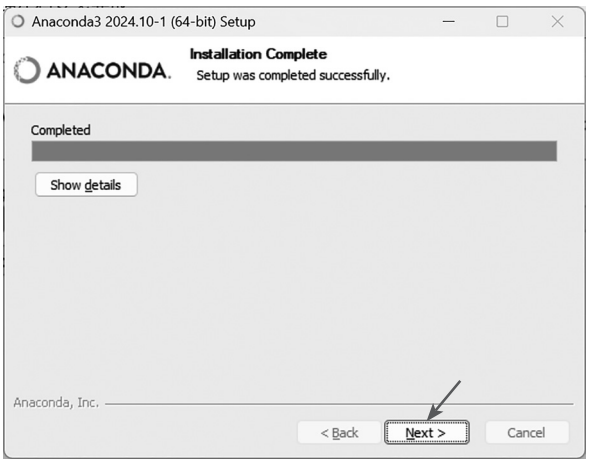

图 1.31　Anaconda 安装窗口 6

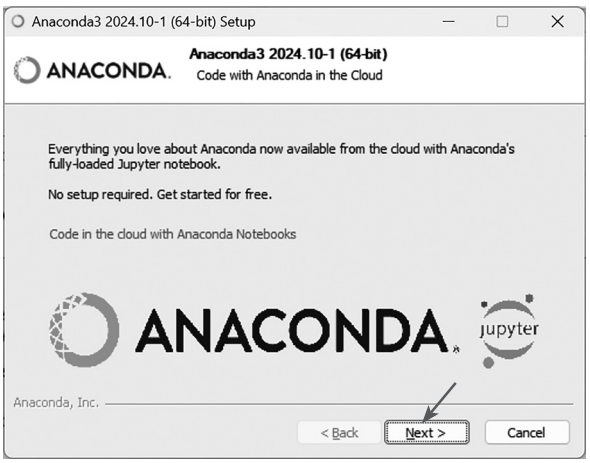

图 1.32　Anaconda 安装窗口 7

(10) 单击图 1.32 中的 Next 按钮，弹出图 1.33 所示的窗口。

(11) 选中 Launch Anaconda Navigator 选项，单击 Finish 按钮，安装完成。

2. 配置环境变量

安装 Anaconda 后，系统默认并不会自动识别 Conda 命令，这意味着无法直接在命令行中输入 Conda 来使用 Anaconda 的相关功能。通过配置环境变量，告诉操作系统 Conda 命令的存储位置，可以使系统能够找到并执行这个命令。

(1) 在运行窗口 (按 Win+R 打开) 中输入 sysdm.cpl(或者在搜索框中输入“查看高级系统设置”)，单击“确定”按钮，打开系统属性对话框，切换至高级选项卡，如图 1.34 所示。

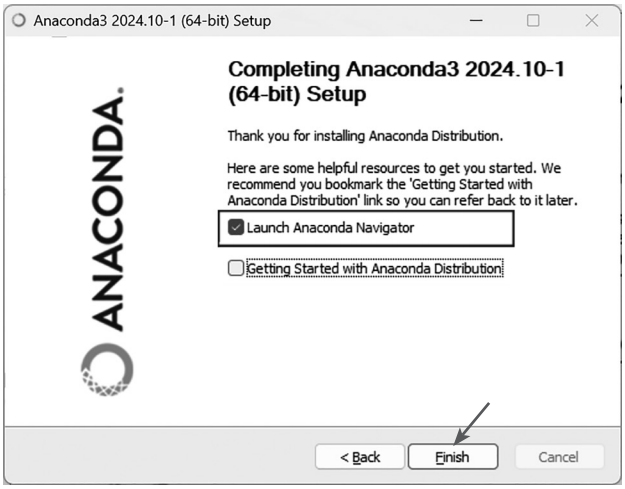

图 1.33　Anaconda 安装窗口 8

图 1.34　“系统属性”对话框

(2) 单击“环境变量”按钮，打开“环境变量”对话框，如图 1.35 所示。

(3) 双击“系统变量”中的 Path，打开“编辑环境变量”对话框，并依次新建 4 条路径，如图 1.36 所示。

(4) 单击“确定”按钮，直至退出“环境变量”对话框。

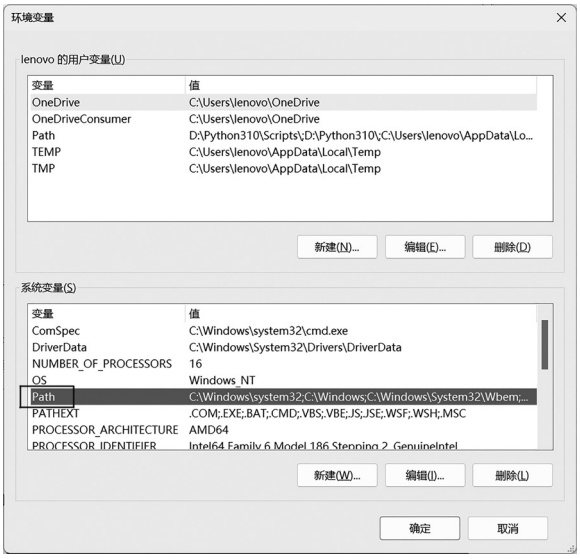

图 1.35　“环境变量”对话框

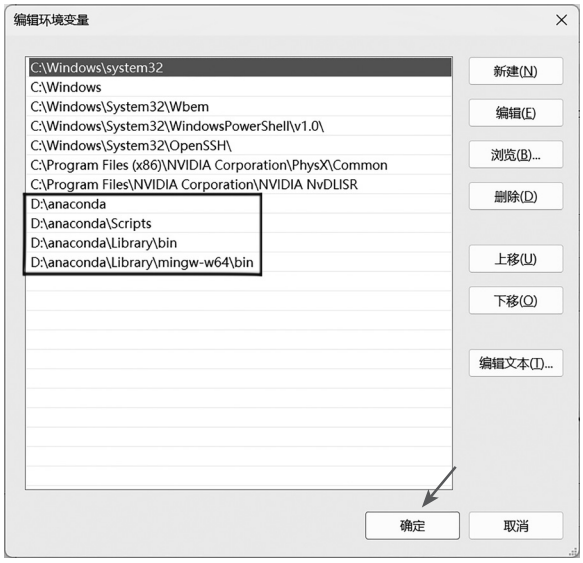

图 1.36　“编辑环境变量”对话框

3. 检测与使用

(1) 打开命令提示符窗口，输入命令 conda --version，按回车键后若显示出 Anaconda 的版本，则说明安装成功，如图 1.37 所示。

(2) 单击“开始”菜单中的 Anaconda，可以看到图 1.38 所示的内容，根据需要单击相关命令。

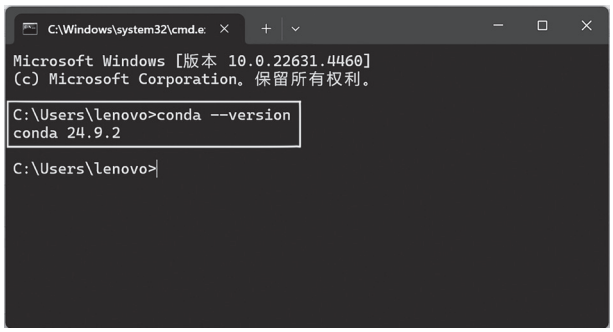

图 1.37　测试 Anaconda 版本

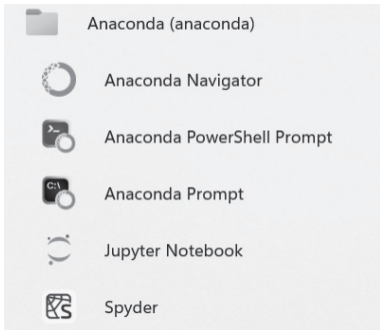

图 1.38　Anaconda 菜单命令

【例 1-1】输入一个十进制整数，输出其对应的二进制数。示例代码如下：

```
1    x = int(input(" 请输入一个整数："))                    # 输入整数
2    b = bin(x)                                            # 利用 bin() 函数将十进制转换为二进制
3    print(x," 对应的二进制数是 ", b)
```

运行结果如下：

```
请输入一个整数：93
93 对应的二进制数是 0b1011101
```

【例 1-2】判断输入的年份是否为闰年。示例代码如下：

要求：在某一集成开发环境中输入以下代码，运行并查看结果。

```
1    year = int(input(" 请输入年份："))                     # 输入年份
2    if(year%400 == 0) or (year%4 == 0 and year%100!= 0):   # 判断是否为闰年
3        print("%d 年是闰年 "%year)
4    else
5        print("%d 年不是闰年 "%year)
```

