校企双元“十五五”精品教材
新形态“互联网+”教材

Python程序设计基础

主编　徐　强　胡德洪　张梦帆

南京大学出版社

图书在版编目(CIP)数据

Python 程序设计基础 / 徐强，胡德洪，张梦帆主编. 南京 ：南京大学出版社，2025. 9. - - ISBN 978 - 7 - 305 - 29425 - 9

Ⅰ. TP312.8

中国国家版本馆 CIP 数据核字第 2025SF7915 号

出版发行　南京大学出版社
社　　址　南京市汉口路 22 号　　邮　　编　210093

书　　名　Python 程序设计基础
Python CHENGXU SHEJI JICHU
主　　编　徐　强　胡德洪　张梦帆
责任编辑　吕家慧　　编辑热线　025 - 83592655

照　　排　南京开卷文化传媒有限公司
印　　刷　常州市武进第三印刷有限公司
开　　本　787 mm×1092 mm　1/16 开　印张 10.25　字数 262 千
版　　次　2025 年 9 月第 1 版　　2025 年 9 月第 1 次印刷
ISBN　978 - 7 - 305 - 29425 - 9
定　　价　39.80 元

网　　址：http://www.njupco.com
官方微博：http://weibo.com/njupco
微信服务号：NJUYUNSHU
销售咨询热线：(025)83594756

前　言

近年来，Python 以其简洁的语法、强大的功能库和广泛的应用领域，迅速成为全球最受欢迎的编程语言之一。无论是数据分析、人工智能、Web 开发，还是自动化运维，Python 都展现了卓越的适应性。对于初学者而言，Python 不仅是踏入编程世界的理想选择，更是未来技术探索的基石。本书旨在通过系统化的知识体系与丰富的实践案例，帮助读者从零基础逐步成长为能够独立解决实际问题的 Python 开发者。

本书特色

循序渐进，由浅入深

本书内容设计遵循“基础→应用→进阶”的逻辑，从 Python 环境搭建、语法基础到文件操作、面向对象编程，层层递进。每个模块均以“任务”为驱动，结合代码示例和动手练习，确保读者在理解理论的同时掌握实践技能。

案例导向，注重实战

书中融入了多个贴近实际的应用场景，如“学生管理系统”“银行账户系统”等综合案例，帮助读者将零散的知识点串联成完整的项目开发思维。此外，通过文件批量处理、字符串操作等实用技巧的讲解，进一步强化读者解决实际问题的能力。

内容全面，细节丰富

不仅涵盖 Python 基础语法、数据结构（列表、元组、字典）、函数与模块化编程，还深入探讨了异常处理、文件操作、面向对象等进阶主题。特别针对开发环境配置（如 PyCharm 的使用与优化）、第三方模块管理等内容，提供了详细的操作指南，扫除初学者的环境配置障碍。

教学资源丰富

每章末配有“小结”梳理核心知识点，并通过“习题”巩固学习成果。代码注释清晰，关键步骤配有图示说明，便于读者自查与复习。

适合读者

高校学生：适用于作为计算机相关专业 Python 入门教材，或非计算机专业选修课参考用书。

自学者：适合零基础编程爱好者系统学习 Python，为后续学习数据分析、机器学习等领域奠定基础。

职业开发者：可作为快速回顾 Python 语法及核心编程思想的工具书。

内容结构

模块 1－2　从 Python 发展历程、环境搭建讲起，夯实语法基础，包括变量、运算符、流程控制等。

模块 3－5　深入讲解字符串处理、列表、元组和字典的应用，培养数据处理能力。

模块 6－7　聚焦函数封装与文件操作，提升代码复用性与工程化思维。

模块 8－9　进阶探讨异常处理机制与面向对象编程，完成从脚本编写到软件设计的跨越。

致谢

本书的编写得益于众多开源社区的技术分享与教育工作者的宝贵建议，在此致以诚挚的感谢。特别感谢一线教师和学生的反馈，帮助我们不断优化内容结构，使本书更贴合实际教学需求。本书推荐 Python 相关的资源平台：海豚大数据及人工智能实验室（www.dolphin-labs.com），希望帮助读者和老师扩展练习。

编程学习是一场充满挑战与成就的旅程。希望本书能成为您探索 Python 世界的可靠伙伴，助您在代码的海洋中扬帆远航。书中疏漏之处，恳请读者不吝指正，我们将持续改进，与您共同进步。

编　者

2025 年 3 月

目　录

模块 1

Python 概述

知识目标

1. 了解 Python 的发展历程
2. 了解 Python 的特点和应用领域

技能要求

1. 独立完成 Python 的安装
2. 会使用 PyCharm 新建 Python 文件
3. 掌握 Python 程序的执行原理

同学们，让我们开启 Python 编程的探索之旅！安装 Python 与 PyCharm，如同为创新梦想搭建坚实基座，每一步操作都培养着我们的系统思维与规范意识。模块导入则像打开知识宝库的金钥匙，教会我们在协作中共享智慧，在复用中提升效率。这不仅是技术工具的学习，更是对精益求精工匠精神的传承。愿大家以代码为笔，以模块为墨，在编程实践中书写创新篇章，用技术赋能社会进步，成为有担当的数字时代建设者！

任务 1.1　认识 Python

1.1.1　Python 的发展历程

Python 语言的发展历程可以追溯到 20 世纪 90 年代初，由荷兰人吉多·范罗苏姆(Guido van Rossum)首次发布。以下是一些关键的发展节点。

1989 年开始研制，1991 年 Python 的第一个版本(0.9.0)诞生。这个版本包含了诸多基本的语言特性，如模块、异常处理、函数以及核心数据类型(如字符串、列表等)。

1994 年发布了 Python 1.0，引入了一些现在看来很基本的特性，如 lambda、map、filter 和 reduce，此时 Python 已经有了一个相对完整的语言结构。

Python在Guido van Rossum的引领下不断发展，并在1995年，他在弗吉尼亚州的国家创新研究公司(CNRI)继续他在Python上的工作，并在那里发布了该软件的多个版本。

2000年，Python 2.0发布，引入了一些重要的新特性，包括垃圾回收机制和Unicode支持，使得Python更加适用于多种领域，从Web开发到科学计算。同年，Guido van Rossum和Python核心开发团队转到BeOpen.com并组建了BeOpen PythonLabs团队，之后该团队又转到Digital Creations(现为Zope Corporation)。

2001年，Python软件基金会(PSF)成立，这是一个专为拥有Python相关知识产权而创建的非营利组织。

2008年，为了解决Python 2版本中的一些设计缺陷和不一致性，Python 3.0(也被称为Python 3000或简称为Py3k)发布。这个版本引入了不兼容的语法和库变化，以提高语言的一致性和清晰度。虽然这个转变初期造成了一些困扰，但Python 3最终为Python语言的未来发展奠定了坚实的基础。

随着Python社区的壮大，涌现出大量优秀的第三方库和框架，如NumPy、Django、Flask等，这些工具为Python在数据科学、Web开发等领域的应用提供了强大的支持。

近年来，随着大数据和人工智能的兴起，Python在数据科学、机器学习和深度学习领域变得越来越流行。Python的简单易学和强大的功能使其成为许多领域首选的编程语言。

总的来说，Python语言的发展历程是一个不断演进和壮大的过程，从最初的一个小众语言发展成为如今在全球范围内广泛使用的流行编程语言。

1.1.2 Python语言的特点

1. 简洁易读

Python采用简洁的语法和清晰的代码结构，使得代码易于阅读和理解。相比其他编程语言，Python代码通常更加简洁，可以用更少的代码实现相同的功能。

2. 动态类型

Python是一种动态类型语言，变量的类型在运行时可以自动推断，无需显式声明。这使得Python编程更加灵活，减少了类型转换的繁琐操作。

3. 面向对象

Python支持面向对象编程，可以使用类和对象来组织和管理代码。面向对象的编程范式使得代码更加模块化、可重用和易于维护。

4. 强大的标准库

Python拥有丰富的标准库，包含了各种功能模块，如文件操作、网络通信、数据库连接等。这些标准库可以帮助开发者快速实现各种功能，提高开发效率。

5. 跨平台

Python 可以在多个操作系统上运行，包括 Windows、Linux、Mac 等。这使得开发者可以在不同的平台上开发和部署 Python 程序，提高了程序的可移植性。