第 2 行代码中的表达式用来判断输入的年份 year 是否为闰年。

运行结果如下：

```
请输入年份：2025
2025 年不是闰年
```

1.3　Python 语言编码总规范

一个好的 Python 代码不仅应该是正确的，还应该是漂亮的、优雅的，具有非常强的可读性和可维护性，读起来令人赏心悦目。遵循一套统一的编码规范是实现这一目标的基础，它能让代码更易于理解和维护。本节将介绍 Python 语言编码规范中的几个核心原则，详细规范在 2.1 节中展开讲述。

(1) 缩进。在 Python 中，严格使用缩进来体现代码的逻辑从属关系。错误的缩进会导致代码无法运行或者可以运行但结果错误。此外，相同级别的代码块应具备相同的缩进量。

(2) 区分大小写字母。Python 对英文字母的大小写是敏感的。比如，字母 A 和 a 在程序中代表不同含义。

(3) 英文半角字符。在编写 Python 代码时，所有定界符和分隔符都应使用英文半角字符，如元素之间的逗号、列表的方括号、字符串的引号、字典的键和值之间的冒号、元组的圆括号等。

(4) 注释。对关键代码或重要代码添加必要的注释，可方便代码的阅读和维护。Python中有两种注释方式：一种是以“#”开头的单行注释；另一种是放在两个三引号之间的多行注释。

(5) 命名规范。给变量、函数、类等程序元素赋予清晰、一致且有意义的名称是提高代码可理解性的关键。Python对此有推荐的命名约定。

(6) 续行。尽量不写过长的语句，应尽量保证一行代码不超过屏幕宽度。如果语句太长而超过屏幕宽度，最好在行尾使用续行符“\”表示下一行代码仍属于本条语句，或者使用圆括号把多行代码括起来。

1.4 习题与实验

一、填空题

1. Python是_____型编程语言。
2. Python程序文件的扩展名是_____。
3. Python自带的集成开发环境的缩写是_____。
4. Python的单行注释语句是_____。

二、选择题

1. Python可以在Windows、Mac、Linux等不同操作系统上运行，体现了Python的(　　)特性。
   A. 可扩展　　B. 可移植
   C. 面向对象　　D. 简单
2. Python语言属于(　　)语言。
   A. 机器语言　　B. 汇编语言
   C. 高级语言　　D. 以上都不是
3. 下列不属于Python集成开发环境的是(　　)。
   A. Python　　B. PyCharm
   C. Jupyter Notebook　　D. IDLE
4. (　　)是基于网页的用于交互计算的集成开发环境。
   A. Python　　B. PyCharm
   C. Jupyter Notebook　　D. IDLE
5. 下列不属于Python语言特征的是(　　)。
   A. 简单易学　　B. 免费开源
   C. 面向对象　　D. 编译性

三、简答题

1. 如何选择合适的Python版本?
2. Python语言有哪些特点?
3. 请简述Python语言的编码总规范。

四、实验题

1. 下载并安装 Python。

2. 在 IDLE 交互式窗口中输入运行以下代码，并查看输出结果。

```
'Python' * 2
```

3. 创建一个 Python 程序文件，输入以下代码，并运行查看输出结果。

```
a = int(input(" 请输入一个整数："))
if (a % 2 == 0)：
        print(a , ' 是偶数 ')
else：
        print(a , ' 是奇数 ')
```

第 2 章 编程基础

易简而天下之理得矣。

——《周易·系辞上》

《周易·系辞上》有言：易简而天下之理得矣。编程之道，亦如是。看似纷繁的代码世界，实则由最基础的数据类型与表达构建，如同阴阳二爻衍生万象。本章将带您进入 Python 编程的基础世界，在这里，逻辑和代码是核心。我们将系统学习 Python 的基本语法，包括注释、标识符、关键字和输入输出；掌握常量、变量以及核心数据类型 (数字、字符串、布尔值)；熟悉各类运算符及其在表达式中的应用；并了解如 range() 等常用内置函数、模块与包的概念以及通过 sys 和 os 模块与系统交互的基础。掌握这些内容是后续深入学习 Python 的关键。

学习目标

(1) 了解 Python 的基本语法规则。
(2) 掌握 Python 中输入与输出的基本操作方法。
(3) 理解常量与变量的概念及其在 Python 中的使用方法。
(4) 掌握 Python 中基本数据类型的使用。
(5) 熟悉各种运算符及表达式的使用方法。
(6) 了解运算符的优先级及其在表达式中的应用。
(7) 掌握常用的内置函数的使用方法。
(8) 了解模块与包的基本概念及其在 Python 中的应用。
(9) 掌握标准模块 sys 和 os 的基本使用方法。

思维导图

2.1 基本语法

本节介绍构成 Python 程序骨架的基本语法规则。掌握这些规则是编写正确、规范 Python 代码的第一步，也是理解后续更复杂概念的前提。本节内容涵盖代码风格基础、注释的使用、标识符命名规则和规范、Python 关键字以及基本的输入与输出操作。

2.1.1 代码风格基础

编写 Python 程序不仅要让计算机理解并执行，更重要的是让人能够轻松阅读、理解和维护。清晰、一致的代码风格是高效编程和团队协作的基础。Python 社区以 PEP 8(Python Enhancement Proposal 8) 作为官方的风格指南。虽然 Python 解释器只强制执行其中一部分规则 (如缩进)，但遵循 PEP 8 的建议是编写高质量 Python 代码的标志。本节将介绍几个最基本的代码风格要素。

1. 缩进

与许多使用花括号 {} 来定义代码块 (如函数体、循环体、条件语句块) 的编程语言不同，Python 使用缩进来划分代码块。这意味着正确的缩进是 Python 语法的一部分，错误的缩进会导致程序无法运行。

PEP 8 强烈推荐使用 4 个空格作为每一级缩进的标准。请务必在整个项目中保持一致，不要混用空格和制表符 (Tab)。示例代码如下：

```
# 使用 4 个空格进行缩进
score = 75
if score >= 60：
    print(" 祝贺你 !")                    #if 语句块内的代码缩进 4 个空格
    print(" 你通过了考核。")               # 同上
else：
    print(" 请继续努力 !")                #else 语句块内的代码缩进 4 个空格
print(" 考核结束。")                      # 此行代码与 if/else 同级，不缩进
```

在上面的示例中，if 和 else 下面的 print 语句通过缩进明确了它们所属的代码块。

2. 空格

虽然空格通常不影响程序的运行，但合理使用空格可以极大地提升代码的可读性。PEP 8 建议在以下地方使用空格。

(1) 在二元运算符 (如 =、+=、==、<、>、+、–、*、/ 等) 的两侧各加一个空格，如 x = y + 1 比 x=y+1 更清晰。

(2) 在逗号后面加一个空格，如 print(a, b) 比 print(a,b) 更易读。

> **注意：**在函数调用“()”或索引“[ ]”的内侧通常不加空格。

3. 行长度

过长的代码行难以阅读，并且可能需要在编辑器中水平滚动。PEP 8 建议每行代码的长度不应超过

79 个字符。如果代码行需要超过这个长度，可以使用以下两种方式换行。

(1) 隐式换行。在括号 ()、方括号 [ ] 或花括号 {} 内的表达式，可以自然地跨越多行。这是推荐的方式。

(2) 显式换行。显式换行是使用反斜杠 (\) 作为续行符的一种方式，但在现代 Python 中应尽量避免。

4. 注释与命名

仅仅遵循格式规范还不够，要写出真正易于理解的代码，还需要注意以下两点。

(1) 注释 (comments)。注释用来解释代码的目的、逻辑或复杂部分。我们将在 2.1.2 节对注释进行详细讨论。

(2) 有意义的命名 (identifiers)。有意义的命名是指为变量、函数、类等选择描述性强的名称。具体的命名规则和规范将在 2.1.3 节介绍。

2.1.2　注释

在编写 Python 程序时，注释是非常重要的组成部分。注释可以帮助程序员理解代码的功能和逻辑，尤其是在代码复杂或需要长期维护的情况下。注释不会被 Python 解释器执行，因此不会影响程序的运行。注释主要分为单行注释和多行注释两种形式。

1. 单行注释

单行注释是指在一行代码中添加的注释，通常用于对某一行或某一段代码进行简短的说明。单行注释以井号 (#) 开头，井号后面的内容即为注释内容。单行注释可以放在代码行的末尾，也可以单独占一行。示例代码如下：

```
# 这是一个单行注释，用于说明下面代码的功能
print("Hello, World!")                    # 输出 Hello, World! 到控制台
```

2. 多行注释

多行注释用于对较长的代码段或复杂的逻辑进行详细说明。多行注释通常使用三个单引号 (''') 或三个双引号 (""") 包围注释内容，可以跨越多行。多行注释适用于对函数、类或模块进行详细的描述，或者在代码中添加较长的解释性文字。示例代码如下：

```
'''
这是一个多行注释的示例。
多行注释可以跨越多行，用于对代码进行详细说明。
以下代码定义了一个函数，用于计算两个数的和。
'''
def add(a, b):
    """
    这是一个多行注释，通常用于函数或类的文档字符串。
    该函数接收两个参数 a 和 b，返回它们的和。
    """
    return a + b
# 调用 add 函数，计算 3 和 5 的和
result = add(3, 5)
print(result)
```

在上述代码中，第 1~6 行多行注释使用三个单引号包围，用于对整个代码段进行详细说明。第 7~14 行多行注释使用三个双引号包围，作为函数的文档字符串 (docstring)，用于描述函数的功能、参

数和返回值。文档字符串是 Python 中一种特殊的多行注释，通常用于生成自动化文档。合理使用多行注释，可以为代码提供详细的说明和解释，帮助程序员更好地理解和维护代码。

2.1.3 标识符

在 Python 编程中，标识符用于标识变量、函数、类、模块和其他对象的名称。标识符是程序中最基本的组成部分，它为程序中的数据和功能提供了唯一的标识，使得程序能够正确地引用和操作这些数据和功能。

1. 标识符的命名规则

在 Python 编程中，标识符的命名规则是确保代码规范性和可读性的基础。遵循这些规则可以避免命名冲突和语法错误，使代码更加清晰和易于维护。以下是具体的 Python 标识符命名规则。

(1) 由字母、数字和下划线组成。标识符可以包含字母、数字和下划线，但不能包含空格、特殊字符或标点符号。

(2) 字母和下划线开头。标识符必须以字母 (大写或小写) 或下划线 (_) 开头，不能以数字开头。这是为了确保标识符与数字常量区分开来。

(3) 区分大小写。Python 标识符要区分大小写，这意味着 Variable 和 variable 是两个不同的标识符。

(4) 避免使用保留字。Python 有一些保留字 (关键字)，它们具有特殊的含义，不能用作标识符。这些保留字包括 if、else、for、while、def 等。

示例代码如下：

```
# 正确的标识符
my_variable = 10
_variable = 20
# 错误的标识符 ( 包含空格和特殊字符 )
#my variable = 30                    # 这行代码会导致语法错误
#variable@= 40                       # 这行代码会导致语法错误
# 错误的标识符 ( 以数字开头 )
#1variable = 30                      # 这行代码会导致语法错误
#Variable 和 variable 是两个不同的标识符
Variable = 10
variable = 20
print(Variable)                      # 输出结果为 10
print(variable)                      # 输出结果为 20
# 错误的标识符 ( 使用保留字 )
#def = 10                            # 这行代码会导致语法错误
```

2. 标识符的命名规范

除了上述必须遵守的硬性规则外，Python 社区还推崇一套被广泛接受的命名规范 (主要源自 PEP 8 指南)，遵循这些规范能够显著提高代码的可读性、一致性和可维护性，是养成良好编程习惯的重要一步。关键的命名规范如下。

(1) 变量和函数名。变量和函数名推荐使用小写字母，并通过下划线来分隔单词，这被称为蛇形命名法 (snake_case)。名称应当清晰、准确地反映变量的用途或函数的功能。示例代码如下：

```
user_name = "Alice"
item_count = 0
def calculate_discount(price，percentage)：
    # 函数体实现折扣计算逻辑
    pass
```

(2) 类名。类名推荐采用首字母大写的单词连接形式，即驼峰命名法，单词之间不使用下划线。示例代码如下：

```
class ShoppingCart：
    # 类定义内容
    pass
class UserProfile：
    # 类定义内容
    pass
```

(3) 常量名。对于意在表示程序运行期间不应改变的值 (常量)，约定使用全部大写字母，单词之间用下划线分隔。示例代码如下：

```
PI = 3.14159
MAX_LOGIN_ATTEMPTS = 5
DEFAULT_TIMEOUT = 30
```

(4) 模块与包名。模块与包名通常建议使用小写字母，单词之间不使用下划线分隔。更多关于模块与包的内容将在 2.6 节进行详细讨论。

2.1.4　关键字

在 Python 编程语言中，关键字 (keywords) 是具有特殊含义的保留字，用于定义语言的语法结构和逻辑操作。关键字是 Python 解释器内置的，不能用作标识符 (如变量名、函数名等)，因为它们是 Python 语法的一部分。理解并正确使用关键字是编写 Python 程序的基础之一。

从概念和历史演变的角度来看，Python 的关键字可以分为硬关键字和软关键字两类。

(1) 硬关键字 (hard keywords)。硬关键字是在任何上下文中都严格保留的词，绝不能用作标识符，如 if，else，while，def，class 等。

(2) 软关键字 (soft keywords)。这些词只有在特定的语法结构中才具有关键字的含义。例如，Python 3.10 中引入的软关键字 match，case 和下划线 (_)，它们仅在 match...case... 模式匹配语句中作为关键字起作用。关于详细的模式匹配将会在第 3 章“程序控制结构”中进行解释。

可以通过 keyword 模块查看当前 Python 版本的硬关键字和软关键字列表。以下是获取两种关键字列表的示例代码：

```
# 导入 keyword 模块
import keyword
# 获取当前 Python 版本的所有硬关键字
hardKeywords = keyword.kwlist
# 打印关键字列表
print("Python 的硬关键字列表：")
for kw in hardKeywords：
    print(kw, end=", ")
# 获取当前 Python 版本的所有软关键字
softKeywords=keyword.softkwlist
print("\nPython 的软关键字列表：")
for kw in softKeywords：
    print(kw, end=", ")
```

以下是 Python 3.10 版本中所有硬关键字的完整列表 (共 35 个)：

```
False, None, True, and, as, assert, async, await, 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
```

以下是 Python 3.10 版本中所有软关键字的完整列表 (共 3 个)：

```
_, case, match
```

为了保持代码的规范性和避免语法错误，使用关键字时需要注意以下几点。

(1) 硬关键字不可用作标识符。硬关键字是 Python 语言中保留的具有特殊含义的词汇，试图将其用作变量名、函数名或其他标识符会导致语法错误。

(2) 软关键字同样应避免用作标识符。为了保持与现有代码 (Python 3.10 以前的代码) 的兼容性，软关键字可以作为标识符使用，且不会像硬关键字那样直接导致语法错误，但这是极其不推荐的做法，这会严重破坏代码的可读性并可能导致混淆。

(3) 避免与关键字名称相似。在定义标识符时，应尽量避免使用与关键字名称相似的名称，以免引起混淆或误解。

(4) 保持对版本变化的敏感性。Python 的关键字列表会随着版本的更新而变化。

这里需要再次强调，无论是硬关键字还是软关键字，都绝对不要将它们用作标识符 (变量名、函数名等)。始终关注 Python 官方文档，了解最新版本的关键字列表和规则变化，是确保代码健壮性和兼容性的良好习惯。

2.1.5 输入与输出

1. 输入函数 input()

在 Python 编程中，input( ) 函数用于从键盘获取输入的数据。该函数会暂停程序的执行，等待用户输入数据，并在用户按下回车键后继续执行程序。input( ) 函数返回一个字符串类型的值，即用户输入的内容。理解和使用 input( ) 函数是实现交互式程序的基础。

input( ) 函数基本语法格式如下：

```
variable = input([prompt_string])
```

调用这个函数会启动用户输入过程。prompt_string(可选参数) 是一个字符串，会在用户输入数据之前显示在屏幕上，作为给用户的提示信息。如果省略这个参数，则不会显示任何提示。通常，会使用赋值操作符 (=) 将这个返回的字符串存储在一个变量 (如以上语法格式中的 variable) 中，以便后续在程序中使用。

input( ) 函数的基本用法示例如下：

```
1    # 使用 input( ) 函数获取用户输入
2    user_input = input(" 请输入你的名字：")
3    # 输出用户输入的内容
4    print(" 你好 , " + user_input)
```

运行结果如下：

```
请输入你的名字：小明
你好 , 小明
```

在上述代码中，input( ) 函数会显示提示信息“请输入你的名字：”，等待用户输入。当用户输入内容并按下回车键后，输入的内容会被赋值给变量 user_input。

需要注意的是，input( ) 函数返回的始终是字符串类型。如果需要其他类型的数据，可以使用相应的类型转换函数对 input( ) 函数的返回值进行转换。例如，将用户输入的内容转换为整数或浮点数。示例代码如下：

```
# 使用 input() 函数获取用户输入，并转换为整数类型
user_age = int(input(" 请输入你的年龄："))
# 输出用户输入的年龄
print(" 你的年龄是：" + str(user_age))
```

运行结果如下：

```
请输入你的年龄：18
你的年龄是：18
```

在上述代码中，input() 函数获取用户输入的年龄，并使用 int() 函数将其转换为整数类型。随后，程序使用 print() 函数输出用户输入的年龄。需要注意的是，在输出时，整数类型的变量 user_age 需要使用 str() 函数转换为字符串类型，以便与其他字符串拼接。

input() 函数的使用应遵循以下原则。

(1) 提供明确的提示信息。在调用 input() 函数时，应提供明确的提示信息，告知用户需要输入的内容。

(2) 处理用户输入的异常情况。在处理用户输入时，应考虑到可能的异常情况，如用户输入非预期的数据类型。可以使用异常处理机制 (如 try-except 语句) 来捕获和处理这些异常。关于异常处理将会在第 7 章“异常处理”中详细解释。