6. 强大的第三方库支持

Python 拥有庞大的第三方库生态系统，包含了各种功能强大的库和框架，如 NumPy、Pandas、Django 等。这些库可以帮助开发者快速构建复杂的应用程序，扩展了 Python 的功能和应用领域。

7. 可扩展性

Python 可以通过调用 C/C++编写的扩展模块来提高性能，还可以与其他语言进行混合编程。这使得 Python 既可以用于快速开发原型，又可以用于性能要求较高的应用。

8. 社区支持和活跃度

Python 拥有庞大而活跃的开发者社区，提供了丰富的资源和支持。

以上特点使得 Python 语言在多个领域都有广泛的应用，包括 Web 开发、数据科学、机器学习、人工智能等。

任务 1.2　Python 安装与运行

1.2.1　下载 Python 解释器

进入 Python 官网(https://www.Python.org/)，在 Downloads 页面选择适合操作系统的 Python 版本进行下载。如果使用的是 Windows 操作系统，选择 Windows 版本下载即可。

1.2.2　安装 Python 解释器

下载完成后，双击 Python 安装程序进行安装。在安装过程中，可以选择自定义安装路径，也可以选择默认安装路径。

在安装选项中，建议勾选“Add Python to PATH”选项，这样可以在系统的任何位置都能直接运行 Python 解释器。另外，还可以根据需要选择是否安装 Python 的文档和 pip 等工具。

等待安装完成，完成后可以在命令行中输入“Python”命令来检查 Python 解释器是否安装成功。如果看到 Python 的版本信息，则说明安装成功。

需要注意的是，在安装 Python 解释器时，应该选择与操作系统位数相匹配的版本。例如，如果操作系统是 64 位的，就应该选择 64 位的 Python 版本进行安装。

另外，Python 有多个版本，如 Python 2 和 Python 3。目前 Python 2 已经停止维护，建议使用 Python 3 进行开发。在下载和安装时，请注意选择正确的版本。