2. 输出函数 print()

在 Python 编程中，print() 函数用于将指定的内容 (如文本、变量的值、表达式的结果等) 输出到标准输出设备，通常是控制台 (终端或命令提示符)。print() 是常用的输出函数之一，是程序向用户展示信息、反馈状态或调试代码的关键工具。

print() 函数基本语法格式如下：

```
print(object(s)[, sep=' ', end='\n', file=sys.stdout, flush=False])
```

上述语法格式中，print() 是函数的名称。参数列表中的参数含义解释如下。

(1) object(s)(零个或多个参数) 是要输出的一个或多个表达式，可以传递任何类型的 Python 对象 (字符串、数字、列表、变量等)。如果传递多个对象，它们之间默认用空格分隔。如果省略所有对象，print() 会默认输出一个空行。

(2) 当输出多个对象时，sep=' '(关键字参数，可选) 指定它们之间的分隔符。默认值是一个空格 (' ')。可以将其设置为空字符串 (") 或其他任何字符串 (如 ', ', '---' 等)。

(3) 在所有对象输出完毕后，end='\n'(关键字参数，可选) 指定追加在末尾的字符串。默认值是换行符 ('\n')，这就是为什么每次调用 print() 后，后续的输出通常会出现在新的一行。可以将其设置为空字符串 (") 来阻止换行，或设置为其他字符串。

(4) file=sys.stdout(关键字参数，可选) 指定输出的目标。默认是标准输出 sys.stdout(控制台)。可以将其设置为一个打开的文件对象，从而将内容输出到文件中。

(5) flush=False(关键字参数，可选) 控制输出是否立即“刷新”到目标设备。通常不需要修改此参数，除非在需要确保输出立即显示的特定场景下 (如实时日志)。

对于基础用法，通常只需要关注 object(s)、sep 和 end 三个参数。

print() 函数的基本用法示例代码如下：

```
# 使用 sep 参数指定分隔符
print("Python", "Java", "C++", sep=", ")
```

```
# 使用 end 参数指定输出结束时的字符
print("Hello", end=" ")
print("World!")
```

运行结果如下：

```
Python, Java, C++
Hello World!
```

上述代码中，sep 参数指定分隔符为逗号和空格，输出结果为“Python, Java, C++”。end 参数指定输出结束时的字符为空格，两个 print( ) 函数的输出结果在同一行显示，输出结果为“Hello World!”。

print( ) 函数还支持格式化输出，可以使用格式化字符串 (f-string) 或 format( ) 方法进行格式化输出，示例代码如下：

```
name = "Alice"
age = 20
print(f" 姓名：{name}, 年龄：{age}")
# 使用 format( ) 方法进行格式化输出
print(" 姓名：{}, 年龄：{}".format(name, age))
```

运行结果如下：

```
姓名：Alice, 年龄：20
姓名：Alice, 年龄：20
```

【拓展阅读 2-1】Python 字符串格式化方法详解

2.2 常量与变量

本节将详细介绍 Python 编程中常量的定义与分类、使用规则，以及变量的定义、命名规则、类型与动态赋值。掌握这些知识，有助于读者在编写程序时有效地管理和操作数据，为后续理解更复杂的编程概念打下坚实的基础。

2.2.1 常量

在 Python 编程中，常量通常用于表示那些在整个程序运行期间保持不变的值，如数学常数、配置信息等。需要注意的是，Python 中没有专门的“常量类型”，一般通过使用命名规范 (如使用全大写变量名) 来表示常量。

常量定义示例代码如下：

```
PI = 3.14159                                    # 定义一个常量，用于表示圆周率
GRAVITY = 9.81                                  # 定义一个常量，用于表示重力加速度
# 使用常量计算圆的面积
radius = 5
area = PI * (radius ** 2)
print(f" 半径为 {radius} 的圆的面积是：{area}")
# 使用常量计算物体的重量
mass = 10
weight = mass * GRAVITY
print(f" 质量为 {mass}kg 的物体在地球上的重量是：{weight}N")
```

运行结果如下：

```
半径为 5 的圆的面积是：78.53975
质量为 10kg 的物体在地球上的重量是：98.10000000000001N
```

在上述代码中，PI 和 GRAVITY 是常量，它们分别表示圆周率和重力加速度。程序使用常量 PI 计算圆的面积，并使用常量 GRAVITY 计算物体的重量。使用这些常量，可以提高代码的可读性和可维护性。

除了数值类型，常量也可以是其他任何 Python 数据类型。下面是一个更综合的示例代码，展示了在程序配置中使用不同类型的常量：

```
# 程序配置常量
MAX_RETRIES = 3                                    # 整型常量：最大重试次数
DEFAULT_TIMEOUT = 15.0                             # 浮点型常量：默认超时时间 ( 秒 )
API_ENDPOINT = "https://api.example.com/v1"        # 字符串常量：API 接口地址
ENABLE_DEBUG_MODE = False                          # 布尔型常量：是否启用调试模式
DEFAULT_HEADERS = None                             # 特殊常量 None：默认请求头 ( 可能稍后填充 )
# 模拟使用配置
print(f " 配置 – 最大重试次数：{MAX_RETRIES}")
print(f " 配置 – API 地址：{API_ENDPOINT}")
if ENABLE_DEBUG_MODE：
    print(" 调试模式已启用。")
else：
    print(" 调试模式未启用。")
```

运行结果如下：

```
配置 – 最大重试次数：3
配置 – API 地址：https://api.example.com/v1
调试模式未启用。
```

上述示例中定义了整数、浮点数、字符串、布尔值以及特殊值 None 类型的常量。将这些配置项定义为常量，使得它们在代码中易于识别。同时，如果需要修改配置 (如更改 API_ENDPOINT)，只需在定义处修改一次即可，提高了代码的可维护性。

2.2.2　变量

1. 变量的定义

在 Python 编程中，变量是用于存储数据且值可以动态更改的量。作为程序中最基本的数据存储单元，变量能够保存和操作不同类型的数据，包括整数、浮点数、字符串等。在程序执行过程中，变量的值可按需要修改，这使得变量在编程中具有极大的灵活性。

变量的定义和使用示例如下：

```
# 定义变量
name = "Alice"                                     # 字符串变量
age = 25                                           # 整数变量
height = 1.75                                      # 浮点数变量
# 输出变量的值
print(f " 姓名 : {name}")
print(f " 年龄 : {age}")
print(f " 身高 : {height} 米 ")
```

运行结果如下：

```
姓名 : Alice
年龄 : 25 岁
身高 : 1.75 米
```

上述代码中定义了三个变量，即 name、age 和 height，分别用于存储字符串、整数和浮点数。程序输出这些变量的值。

在为变量命名时，必须遵循“2.1.3 标识符”一节中详述的规则。为保证代码的清晰和一致性，推荐采用蛇形命名法来命名变量，如 user_name 或 total_score。

变量与常量 (如上一节讨论的 PI) 的主要区别在于其可变性。变量的值可在程序执行过程中动态更改，而常量的值在程序执行过程中是固定不变的。这种灵活性使变量成为编程中动态处理信息的核心机制。示例代码如下：

```
# 定义常量
PI = 3.14159                                    # 圆周率常量
# 定义变量
radius = 5                                      # 圆的半径
area = PI * (radius ** 2)                       # 计算圆的面积
# 输出圆的面积
print(f " 半径为 {radius} 的圆的面积是 : {area}")
# 修改变量的值
radius = 10
area = PI * (radius ** 2)                       # 重新计算圆的面积
# 输出新的圆的面积
print(f " 半径为 {radius} 的圆的面积是 : {area}")
```

运行结果如下：

```
半径为 5 的圆的面积是 : 78.53975
半径为 10 的圆的面积是 : 314.159
```

在上述代码中，radius 是一个变量，它的值由 5 更改为 10，促使 area 重新计算，而常量 PI 的值保持不变。

2. 变量的类型

在 Python 编程中，变量是动态类型机制，即定义变量时不需要显式声明其类型，Python 解释器会根据所赋的值自动确定变量的类型，这一特性是 Python 语言灵活性和易用性的重要体现。也就是说，变量的类型由其赋值的值决定。例如，将一个整型数赋值给变量，该变量的类型就是整型；将一个字符串赋值给变量，该变量的类型就是字符串。示例代码如下：

```
# 定义变量并赋值
num = 10                                        # 整数类型
name = "Alice"                                  # 字符串类型
pi = 3.14                                       # 浮点数类型
is_active = True                                # 布尔类型
# 输出变量的值和类型
print(f "num 的值是 : {num}，类型是 : {type(num)}")
print(f "name 的值是 : {name}，类型是 : {type(name)}")
print(f "pi 的值是 : {pi}，类型是 : {type(pi)}")
print(f "is_active 的值是 : {is_active}，类型是 : {type(is_active)}")
```

运行结果如下：

```
num 的值是 : 10，类型是 : <class "int'>
name 的值是 : Alice，类型是 : <class 'str'>
pi 的值是 : 3.14，类型是 : <class 'float'>
is_active 的值是 : True，类型是 : <class 'bool'>
```

在上述代码中，变量 num、name、pi 和 is_active 分别被赋值为整数、字符串、浮点数和布尔值。Python 解释器根据赋值内容自动确定这些变量的类型，并通过 type( ) 函数输出变量的类型。

3. 变量的重新赋值和覆盖

由于 Python 的动态类型特性，变量类型可通过赋予不同类型的值实现动态改变。同一变量在程序的不同阶段可被赋予不同类型的值。当变量被重新赋值时，其存储的内容会更新，关联的类型也可能随之改变。示例代码如下：

```
# 定义变量并赋值
data = 100                                                    # 整数类型
print(f "data 的值是 : {data}，类型是 : {type(data)}")
# 重新赋值
data = 200
print(f " 重新赋值后，data 的值是 : {data}，类型是 : {type(data)}")
# 覆盖变量
data = "Python"
print(f " 覆盖后，data 的值是 : {data}，类型是 : {type(data)}")
```

运行结果如下：

```
data 的值是 : 100，类型是 : <class 'int'>
重新赋值后，data 的值是 : 200，类型是 : <class 'int'>
覆盖后，data 的值是 : Python，类型是 : <class 'str'>
```

在上述代码中，变量 data 最初被赋值为整数 100，随后被重新赋值为整数 200，其类型仍为 int。接着，变量 data 被覆盖赋值为字符串“Python”，其类型变为 str。重新赋值和覆盖，可以动态地改变变量的值和类型。

2.3　基本数据类型

数据类型是编程语言中用于定义变量和常量的数据形式，它决定了数据的存储方式以及可以执行的操作。通过学习本节内容，读者可以系统掌握数字类型的基本概念及运算、字符串类型的特点及操作方式以及布尔类型的规则和逻辑运算。

2.3.1　数字类型

在 Python 编程中，数字类型是最基本的数据类型，用于表示和操作数值。Python 支持多种数字类型，包括整型 (int)、浮点型 (float) 和复数 (complex)。

1. 整型

Python 中的整型用于表示没有小数的整数。其取值范围受限于可用内存，而不是特定的位数。这一特性赋予了 Python 整数类型高度的灵活性，既适用于处理普通的数值计算，也能处理需要大数运算的复杂场景。简而言之，在 Python 3 中，整型数值采用动态分配的内存存储，因此即使是非常大的整数，也可以直接在代码中处理，而不需要额外的配置。

(1) 整型的表示方法。整型可通过十进制、二进制、八进制和十六进制表示，这些多样化的表示形式为整型数据的灵活使用提供了便利，特别是在计算机底层开发或网络通信等需要处理不同进制的场景中非常有用。示例代码如下：

```
# 十进制表示
decimal_number = 123
# 二进制表示
binary_number = 0b1101
# 八进制表示
octal_number = 0o17
# 十六进制表示
hexadecimal_number = 0x1A
# 输出不同进制的整型变量
```

```
print(f" 十进制 : {decimal_number}, 类型 : {type(decimal_number)}")
print(f" 二进制 : {binary_number}, 类型 : {type(binary_number)}")
print(f" 八进制 : {octal_number}, 类型 : {type(octal_number)}")
print(f" 十六进制 : {hexadecimal_number}, 类型 : {type(hexadecimal_number)}")
```

运行结果如下：

```
十进制 : 123, 类型 : <class 'int'>
二进制 : 13, 类型 : <class 'int'>
八进制 : 15, 类型 : <class 'int'>
十六进制 : 26, 类型 : <class 'int'>
```

(2) 整型的特性。Python 的整型可以表示任意大小的整数，只要内存允许。例如，可以直接表示上亿甚至更大的整数而不溢出。示例代码如下：

```
# 定义一个非常大的整数
large_number = 12345678901234567890123456789O
# 输出大整数及其类型
print(f" 大整数 : {large_number}, 类型 : {type(large_number)}")
```

运行结果如下：

```
大整数：12345678901234567890123456789O，类型：<class 'int'>
```

(3) 与其他类型的自动转换。整型可以与浮点数或复数自动参与运算，结果会根据运算规则进行类型转换。示例代码如下：

```
# 定义整型和浮点型变量
int_number = 10
float_number = 3.14
# 执行混合运算
result = int_number + float_number
# 输出运算结果及其类型
print(f " 混合运算结果 : {result}, 类型 : {type(result)}")
```

运行结果如下：

```
混合运算结果 : 13.14, 类型 : <class 'float'>
```

2. 浮点型

浮点型是 Python 中用于表示实数的一种数据类型，适合描述带有小数部分的数值。浮点型数据在科学计算、数据分析和工程计算中具有广泛的应用价值。

Python 的浮点型数据基于 IEEE 754 标准实现，采用双精度 (64 位) 浮点数表示，虽能覆盖较大的数值范围，但小数部分的精度有一定限制。因此，浮点型在使用过程中可能会出现精度损失的问题，需要程序设计者注意。

(1) 浮点型的表示方法。浮点型的表示方法有普通表示法和科学计数法。普通表示法，直接使用小数形式表示，如 3.14；科学计数法，使用 e 或 E 表示指数部分，如 1.23e4。这些多样化的表示方法，使浮点型数值能够灵活适应不同精度和数量级。示例代码如下：

```
# 普通表示法
a = 3.14
# 科学计数法
b = 1.23e4                                    # 等价于 12300.0
# 输出浮点型变量的值和类型
print(f "a 的值是 : {a}, 类型是 : {type(a)}")
print(f "b 的值是 : {b}, 类型是 : {type(b)}")
```

运行结果如下：

```
a 的值是 : 3.14, 类型是 : <class 'float'>
b 的值是 : 12300.0, 类型是 : <class 'float'>
```

上述代码中定义并输出了使用普通表示法和科学计数法表示的浮点型变量，展示了 Python 对浮点型数值的支持。

(2) 浮点型的特性。浮点型可表示非常大的或非常小的数值范围，具体范围与底层实现有关。浮点型的数据存储是有限的，因此可能会产生舍入误差。示例代码如下：

```
# 浮点型的精度限制示例
x = 0.1 + 0.2
print(f "0.1 + 0.2 的结果是 : {x}")
```

运行结果如下：

```
0.1 + 0.2 的结果是 : 0.30000000000000004
```

在上述代码中，计算 0.1 + 0.2 的结果可能会出现微小的舍入误差，这表现出浮点型的精度限制问题。

3. 复数

复数在 Python 中用于表示由实部和虚部组成的数值类型。复数广泛应用于电路分析、信号处理、量子计算以及其他科学计算领域。Python 原生支持复数类型，允许开发者直接在代码中操作和运算复数，这一特性在编程语言中较为独特。

Python 中的复数使用 $a + b$j 的形式表示，其中 a 是实部，b 是虚部，j 是虚数单位，表示 $\sqrt{1}$。虚部的数值部分可以是正数、负数或零。

复数的表示方法主要包括直接赋值和类型转换。

(1) 直接赋值。通过 $a + b$j 的形式直接定义复数。

(2) 类型转换。使用 complex(real，imag) 构造函数，将实部和虚部分别作为参数，生成复数。

2.3.2　字符串类型

字符串 (在 Python 中类型为 str) 是 Python 中常用且功能强大的基本数据类型之一，被广泛应用于存储和处理文本数据。它是由一系列字符组成的有序集合，每个字符均占据特定的位置。无论是单词、句子，还是更复杂的文本内容，字符串都能有效表示和处理。在 Python 中，字符串是一种不可变的数据类型，即一旦字符串被创建，其内容将无法直接被修改。如果需要更改字符串的内容，只能通过字符串操作生成新的字符串。这种不可变性不仅保证了字符串的安全性，也优化了内存使用和性能。

在 Python 3 中，字符串默认采用 Unicode 编码。Unicode 是一种国际通用的字符集，为世界上几乎所有的文字、符号和表情定义了统一的编码规则。Unicode 的出现有效解决了不同语言和地区间字符编码冲突的问题，使在同一个程序中处理多语言文本成为可能。Python 3 的字符串类型 (str) 直接表示 Unicode 字符，而不像 Python 2 中需要区分 Unicode 字符和普通字符串。这一设计显著提高了字符串处理的易用性和兼容性。

1. 字符串的表示方法

在 Python 中，字符串可以由单引号、双引号或三引号包裹的字符序列表示。不同的表示方法提供了灵活性，以便开发者在不同的场景中使用字符串。

(1) 单引号字符串。单引号字符串是使用单引号 (') 包裹的字符序列，如 'Hello'。单引号字符串通常用于表示简单的文本数据。示例代码如下：

```
# 单引号字符串
single_quote_str = 'Hello'
```

```
3    # 输出单引号字符串及其类型
4    print(single_quote_str)
5    print(type(single_quote_str))
```