在安装完成后，还可以配置 Python 的 IDE（集成开发环境），如 PyCharm、VS Code 等，以便更方便地进行 Python 开发。

1.2.3 Python 的安装与卸载

1. Python 的安装

勾选添加环境，选择自定义安装。

图 1.1 Python 安装程序启动界面

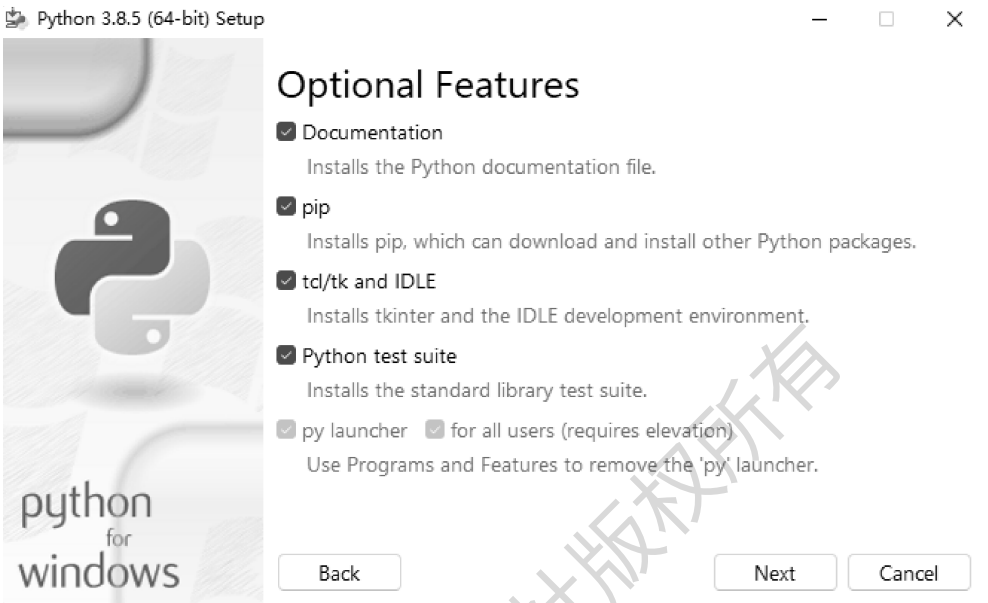

图 1.2　Python 安装程序特征选择

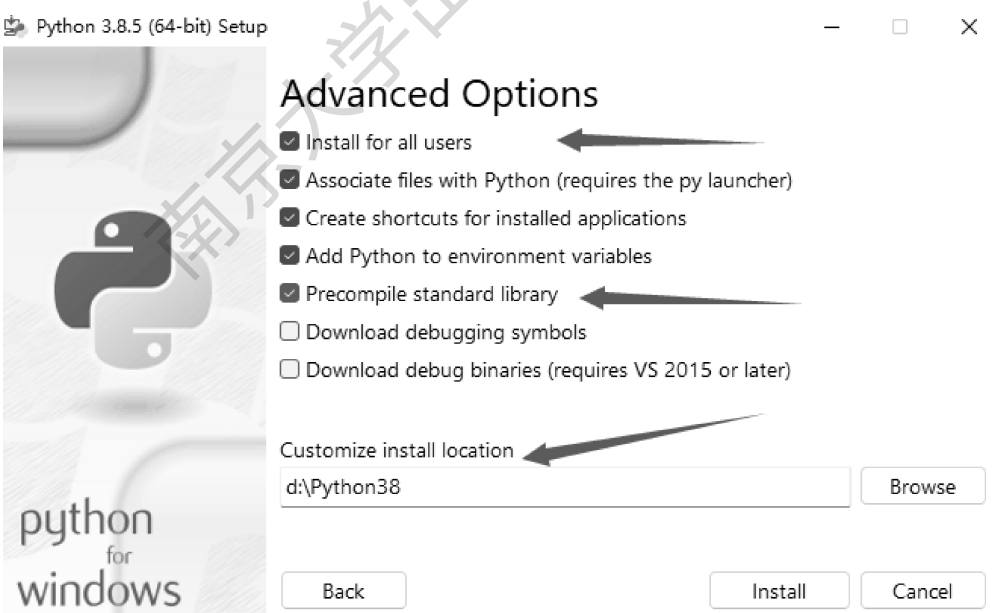

图 1.3　Python 安装程序路径选择

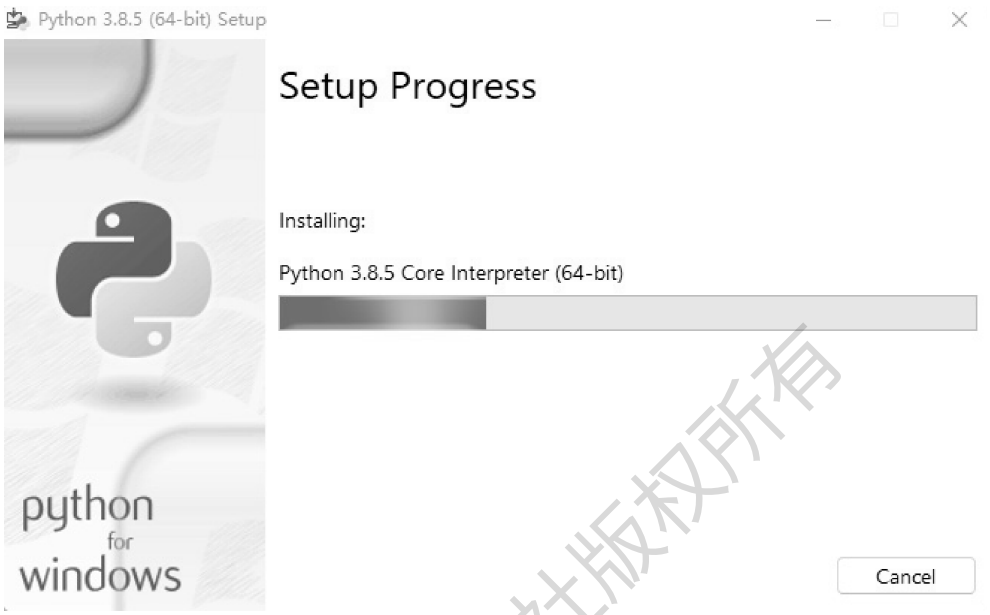

图 1.4　Python 安装进度界面

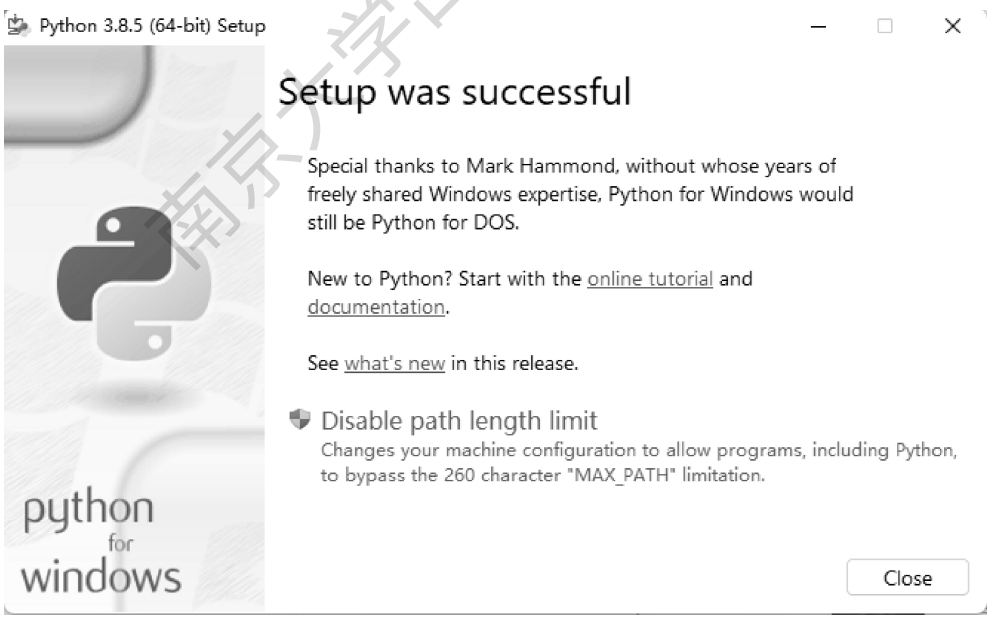

图 1.5　Python 安装完成

2. Python 的卸载

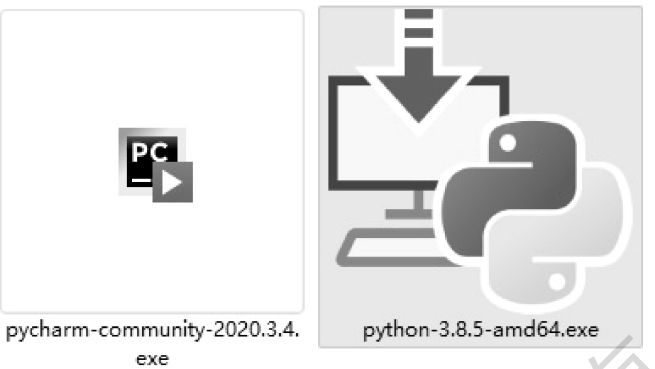

图 1.6　可使用 Python 安装包进行卸载

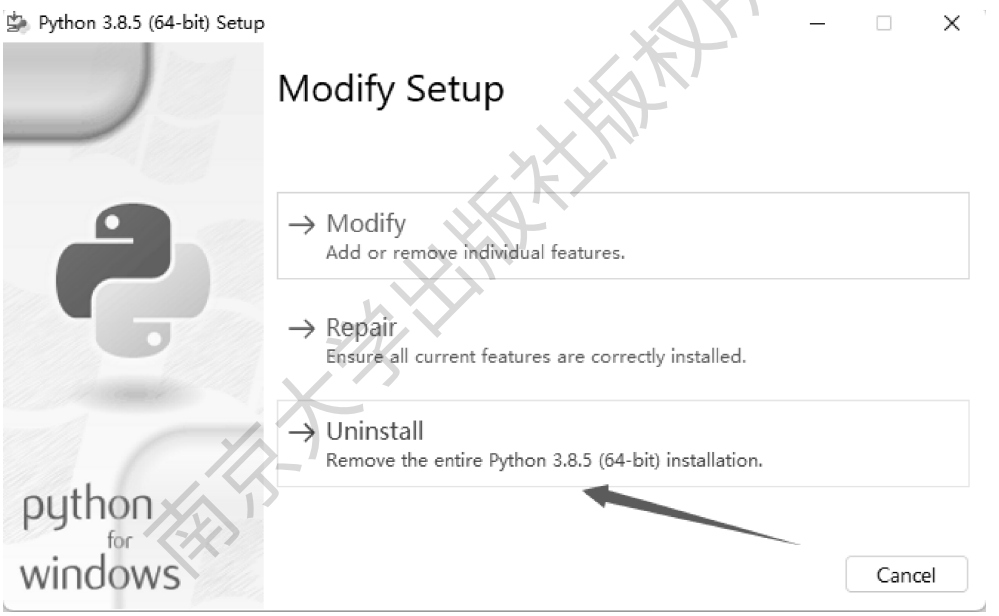

图 1.7　Python 修改设置界面

1.2.4　Python 的运行

安装完 Python 后，可以通过以下几种方式运行 Python 代码。

1. 交互式解释器

打开命令行（在 Windows 上是命令提示符或 PowerShell，在 macOS 和 Linux 上是终端），然后输入 Python 或 Python 3（取决于系统配置和 Python 版本）。这将启动 Python 的交互式解释器，可以在其中逐行输入 Python 代码并立即看到结果，如图 1.8 所示。

```
$ python
Python 3.x.x (default, ...)
Type 'help', 'copyright', 'credits' or 'license' for more information.
>>> print("Hello, World!")
Hello, World!
>>>
```

图 1.8　Python 解释器

输入 exit()或按 Ctrl+D 键(在 Unix 系统上)或按 Ctrl+Z 键(在 Windows 系统上)可以退出解释器。

2. 运行脚本文件

将 Python 代码保存为.py 文件,然后在命令行中使用 Python 或 Python 3 命令加上脚本文件名来运行它。例如,如果有一个名为 script.py 的脚本,可以如图 1.9 所示运行它。

```
$ python script.py
```

图 1.9　运行 script.py 的脚本

如果系统默认使用 Python 3,可能只需要输入如图 1.10 所示代码。

```
$ script.py
```

图 1.10　运行脚本

3. 集成开发环境(integrated development environment,IDE)

除了命令行,还可以使用集成开发环境(IDE)来编写和运行 Python 代码。PyCharm、Visual Studio Code、Spyder 等都是流行的 Python IDE。在 IDE 中,通常可以创建一个新的 Python 文件,编写代码,然后直接通过 IDE 的运行按钮或快捷键来执行代码。

4. Jupyter Notebook

Jupyter Notebook 是一个 Web 应用程序,允许创建和共享包含实时代码、文本和可视化内容的文档。安装 Jupyter 后,可以在命令行中通过输入 jupyter notebook 来启动它,然后在 Web 界面中创建一个新的 Notebook,并在其中编写和运行 Python 代码。

5. Python Shell 或 REPL

有些图形用户界面(graphical user interface,GUI)应用程序提供了 Python Shell 或 REPL(read-eval-print loop)的功能,允许在一个图形界面中与 Python 解释器交互。

确保在尝试运行 Python 代码之前,环境变量已经正确设置,以便在命令行中能够找到 Python 或 Python 3 命令。如果在安装过程中选择了将 Python 添加到系统路径,这一步通常会自动完成。如果没有,可能需要手动设置环境变量。

任务 1.3　PyCharm 下载与安装使用

PyCharm 是一种 Python 的集成开发环境(IDE),旨在帮助用户在使用 Python 语言开发时提高效率。它提供了一系列的功能和工具,包括智能代码补全、实时错误检查、快速修复、自动化代码重构、代码导航、单元测试、版本控制等。

此外,PyCharm 还支持多种 Web 开发框架,如 Django、Flask、Google App Engine、Pyramid 和 web2py,并为这些框架提供了丰富的框架针对性支持。它还与 IPython Notebook 集成,提供交互式 Python 控制台,并支持 Anaconda 和多种科学化的包,如 Matplotlib 和 NumPy。

PyCharm 有专业版和社区版两种。专业版功能更全面,但需要付费;社区版是免费的,但一些高级功能不支持。安装 PyCharm 可以选择从官网下载对应的安装包,然后按照提示进行安装。安装完成后,可以通过 PyCharm 创建项目,选择项目地址和 Python 版本,然后开始编写代码。

总的来说,PyCharm 是一款功能强大的 Python IDE,适用于各种 Python 开发场景,无论是初学者还是专业开发者都可以使用它来提高开发效率。

1.3.1　PyCharm 的下载

下载 PyCharm 可以访问 PyCharm 的官方网站:https://www.jetbrains.com/pycharm/。

在网站上,将看到三个版本可供选择:Professional(专业版,收费)、Community(社区版,免费)和 Education(教育版)。对于大多数用户来说,社区版已经足够使用,这三个版本的主要区别如下。

1. 功能差异

Community 版:这是 PyCharm 的免费版本,适用于个人和小型团队开发人员。它提供了基本的代码编辑、调试、自动完成、版本控制等功能,适合初学者和编程爱好者使用。但是,它缺少一些高级功能,如科学工具、远程开发、Web 开发框架支持等。

Professional 版:这是 PyCharm 的商业版本,需要付费购买。它在 Community 版的基础上增加了许多高级功能,如 Web 开发、Python 分析器、PythonWeb 框架、远程开发、数据库与 SQL 支持等。适合专业开发者、团队和企业使用。

Education 版：这个版本专门为教育机构和学生设计，包含了 Professional 版的所有功能，但仅限于教育用途。学生可以免费使用，而教师需要学校向 PyCharm 官方申请才能使用。