运行结果如下：

```
Hello
<class 'str'>
```

上述代码中定义了一个单引号字符串 single_quote_str，并输出其值和类型。

(2) 双引号字符串。双引号字符串是使用双引号 (") 包裹的字符序列，如 "World"。双引号字符串与单引号字符串功能相同，但在包含单引号字符时使用双引号字符串更为方便。示例代码如下：

```
1    # 双引号字符串
2    double_quote_str = "World"
3    # 输出双引号字符串及其类型
4    print(double_quote_str)
5    print(type(double_quote_str))
```

运行结果如下：

```
World
<class 'str'>
```

上述代码中定义了一个双引号字符串 double_quote_str，并输出其值和类型。

(3) 三引号字符串。三引号字符串是使用三引号 (''' 或 """) 包裹的字符序列，可以定义多行字符串，如 '''Multiline String'''。三引号字符串可以跨越多行，适用于表示包含换行符的长文本。

示例代码如下：

```
1    # 三引号字符串
2    triple_quote_str = '''This is a
3    multiline string
4    example.'''
5    # 输出三引号字符串及其类型
6    print(triple_quote_str)
7    print(type(triple_quote_str))
```

运行结果如下：

```
This is a
multiline string
example.
<class 'str'>
```

上述代码中定义了一个三引号字符串 triple_quote_str，并输出其值和类型。

(4) 包含引号的字符串。当字符串中需要包含引号字符时，可以使用不同类型的引号包裹字符串，如在单引号字符串中嵌入双引号字符，或在双引号字符串中嵌入单引号字符。示例代码如下：

```
1    # 包含双引号的单引号字符串
2    quote_str1 = 'He said, "Hello!"'
3    # 包含单引号的双引号字符串
4    quote_str2 = "It's a beautiful day."
5    # 输出包含引号的字符串
6    print(quote_str1)
7    print(quote_str2)
```

运行结果如下：

```
He said, "Hello!"
It's a beautiful day.
```

上述代码中定义了包含引号字符的字符串 quote_str1 和 quote_str2，并输出其值。

(5) 转义字符。在 Python 字符串中，反斜杠 (\) 扮演着特殊角色——转义字符 (escape character)。当

反斜杠出现在某些特定字符之前时，它会改变这些字符的原始含义，或用于表示一些无法直接输入的特殊控制字符。例如，如果想在一个用单引号定义的字符串中包含单引号本身，需要使用“\'”来转义，“\n”用来表示一个换行符，“\t”表示一个制表符 (通常相当于几个空格的宽度，用于对齐)，“\\”则用来表示一个普通的反斜杠字符本身。

表 2.1 列出了更多常见的转义字符。

表 2.1　常见转义字符

转义字符	含义	示例代码	输出结果
\'	单引号	'She\'s here.'	She's here.
\"	双引号	"He said：\"Hello\"."	He said："Hello".
\\	反斜杠	"This is a backslash：\\."	This is a backslash: \.
\n	换行	"Line1\nLine2"	Line1 Line2
\t	制表符	"Name：\tHarry"	Name:　Harry
\r	回车符	"Hello\rWorld"	World
\b	退格符	"AB\bC"	AC
\uXXXX	16 位 Unicode 字符	"\u4F60\u597D"	你好
\UXXXXXXXX	32 位 Unicode 字符	"\U0001F600"	😀
\N{name}	Unicode 字符 (按名称)	"\N{GREEK CAPITAL LETTER DELTA}"	Δ
\000	八进制值	"\101"	A
\xHH	十六进制值	"\x41"	A

下面的代码示例将演示如何在字符串中使用多种转义字符：

```
# 演示多种转义字符的使用
# 包含换行 (\n)、制表符 (\t)、单引号 (\')、双引号 (\")、反斜杠 (\\)
# 以及 Unicode 字符 (\uXXXX)
complex_string = ' 他说 : "Python\'s escape sequences are useful!"\n\t\\\u4F60\u597D 世界 !\\'
                 # 注意上面这行为了演示，故意把字符串分成了两行
                 # 但实际上它们是在代码层面连接成一个字符串的
''' 原始字符串 (Raw String) 加上前缀 r 时，
  表示原始字符串中的反斜杠失去转义作用，按其字面意义处理 '''
raw_string = r' 这是一个原始字符串：\n 不会换行，\\ 也只是两个反斜杠。'
print("--- 包含转义字符的字符串输出 ---")
print(complex_string)
print("\n--- 原始字符串输出 ---")
print(raw_string)
# 演示其他转义：退格符 (\b) 和制表符 (\t) 的效果
print("\n--- 其他转义效果 ---")
print(" 使用退格符 : A B\b C")#\b 会尝试删除前面的字符 B
print(" 使用制表符对齐 : \nName\tAge\tCity\nAlice\t30\tNew York\nBob\t25\tParis")
```

运行结果如下：

```
--- 包含转义字符的字符串输出 ---
他说 : "Python's escape sequences are useful!"
        \ 你好世界 !\
```

```
--- 原始字符串输出 ---
这是一个原始字符串 : \n 不会换行，\\ 也只是两个反斜杠。

--- 其他转义效果 ---
使用退格符 : A    C
使用制表符对齐 :
Name        Age        City
Alice       30         New York
Bob         25         Paris
```

在上述代码中，第 4 行定义了 complex_string，它演示了多种转义字符。

① \" 和 \' 用于在字符串内部包含双引号和单引号。

② \n 实现了换行。

③ \t 在换行后插入了一个制表符，产生了缩进效果。

④ \\ 输出了一个反斜杠字符。

⑤ \u4F60\u597D 使用 Unicode 编码输出了汉字“你好”。

⑥ 字符串末尾的 \\ 保证了即使字符串跨行定义 (第 5、第 6 行)，最后一个反斜杠也被视为字面量输出 (注意：Python 中字符串字面量跨行时会自动连接，这里的 \ 效果是输出一个反斜杠)。

第 9 行定义了一个原始字符串 (raw string)raw_string，在字符串引号前加上 r 前缀。在原始字符串中，所有的反斜杠都失去了转义功能，被当作普通字符处理。因此，输出时 \n 和 \\ 都按原样显示。原始字符串在处理正则表达式或 Windows 文件路径时非常有用。

第 11 行和第 13 行分别打印了这两个字符串，可以清晰地看到转义字符的效果和原始字符串的区别。

第 16 行演示了退格符 \b，它会尝试将光标向左移动一格并可能覆盖前一个字符 (具体效果可能依赖于终端环境)，这里 B 被 C 覆盖了。

第 17 行通过 \n 和 \t 组合，演示了如何使用制表符来尝试在控制台中创建对齐的列。

2. 字符串的特性

字符串作为 Python 中的重要基础数据类型之一，具有以下三大显著特性：不可变性、索引与切片以及支持格式化。这些特性为字符串的操作提供了灵活性和便利性，能够满足绝大多数文本处理场景的需求。以下将对每一特性进行详细阐述。

(1) 不可变性。字符串是 Python 中的一种不可变 (immutable) 数据类型。一旦字符串对象被创建，其内容将无法直接修改。如果需要对字符串内容进行变更，则需要通过操作生成新的字符串。

(2) 索引与切片。Python 中的字符串可以看作由字符组成的序列，因此支持通过索引访问单个字符，或通过切片操作访问字符串的部分内容。

① 索引的特性。索引从 0 开始，支持正向索引和负向索引；正向索引从字符串的左端开始计数 (0，1，2，...)，负向索引从右端开始计数 (−1，−2，...)。

② 切片的特性。切片操作可通过［start：end：step］的形式访问字符串的子串。start 表示切片的起始位置，end 表示结束位置 (不包含该位置的字符)，step 表示步长 (默认为 1)。

示例代码如下：

```
1    # 示例 1：字符串的索引操作
2    s = "Python"
3    print(" 第一个字符 : ", s[0])
```

```
print(" 最后一个字符 : ", s[-1])
# 示例 2：字符串的切片操作
print(" 前四个字符 : ", s[:4])
print(" 从第三个字符开始 : ", s[2:])
print(" 反向切片 : ", s[::-1])
```

运行结果如下：

```
第一个字符 : P
最后一个字符 : n
前四个字符 : Pyth
从第三个字符开始 : thon
反向切片 : nohtyP
```

上述代码展示了索引操作允许快速访问字符串中的单个字符。切片操作提供了灵活的子串提取方式，尤其是支持负向切片和步长调整。

(3) 支持格式化。字符串格式化能够高效地将变量插入字符串模板，用于生成动态内容。Python 提供了三种主流的字符串格式化方式。

【拓展阅读 2-1】Python 字符串格式化方法详解

① 旧式格式化。这种格式化方式使用 % 运算符。

② str.format( ) 方法。通过占位符 {} 指定插入位置。

③ f-string(格式化字符串字面量)。通过在字符串前加 f，直接嵌入变量。

示例代码如下：

```
# 示例 1：旧式格式化
name = "Alice"
age = 25
print(" 旧式格式化 : 姓名 : %s, 年龄 : %d"%(name, age))
# 示例 2：str.format( ) 方法
print("str.format( ) 方法 : 姓名 : {}, 年龄 : {}".format(name, age))
# 示例 3：f-string 格式化
print(f "f-string 格式化 : 姓名 : {name}, 年龄 : {age}")
```