2. 授权和使用

Community 版：免费提供给个人使用，不需要购买激活码。

Professional 版：需要购买激活码才能解锁全部功能，适用于公司或团队进行专业开发。

Education 版：免费提供给学校和学生使用，但教师使用需要学校进行申请。

3. 适用人群

Community 版：适合初学者、编程爱好者和小型项目开发者。

Professional 版：适合专业开发者、大型项目团队、需要高级功能支持的公司或组织。

Education 版：适用于教育机构、教师和学生，主要用于教学和学习目的。

单击“Download”按钮开始下载。根据操作系统（Windows、macOS 等），选择相应的下载链接。

在下载页面，可以选择想要的安装包类型（例如，对于 Windows 用户，可以选择.exe 或.zip 文件）。选择好后，单击“Download”按钮开始下载。

下载完成后，找到保存的安装包文件，双击打开进行安装。

注意，下载和安装软件时，确保从官方或可信的源获取软件，以避免安全风险。同时，安装过程中仔细阅读每个步骤，确保正确完成安装。

1.3.2 PyCharm 的安装

找到下载的 PyCharm 安装包（对于 Windows 用户，通常是.exe 文件），双击打开，开始安装，如图 1.11 所示。

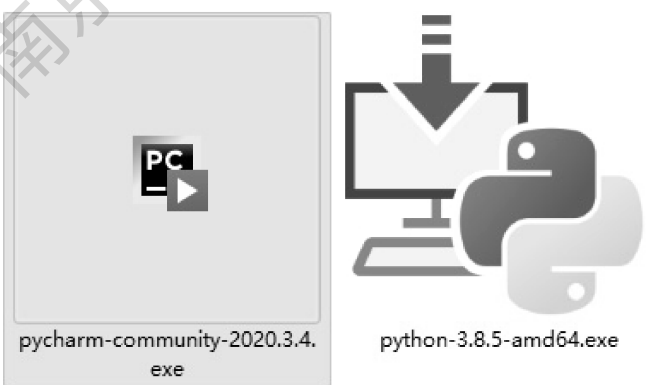

图 1.11 PyCharm 安装程序包

双击.exe 文件后，进入安装向导。

单击“Next”进入安装路径选择页面，选择一个合适的安装位置（尽量不要选择带有中文和空格的目录），如图 1.12 所示。

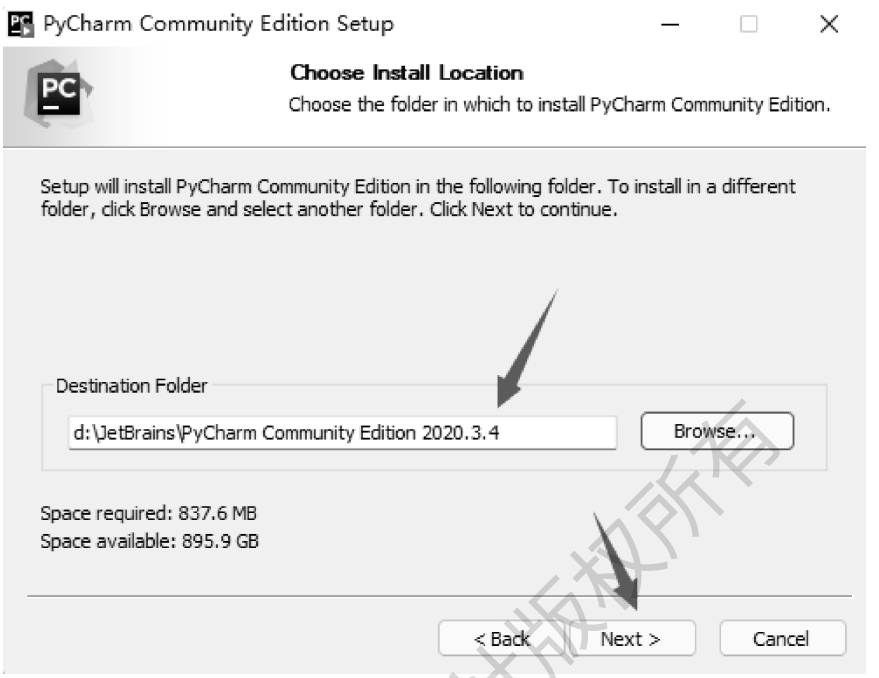

图 1.12　选择安装路径

选择好路径后，再次单击“Next”进入安装选项页面，根据需要勾选相应的选项（通常建议全部勾选），如图 1.13 所示。

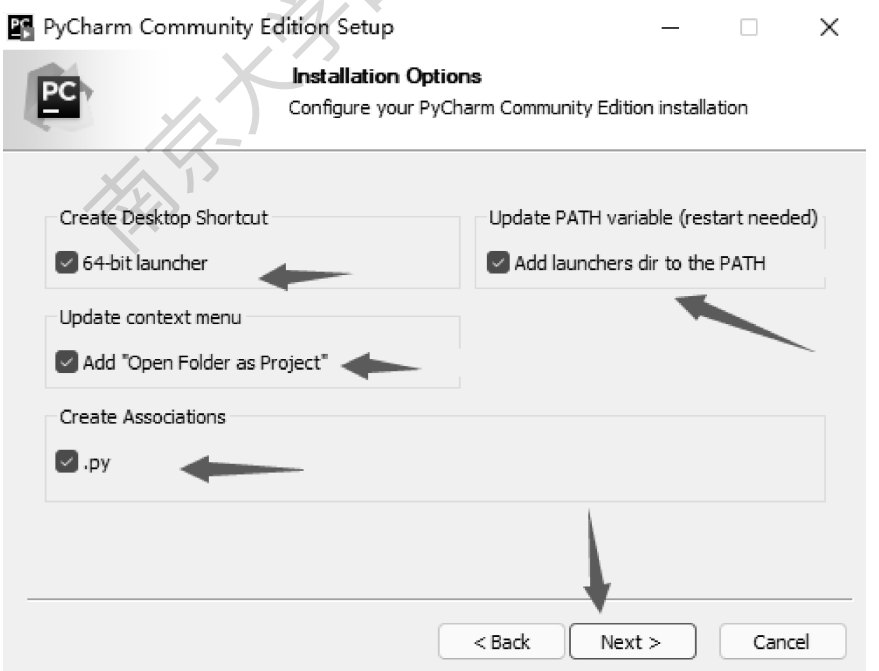

图 1.13　安装选项设置

单击“Next”进入开始菜单文件夹选择页面，通常保持默认设置即可。

单击“Install”开始安装，如图 1.14 所示。

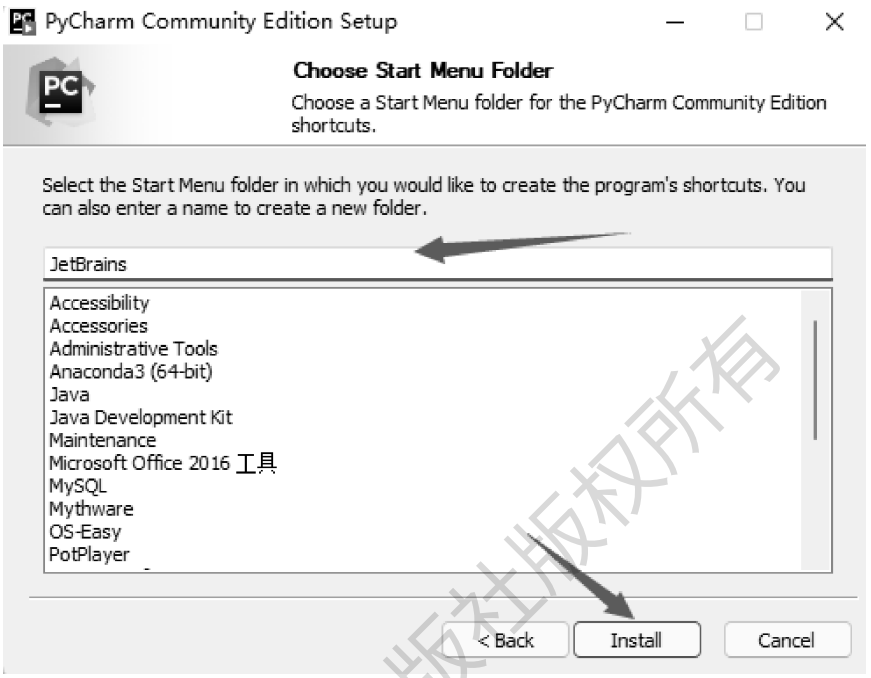

图 1.14　选择开始菜单

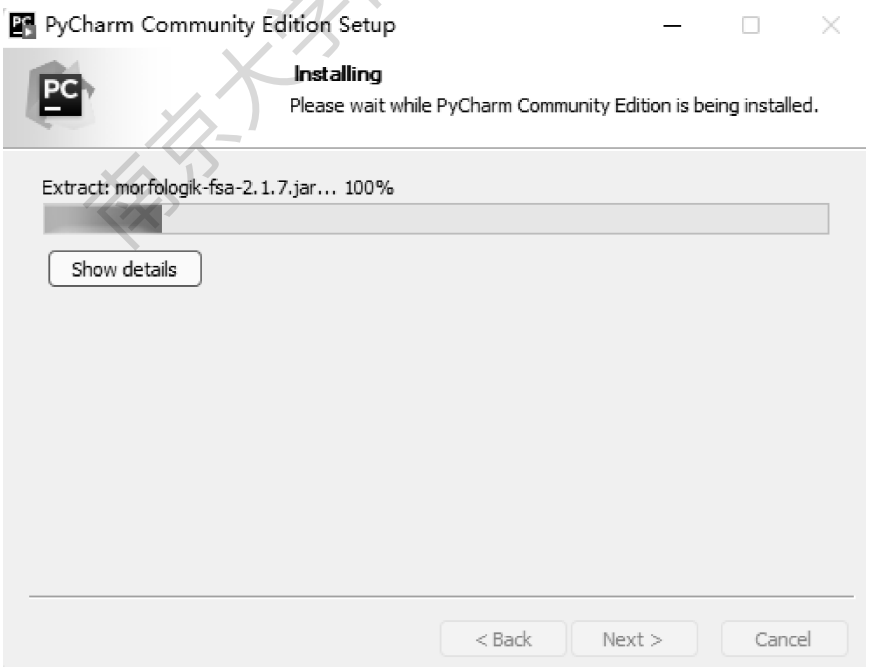

图 1.15　正在安装

安装完成后，选择稍后重启，单击“Finish”完成安装，如图 1.16 所示。

图 1.16　立即重启与稍后重启

安装完成后，可进行一些配置，如设置 Python 解释器、配置虚拟环境等。这些配置可以在 PyCharm 的偏好设置或设置中完成。

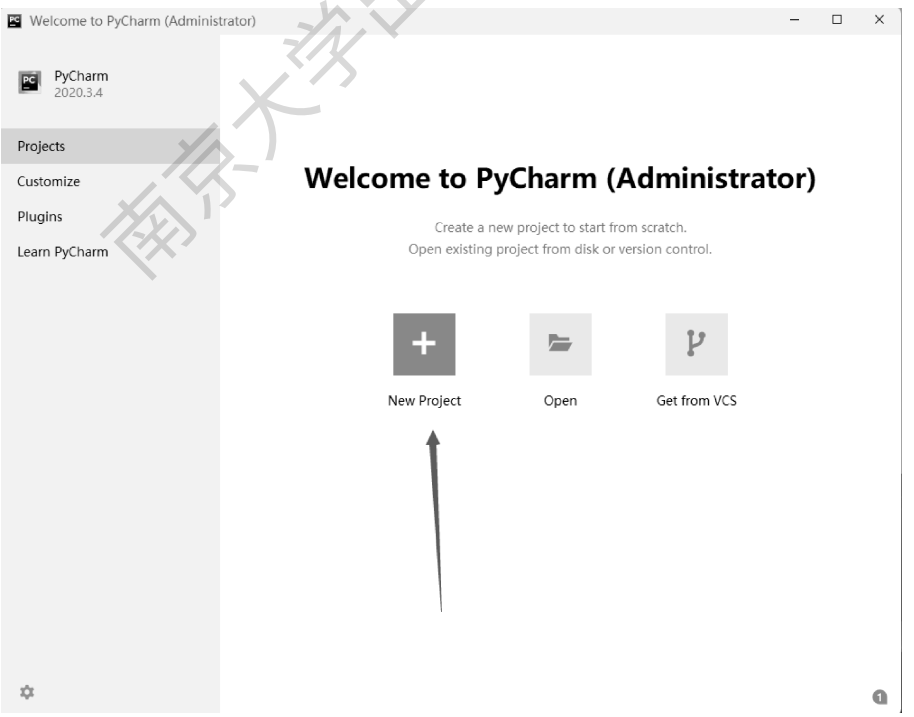

图 1.17　选择新建工程

先指定新建工程所在目录，再确定 Python 安装位置(需在 PyCharm 开始安装前完成)，如图 1.18 所示。

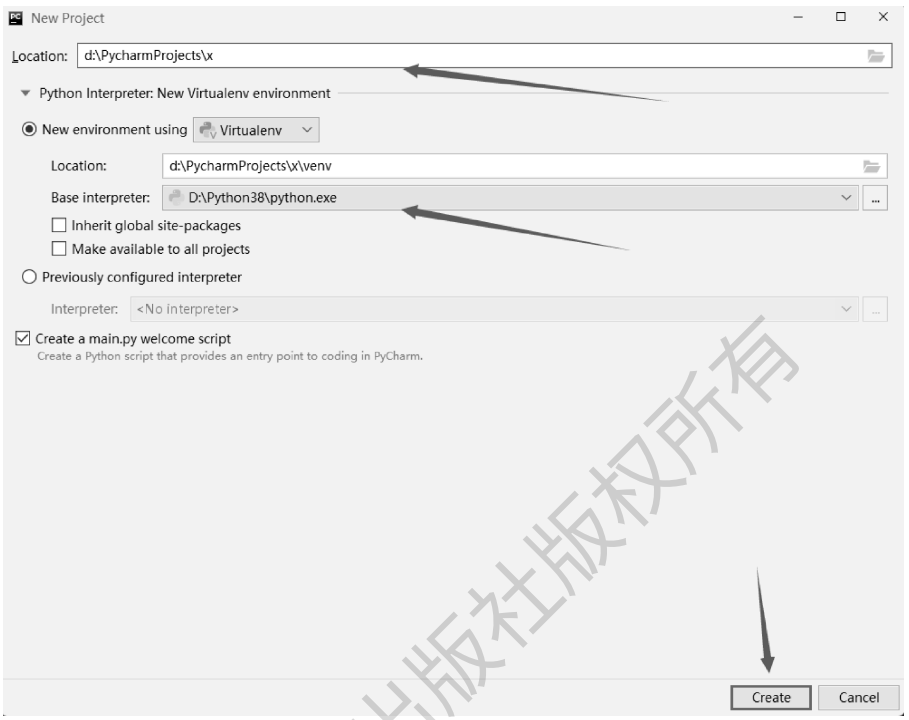

图 1.18　新建工程设置界面

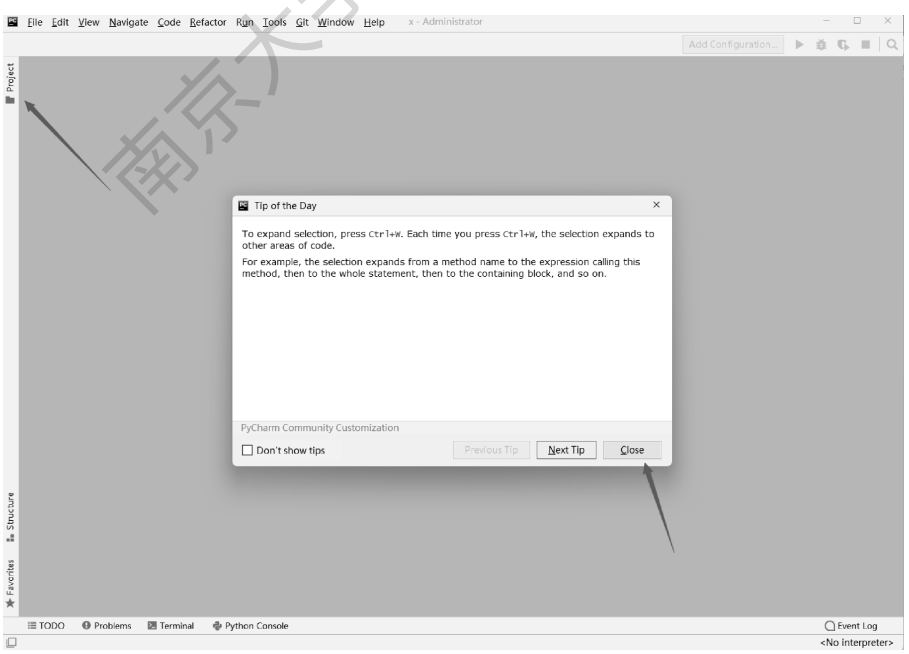

图 1.19　项目管理界面

选择新建 Python 文件，如图 1.20 所示。

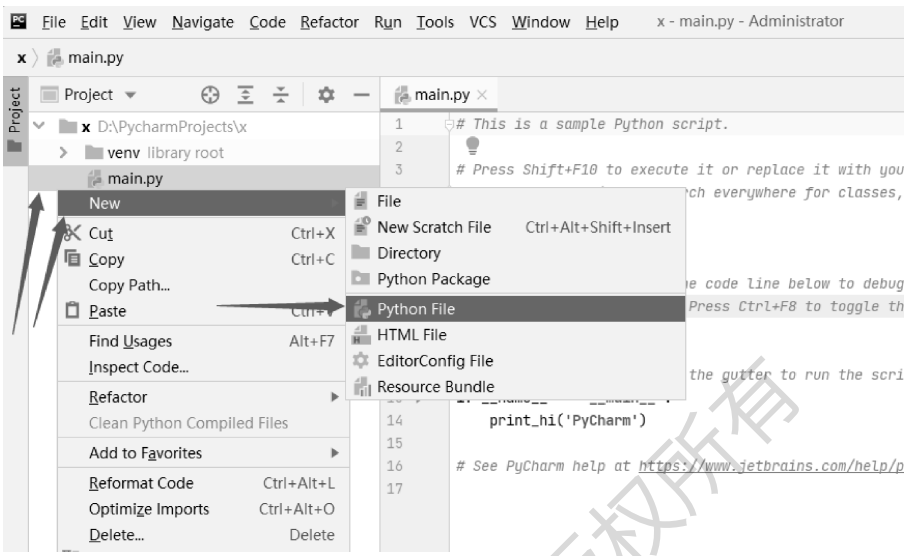

图 1.20　新建 Python 文件

输入需要新建的.py 文件名称，如图 1.21 所示，输入为 1。

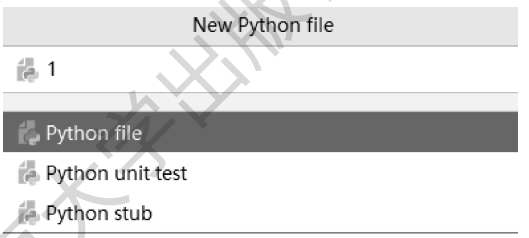

图 1.21　“New Python file”窗口

如图 1.22 所示，工程目录创建完成。

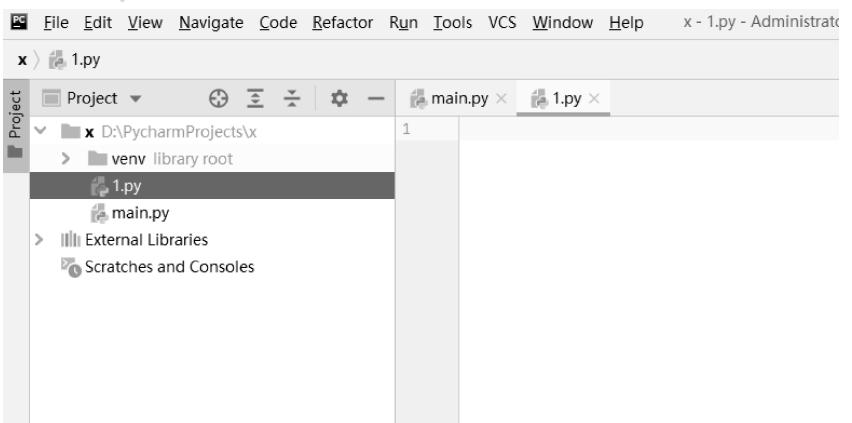

图 1.22　添加 Python 文件后的窗口

1.3.3 PyCharm 的卸载

再次单击 PyCharm 的安装文件，即可进行卸载，如图 1.23 所示。

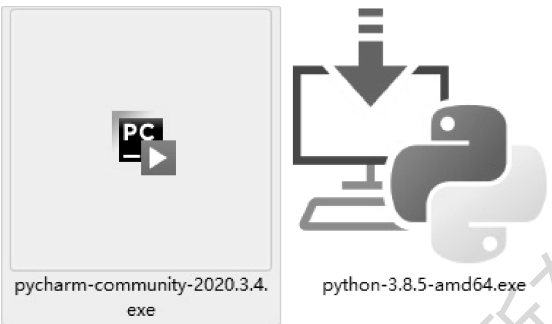

图 1.23 PyCharm 程序卸载

选择“是”，如图 1.24 所示。

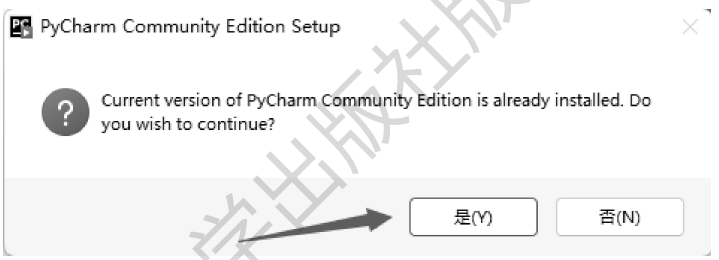

图 1.24 PyCharm 安装程序对话框

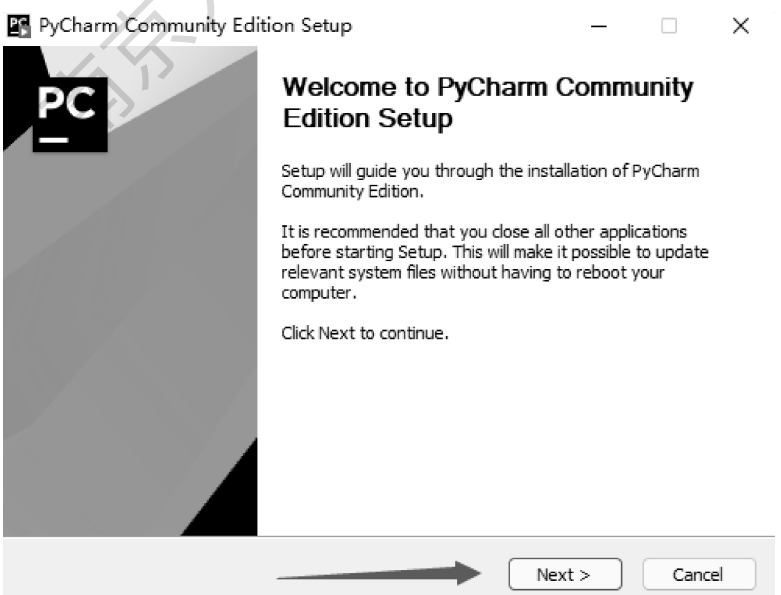

图 1.25 PyCharm 安装程序界面

如图 1.26 所示进行勾选。

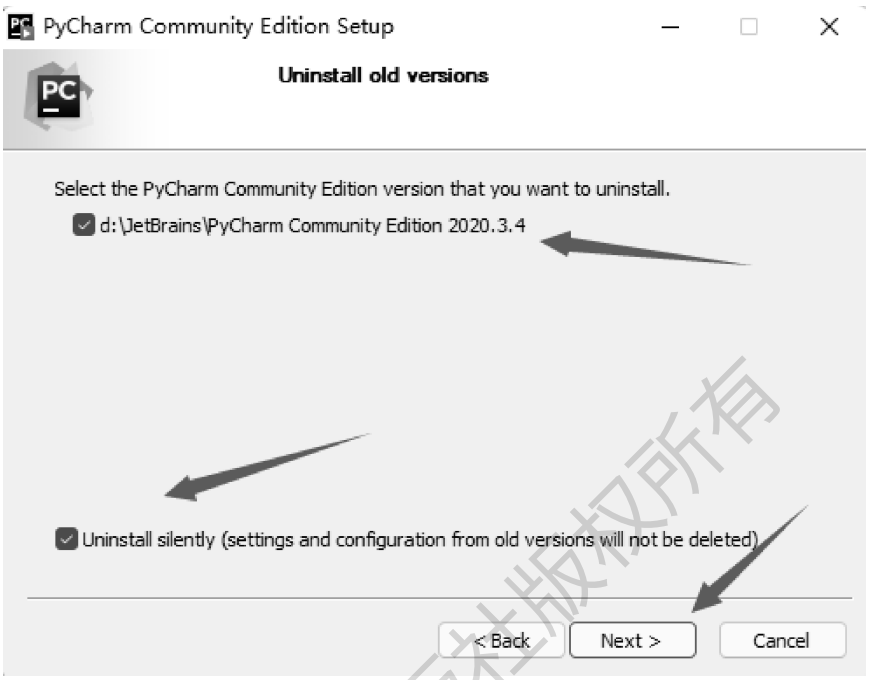

图 1.26　PyCharm 安装程序卸载界面

卸载完成，如图 1.27 所示。

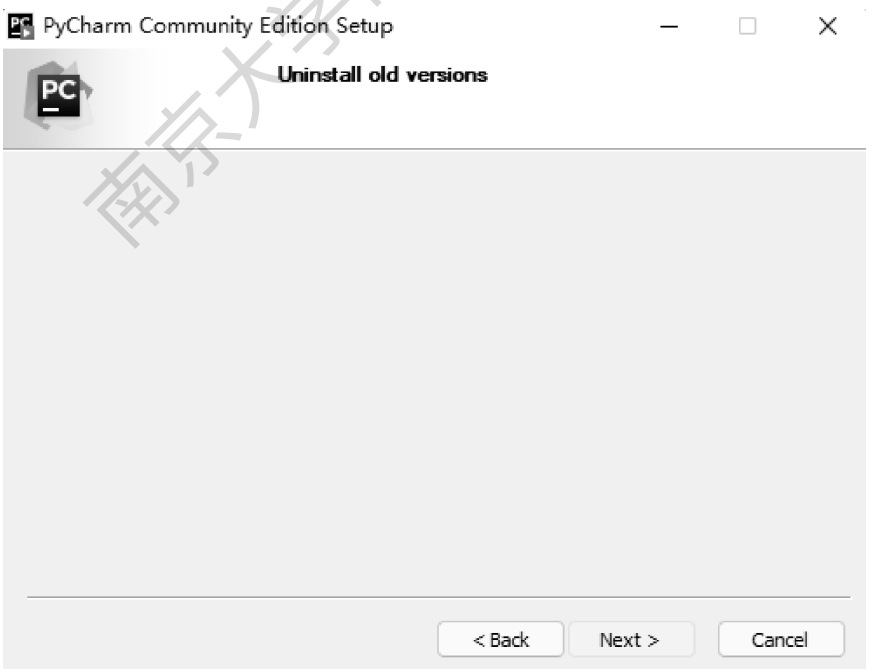

图 1.27　卸载完成

1.3.4 PyCharm的常用设置

1. 设置运行参数

当需要给程序指定输入参数时，可以单击“Run”菜单下的“Edit Configurations…”命令。在 Script parameters 处添加所需的参数。

2. 设置背景图片

按两下 Shift 键，出现搜索框，输入“set background”，找到“Set Background Image”。双击进入，选择想要的图片，然后单击“OK”保存。

3. 设置字体大小和快捷键

字体大小设置：可以通过 File→Settings→Editor→Font 来更改字体大小。

快捷键设置如下。

放大字体：File→Settings→Keymap，在搜索框中输入“increase”，找到“Increase Font Size”，然后设置相应的快捷键，如 Ctrl＋向上滑轮。

缩小字体：同样的路径下，搜索“decrease”，找到“Decrease Font Size”，然后设置相应的快捷键，如 Ctrl＋向下滑轮。

4. 设置 Python 自动引入包

File→Settings→Editor→General→Auto import，然后勾选“Python：show import popup”。

5. 设置编辑器颜色和主题

File→Settings→Editor→Colors Scheme，可以选择不同的颜色主题，如“Darcula”。

6. 设置 Python 文件默认编码

File→Settings→Editor→File Encodings，将 Global Encoding 和 Project Encoding 都设置为“UTF－8”。

7. 代码风格设置

在 Settings 的 Editor 下的 Code Style 中，可以设置代码风格，包括缩进、空格、换行等。

上述设置可以帮助个性化 PyCharm 的开发环境，提高编程效率。注意，不同版本的 PyCharm 可能在界面和设置路径上有所差异，建议参考具体版本的官方文档进行设置。

任务 1.4　模块的安装、导入与使用

1.4.1　Python 模块的安装、导入与使用

在 Python 中，模块是一种组织代码的方式，它允许将相关的函数、类和变量封装在一起，并在其他 Python 程序中重用。下面将介绍 Python 模块的导入、安装和使用。

1. 安装模块

在 Python 中，可以使用 pip 工具来安装模块。pip 是 Python 的包管理器，它允许安装、升级和卸载 Python 包（即模块）。