运行结果如下：

```
旧式格式化 : 姓名 : Alice, 年龄 : 25
str.format( ) 方法 : 姓名 : Alice, 年龄 : 25
f-string 格式化 : 姓名 : Alice, 年龄 : 25
```

上述代码展示了每种格式化方法的适用场景，其中 f-string 语法简单直观，是现代 Python 开发的首选。

3. 字符串的常见操作与方法

字符串是 Python 中功能强大的数据类型，提供了多种内置操作和方法以满足文本处理需求。以下将详细阐述字符串的基本操作、内置函数及常用字符串方法。

(1) 基本操作。字符串在 Python 编程中支持多种基本操作，包括字符串连接、重复和比较。这些基本操作为字符串处理提供了强大的灵活性和便捷性，使得文本数据的操作更加简单和高效。表 2.2 列出了字符串常见的基本操作。

表 2.2　字符串基本操作

操作类型	运算符 / 方法	示例代码	说明	输出结果
字符串连接	+	"Hello"+", "+"World!"	使用 + 将多个字符串连成一个新字符串	Hello, World!

（续表）

操作类型	运算符 / 方法	示例代码	说明	输出结果
字符串重复	*	"Python!"*3	使用 * 生成重复指定次数的字符串	Python! Python! Python!
相等比较	==	"abc"=="abc"	判断两个字符串内容是否完全一致	True
不等比较	!=	"abc"!="xyz"	判断两个字符串内容是否不同	True
字典序比较	</>	"abc"<"xyz"	按 Unicode 编码值比较字符串的字典序	True
忽略大小写	lower( )	"Python".lower( )=="python".lower( )	将字符串统一为小写后再进行比较	True

下面的代码示例将具体演示表 2.2 中的基本操作：

```
# 定义示例字符串
str_hello = "Hello"
str_world = "World"
str_python = "Python"
separator = ", "
num = 3
#--- 字符串连接 (+)---
greeting = str_hello + separator + str_world +"!"
print(f " 连接示例 : {greeting}")
#--- 字符串重复 (*)---
repeated_python =(str_python +" ")*num
print(f " 重复示例 : {repeated_python}")
#--- 相等比较 (==)---
is_equal_same =(str_hello == "Hello")
is_equal_different_case =(str_hello == "hello")                    # 大小写敏感
print(f " 相等比较 ('Hello' == 'Hello'): {is_equal_same}")
print(f " 相等比较 ('Hello' == 'hello'): {is_equal_different_case}")
#--- 不等比较 (!=)---
is_not_equal_diff =(str_hello!= str_world)
is_not_equal_same =(str_hello!="Hello")
print(f " 不等比较 ('Hello'! = 'World'): {is_not_equal_diff}")
print(f" 不等比较 ('Hello'! = 'Hello'): {is_not_equal_same}")
#--- 字典序比较 (<, >)---
# 比较基于字符的 Unicode 编码值
compare_alpha =("apple" < "banana")
compare_case =(str_python < "python")                              # 大写字母编码值小于小写字母
print(f " 字典序比较 ('apple' < 'banana'): {compare_alpha}")
print(f " 字典序比较 ('Python' < 'python'): {compare_case}")
#--- 忽略大小写的比较 ( 使用 lower( ) 方法 )---
str_a_upper = "PYTHON"
str_a_lower = "python"
compare_ignore_case =(str_a_upper.lower( )== str_a_lower.lower( ))
print(f " 忽略大小写比较 ('PYTHON'.lower( )== 'python'.lower( )): {compare_ignore_case}")
```

运行结果如下：

```
连接示例 : Hello, World!
重复示例 : Python Python Python
相等比较 ('Hello' == 'Hello'): True
相等比较 ('Hello' == 'hello'): False
不等比较 ('Hello'! = 'World'): True
不等比较 ('Hello'! = 'Hello'): False
字典序比较 ('apple' < 'banana'): True
字典序比较 ('Python' < 'python'): True
忽略大小写比较 ('PYTHON'.lower( ) == 'python'.lower( )): True
```

(2) 内置函数。Python 提供了一系列强大的内置函数，用于操作和处理字符串中的常见任务。这些函数在设计上既简洁又高效，为开发者提供了丰富的工具支持。以下将详细介绍两个重要的字符串相关内置函数 len() 和 str()，并结合实际场景和示例代码进行说明。

① 获取字符串长度 (len())。len() 函数用于获取字符串的长度，即字符串中字符的个数。该函数返回一个整数，表示字符串的长度。示例代码如下：

```
# 示例 1：获取字符串长度
simple_str = "Hello, Python!"
length = len(simple_str)                                    # 使用 len() 获取字符串长度
print(" 字符串内容 : ", simple_str)
print(" 字符串长度 : ", length)
# 示例 2：计算空字符串的长度
empty_str = ""
print(" 空字符串的长度 : ", len(empty_str))
# 示例 3：统计用户输入的文本长度
user_input = input(" 请输入一段文本 : ")                      # 获取用户输入
print(f" 您输入的文本长度为 : {len(user_input)}")
```

运行结果如下：

```
字符串内容 : Hello, Python!
字符串长度 : 14
空字符串的长度 : 0
请输入一段文本 : Hello, World! [ 此处按下 Enter 键 ]
您输入的文本长度为 : 13
```

在上述代码中，示例 1 展示了 len() 的基础用法，直接统计字符串中的字符数量。示例 2 表明即使字符串为空，len() 也能够准确返回其长度为 0。示例 3 结合用户交互，说明了 len() 在动态字符串处理中的实际应用。

② 转换为字符串 (str())。str() 函数用于将其他数据类型转换为字符串。该函数返回一个字符串表示形式，可以用于将数字、布尔值、列表等数据类型转换为字符串。示例代码如下：

```
# 示例 1：将数值转换为字符串
num = 42
pi = 3.14159
print(" 整数的字符串形式 : ", str(num))
print(" 浮点数的字符串形式 : ", str(pi))
# 示例 2：将布尔值转换为字符串
is_valid = True
print(" 布尔值的字符串形式 : ", str(is_valid))
# 示例 3：将列表和字典转换为字符串
data_list =[1, 2, 3, "Python"]
data_dict ={"name": "Alice", "age": 25}
print(" 列表的字符串形式 : ", str(data_list))
print(" 字典的字符串形式 : ", str(data_dict))
# 示例 4：拼接字符串
age = 25
message = " 年龄是 " + str(age)+ " 岁。"                     # 将整数转换为字符串后拼接
print(message)
```

运行结果如下：

```
整数的字符串形式 : 42
浮点数的字符串形式 : 3.14159
布尔值的字符串形式 : True
列表的字符串形式 : [1, 2, 3, 'Python']
字典的字符串形式 : {'name': 'Alice', 'age': 25}
年龄是 25 岁。
```

在上述代码中，示例 1 展示了 str() 对整数和浮点数的转换。示例 2 表明布尔值可以被直接转换为

其字符串表示形式。示例 3 强调了 str( ) 能够处理复杂数据结构 (如列表和字典)。示例 4 提供了实际应用场景，将数值类型转换为字符串后，与其他文本安全拼接。

关于更多的字符串的常用内置函数可查阅表 2.3。

表 2.3　字符串的常用内置函数

内置函数	功能描述	示例代码	输出结果
len(s)	获取字符串长度	len("Python")	6
str(obj)	将对象转换为字符串	str(42)	"42"
ord(c)	返回字符的 Unicode 编码值	ord('A')	65
chr(i)	返回对应 Unicode 编码值的字符	chr(65)	'A'
max(s)	返回字符串中 Unicode 值最大的字符	max("Python")	'y'
min(s)	返回字符串中 Unicode 值最小的字符	min("Python")	'P'
reversed(s)	返回反转字符串的迭代器	".join(reversed("Python"))	'nohtyP'
sorted(s)	按字符的 Unicode 编码值对字符串排序	sorted("Python")	['P', 'h', 'n', 'o', 't', 'y']

(3) 常用字符串方法。Python 提供了一系列内置的字符串方法，用于满足文本的常见需求，如大小写转换、去除空白、查找替换以及分割与拼接等。这些方法功能强大且易于使用，能够显著提高字符串处理的效率。以下将对常用的字符串方法进行详细说明，并配以示例代码和解析。