要安装一个模块，打开命令行终端（Windows 系统上为命令提示符或 PowerShell，macOS 和 Linux 系统上为 Terminal），然后输入以下命令。

```bash
pip install module_name
```

其中 module_name 是安装的模块的名称。例如，要安装 requests 模块，可以运行如下代码。

```bash
pip install requests
```

如果使用的是 Python 3，并且系统中同时安装了 Python 2 和 Python 3，可能需要使用 pip3 代替 pip。

```bash
pip3 install requests
```

2. 导入模块

在 Python 脚本中，可以使用 import 语句来导入模块。有以下几种导入模块的方式。

（1）导入整个模块

```python
import module_name
```

然后可以使用 module_name 来访问模块中的函数、类或变量。

(2) 导入模块并赋予别名

```python
import module_name as alias_name
```

允许使用较短的别名来引用模块。

(3) 仅导入模块中的特定部分

```python
from module_name import function_name, class_name
```

或者：

```python
from module_name import *
```

使用 *会导入模块中的所有公共对象，但通常不推荐这样做，因为它可能会导致命名空间的污染。

3. 使用模块

一旦模块被导入，就可以在 Python 脚本中使用它提供的函数、类和变量。例如，如果导入了 requests 模块，可以这样使用它：

```python
import requests

response = requests.get('https://api.example.com/data')
print(response.json())
```

在上面的例子中，导入了 requests 模块，并使用它的 get 函数发送了一个 HTTP GET 请求。然后，打印响应的 JSON 内容。

注意，每次导入一个模块时，Python 都会尝试查找该模块。它首先在当前目录的 __init__.py 文件中查找，然后在 Python 的 sys.path 列表中列出的所有目录中查找。通常，这包括 Python 的安装目录和当前工作目录。

通过遵循上述步骤，就能够在 Python 中成功地安装、导入和使用模块。

1.4.2　PyCharm 模块的安装、导入与使用

在 PyCharm 中安装、导入和使用模块，可以按照以下步骤进行。

1. 安装模块

首先，需要确定要安装的模块名称。以 requests 和 openpyxl 为例。

在 PyCharm 的界面，单击 File，然后选择 Settings（在 macOS 系统上是 PyCharm→Preferences）。

在左侧导航栏中，选择 Project：[Your Project Name]→Python Interpreter。

在这里，会看到已经安装的模块列表。要安装新的模块，点击右下角的＋按钮。

在搜索框中，输入想要安装的模块名称（如 requests 或 openpyxl），然后按回车键进行搜索。

在搜索结果中找到模块后，单击它旁边的 Install Package 按钮进行安装。安装完成后，模块将出现在已安装的模块列表中。

2. 导入模块

在 Python 代码文件中，可以使用 import 语句来导入模块。例如，要导入 requests 模块，可以在代码的开头添加 import requests。