① 大小写转换。字符串的大小写转换方法包括 upper( ) 和 lower( )。upper( ) 可以将字符串中的所有字母转换为大写。lower( ) 可以将字符串中的所有字母转换为小写。示例代码如下：

```
# 示例 1：基本大小写转换
s = "Python Programming"
print(" 转换为大写 : ", s.upper( ))
print(" 转换为小写 : ", s.lower( ))
# 示例 2：统一大小写以进行比较
input_str = "Hello"
expected_str = "hello"
print(" 忽略大小写比较 : ", input_str.lower( )== expected_str.lower( ))
```

运行结果如下：

```
转换为大写 : PYTHON PROGRAMMING
转换为小写 : python programming
忽略大小写比较 : True
```

上述代码说明 upper( ) 和 lower( ) 不改变原字符串，而是返回转换后的新字符串。

② 去除空白。Python 提供了以下方法用于去除字符串两端或单侧的空白。strip( ) 可以去除字符串两端的空白。lstrip( ) 可以去除字符串左侧的空白。rstrip( ) 可以去除字符串右侧的空白。另外，strip( ) 也可去除指定字符。示例代码如下：

```
# 示例 1：去除两端空白
s = "    Hello, World!    "
print(" 去除两端空白 : ", s.strip( ))
# 示例 2：去除单侧空白
print(" 去除左侧空白 : ", s.lstrip( ))
print(" 去除右侧空白 : ", s.rstrip( ))
# 示例 3：去除特定字符
s2 = "###Python###"
print(" 去除特定字符 : ", s2.strip("#"))
```

运行结果如下：

```
去除两端空白 : Hello, World!
去除左侧空白 : Hello, World!
去除右侧空白 :      Hello, World!
去除特定字符 : Python
```

上述代码说明 strip() 方法默认去除空白字符，也可以去除指定字符。

③ 查找和替换。查找和替换操作是文本处理中常见的操作。find() 可以返回子字符串在字符串中第一次出现的索引，未找到时返回 –1。replace() 可以将字符串中的指定子字符串替换为新内容。示例代码如下：

```
# 示例 1：查找子字符串
s = "Python Programming"
print(" 查找 'Pro' 的位置 : ", s.find("Pro"))
print(" 查找不存在的子字符串 : ", s.find("Java"))
# 示例 2：替换子字符串
print(" 替换 'Python' 为 'Java': ", s.replace("Python", "Java"))
# 示例 3：替换多次出现的子字符串
s2 = "banana"
print(" 替换所有 'a': ", s2.replace("a", "o"))
```

运行结果如下：

```
查找 'Pro' 的位置 : 7
查找不存在的子字符串 : –1
替换 'Python' 为 'Java': Java Programming
替换所有 'a': bonono
```

上述代码说明 find() 方法适用于定位子字符串的位置，可结合条件判断子字符串是否存在。replace() 不修改原字符串，而是返回替换后的新字符串。

④ 分割与拼接。分割与拼接方法用于将字符串拆分为多个部分或将多个部分组合成一个字符串。split() 可以根据指定分隔符将字符串拆分为列表。join() 可以将可迭代对象 (如列表) 中的字符串连接为一个整体。示例代码如下：

```
# 示例 1：分割字符串
s = "Python, Java, C++"
languages = s.split(", ")                    # 按逗号分割字符串
print(" 分割后的列表 : ", languages)
# 示例 2：拼接字符串
joined_str ="–".join(languages)              # 使用连字符拼接
print(" 拼接后的字符串 : ", joined_str)
# 示例 3：按空格分割
sentence = "Hello World Python"
print(" 按空格分割 : ", sentence.split())
```

运行结果如下：

```
分割后的列表：[ 'Python', 'Java', 'C++' ]
拼接后的字符串：Python–Java–C++
按空格分割：[ 'Hello', 'World', 'Python' ]
```

上述代码说明 split() 方法默认以空格作为分隔符，可以通过参数指定其他分隔符。join() 方法常用于将分割后的列表重新合并为字符串。

关于更多的字符串的常用内置方法可查阅表 2.4。

表 2.4　字符串常用内置方法

方法	功能描述	示例代码	输出结果
upper( )	将字符串转换为大写	"abc".upper( )	'ABC'
lower( )	将字符串转换为小写	"ABC".lower( )	'abc'
strip( )	去除两端空白或指定字符	" abc ".strip( )	'abc'
lstrip( )	去除左侧空白或指定字符	" abc ".lstrip( )	'abc '
rstrip( )	去除右侧空白或指定字符	" abc ".rstrip( )	' abc'
find( )	查找子字符串，返回索引	"abcabc".find("b")	1
replace( )	替换子字符串	"abcabc".replace("a"，"x")	'xbcxbc'
split( )	按指定分隔符将字符串拆分为列表	"a，b，c".split("，")	['a'，'b'，'c']
join( )	按指定分隔符将列表拼接为字符串	"，".join(['a'，'b'，'c'])	'a，b，c'

2.3.3　布尔类型

布尔类型 (bool) 是 Python 中的一种基本数据类型，用于表示逻辑值，仅有两个取值：True(真) 和 False(假)。布尔值在条件判断、逻辑运算和控制流中至关重要，广泛应用于程序的逻辑控制和决策过程。布尔类型的值可由比较运算、逻辑运算或显式赋值生成。

布尔类型的本质是整数类型的一个子类型，其中 True 等价于整数 1，而 False 等价于整数 0。这种设计使布尔值既能直接参与数学运算，又能在逻辑运算中作为条件表达式的结果。

1. 布尔值的表示方法

在 Python 中，布尔值可直接使用关键字 True 和 False 表示，也可通过比较运算或逻辑运算得出。示例代码如下：

```
# 直接赋值布尔值
is_true = True
is_false = False
# 通过比较运算产生布尔值
a = 10
b = 20
comparison_result = (a < b)
# 通过逻辑运算产生布尔值
logical_result =(a < b)and (a > 5)
# 输出布尔值及其类型
print(f"is_true: {is_true}, 类型 : {type(is_true)}")
print(f"is_false: {is_false}, 类型 : {type(is_false)}")
print(f" 比较运算结果 : {comparison_result}, 类型 : {type(comparison_result)}")
print(f" 逻辑运算结果 : {logical_result}, 类型 : {type(logical_result)}")
```

运行结果如下：

```
is_true: True, 类型 : <class 'bool'>
is_false: False, 类型 : <class 'bool'>
比较运算结果 : True, 类型 : <class 'bool'>
逻辑运算结果 : True, 类型 : <class 'bool'>
```

在上述代码中，布尔值 is_true 和 is_false 通过直接赋值表示，布尔值 comparison_result 通过比较运算生成，布尔值 logical_result 通过逻辑运算生成。程序最后会输出这些布尔值及其类型。

2. 布尔运算的基本规则

布尔运算包括逻辑与 (and)、逻辑或 (or) 和逻辑非 (not) 三种基本运算。这些逻辑运算用于组合和操作布尔值以生成新的布尔值。具体逻辑运算方法详见“ 2.4.4 逻辑运算符”。此外，布尔值也可以直接参与数学运算，True 等价于 1，False 等价于 0。示例代码如下：

```
print(" 布尔值的加法 : ", True + False)          # 输出 : 1
print(" 布尔值的乘法 : ", True*5)                # 输出 : 5
```

3. 布尔值的类型转换

在 Python 中，布尔值可以通过内置函数 bool() 从其他数据类型转换而来。任何非零数值、非空字符串或非空集合 (如列表、元组、集合、字典等) 都会转换为 True，而零数值、空字符串或空集合会转换为 False。示例代码如下：

```
# 数值类型的布尔值
print("bool(100): ", bool(100))
print("bool(0): ", bool(0))
# 字符串类型的布尔值
print("bool('Python'): ", bool("Python"))
print("bool(''): ", bool(""))
# 容器类型的布尔值
print("bool([1, 2, 3]): ", bool([1, 2, 3]))
print("bool([]): ", bool([]))
#None 的布尔值
print("bool(None): ", bool(None))
```

运行结果如下：

```
bool(100): True
bool(0): False
bool('Python'): True
bool(''): False
bool([1, 2, 3]): True
bool([]): False
bool(None): False
```

2.4　运算符与表达式

在 Python 编程中，运算符与表达式是实现各种计算、逻辑判断及数据处理的核心工具。本节将详细介绍 Python 提供的多种运算符及其在表达式中的使用方法。通过学习本节内容，读者可以掌握算术运算符的基本功能与应用、字符运算符的特殊用途、比较和逻辑运算符在条件判断中的重要性以及位运算符的基本原理与应用场景。此外，本节还将介绍成员运算符与一致性运算符的独特作用，并探讨运算符优先级对表达式执行顺序的影响。

2.4.1　算术运算符

算术运算符是 Python 中基本的运算符之一，用于实现各种数学计算和数值处理。它们可对数字类型 (如整数和浮点数) 执行加、减、乘、除等常见操作，是构建数学表达式的核心工具。在 Python 中，算术运算符不仅支持简单的数值运算，还能与变量及更复杂的表达式结合使用。