对于 openpyxl 这样的库，可能需要导入特定的部分或全部内容。例如，from openpyxl import Workbook 将导入 Workbook 类。

3. 使用模块

一旦模块被导入，就可以在代码中使用它了。例如，使用 requests 库来发送 HTTP 请求，或使用 openpyxl 库来读取和写入 Excel 文件。

注意，当安装新的模块时，PyCharm 会自动下载并安装它们。这些模块通常会被安装在 PyCharm 的虚拟环境中，以确保项目依赖的一致性和隔离性。

另外，也可以通过命令行模式来安装模块。在 PyCharm 的界面中，按 Alt＋F12 键可以调出命令行窗口，然后可以输入 pip install module_name 来安装模块。例如，pip install requests 将安装 requests 模块。

小　结

本模块首先从 Python 的发展历程和语言特点两个方面简单介绍了 Python，然后介绍 Python 和 PyCharm 的安装与使用方法，最后介绍了模块的安装、导入与使用。通过学习本模块的内容，读者能对 Python 语言有简单的认识，能够熟练搭建 Python 开发环境，建议大家在初学 Python 时，能按照老师的操作步骤，在实验室和个人电脑上进行多次的安装、卸载及环境配置的操作以加深印象。

习　题

一、填空题

1. Python 是面向_____的高级语言。
2. Python 模块的本质是_____文件。
3. Python 可以在多种平台运行，这体现了 Python 语言_____的特性。
4. 使用_____关键字可以在当前程序中导入模块。
5. 使用_____语句可以将指定模块中的全部内容导入当前程序。

二、判断题

1. 相比C++ 程序，Python 程序的代码更加简洁、语法更加优美，但效率较低。（　　）
2. Python 3.x 版本完全兼容 Python 2.x。（　　）
3. “from 模块名 import *”语句与“import 模块名”都能导入指定模块的全部内容，相比之下，“from 模块名 import *”导入的内容无须指定模块名，可直接调用，使用更加方便，因此更推荐在程序中以此种方式导入指定模块的全部内容。（　　）
4. PyCharm 是 Python 的集成开发环境。（　　）
5. 模块文件的后缀名必定是.py。（　　）

三、选择题

1. 下列关于 Python 的说法中，错误的是（　　）。
A. Python是从 ABC 发展起来的
B. Python 是一门高级计算机语言
C. Python 只能编写面向对象的程序
D. Python 程序的效率比 C 程序的效率低
2. 下列哪个不是 Python 的应用领域（　　）。
A. Web 开发　　B. 科学计算
C. 游戏开发　　D. 操作系统管理
3. 下列选项中，不是 Python 语言特点的是（　　）。
A. 简洁　　B. 开源
C. 面向过程　　D. 可移植

四、简答题

1. 简述 Python 的特点。
2. 简述 Python 中模块、包和库的意义
3. 简单介绍如何导入与使用模块。

模块 2

Python 基础语法

知识目标

1. 掌握 Python 中的变量和变量类型
2. 掌握 Python 中的标识符，能准确判断标识符的合法性
3. 了解 Python 中的关键字，会借助工具查看关键字信息
4. 了解不同运算符的作用，会进行不同的数值运算

技能要求

1. 能够使用 Python 中的变量和变量类型
2. 能够准确判断 Python 中的标识符的合法性

变量如思想的容器，数据类型是知识分类的智慧，运算符与控制结构则是逻辑思维的精密齿轮。每个语法规则都像社会运行的法则，既需严谨遵循，又能灵活创新。这不仅是代码世界的“语法课”，更是培养我们规则意识与创新精神的“思政课”。让我们在编写第一行代码时，就树立起用技术解决问题的责任感，让基础语法成为改变世界的钥匙，做既有扎实功底又具人文情怀的数字时代先锋！

任务 2.1　基本语法

【微信扫码】
Python 语言规范

Python 中的语法与我们日常生活中的语法虽然都属于“语法”的范畴，但两者之间存在显著的差异。下面将从以下几个方面对这两者进行比较。

1. 定义与用途

(1) Python 中的语法

① 定义：Python 中的语法是指 Python 编程语言中用于定义程序结构和功能的规则。

② 用途：Python 中的语法是编写 Python 程序的基础，它规定了如何组织代码、定义变

量、控制程序流程等。

(2) 日常生活中的语法

① 定义:日常生活中的语法通常指的是自然语言(如汉语、英语等)中的语法规则。

② 用途:自然语言语法是人们交流思想、表达情感的工具,它规定了语言的组织结构、句子成分、语序等。

2. 结构与规则

(1) Python 中的语法

① 严格的结构要求:Python 的语法要求代码具有明确的结构,如缩进、代码块、函数定义等。

② 明确的规则:Python 的语法规则明确,如变量命名规则、数据类型、运算符优先级等。

③ 面向对象的特性:Python 支持面向对象编程,具有类、对象、继承、多态等概念。

(2) 日常生活中的语法

① 较为灵活的结构:自然语言的结构相对灵活,句子成分和语序可以有多种变化。

② 复杂的规则系统:自然语言的语法规则复杂且多变,包括词汇、短语、句子结构等多个层面。

③ 语境依赖性:自然语言的语法和意义往往依赖于特定的语境和背景。

3. 表达方式与特点

(1) Python 中的语法

① 精确的表达方式:Python 中的语法通过精确的代码和符号来表达程序逻辑。

② 高效性:Python 中的语法允许程序员以高效的方式编写和执行程序。

③ 抽象性:Python 中的语法通过抽象的概念(如变量、函数、类等)来组织代码。

(2) 日常生活中的语法

① 丰富的表达方式:自然语言具有丰富的词汇和表达方式,可以表达复杂的思想和情感。

② 模糊性:自然语言在表达上具有一定的模糊性和歧义性,需要依靠语境和背景来理解。

③ 直观性:自然语言更贴近人们的日常生活和经验,因此具有更高的直观性和易理解性。

4. 应用场景与功能

(1) Python 中的语法

① 应用于软件开发、数据分析、人工智能等领域。

② 实现程序自动化、数据处理、算法实现等功能。

(2) 日常生活中的语法

① 应用于人际交往、文化传播、教育等领域。

② 实现思想交流、情感表达、文化传承等功能。

Python 中的语法与我们日常生活中的语法在定义、结构、规则、表达方式、特点以及应

用场景等方面都存在显著的差异。Python 语法是编程领域的专业术语和规则，而日常生活中的语法则是自然语言交流的基础。

2.1.1　注释

在 Python 中，注释是用来解释代码的功能、目的或任何其他相关信息的文本。Python 解释器不会执行注释，它们主要用于提高代码的可读性和可维护性。Python 中有两种主要类型的注释。

1. 单行注释

单行注释以 # 符号开头，并且从 # 开始到该行结束的所有内容都会被 Python 解释器忽略。

```
#  这是一个单行注释
print("Hello, World!")  #  这也是一个单行注释
```

2. 多行注释

Python 本身并没有提供专门的多行注释语法，但可以通过三个单引号'''或三个双引号"""来创建多行字符串，这些字符串通常被用作多行注释。虽然它们实际上是字符串字面量，但如果不在代码中使用它们，它们就可以被当作注释来用。

```
'''
这是一个
多行注释
'''

print("Hello, World!")

"""
这也是一个
多行注释
尽管它们实际上是字符串字面量
"""
```

需要注意的是，尽管多行字符串经常被用作注释，但它们并不是真正的多行注释语法。这意味着如果你在一个函数或类的定义中使用它们，它们实际上会创建字符串对象，并可能导致不期望的行为。

在实际编程中，通常推荐使用单行注释来注释单行代码或简短的功能描述，而使用多行注释来注释函数、类或更长的代码块。编写清晰、有意义的注释是提高代码质量和可维护性的重要步骤。

2.1.2 行与缩进

在 Python 中，行和缩进是非常重要的概念，它们不仅影响代码的可读性，还直接影响代码的逻辑结构和执行流程。下面将详细解释 Python 中的行和缩进规则。

1. 行

(1) 代码行长度

Python 的官方风格指南 PEP 8 建议每行代码不应超过 79 个字符。这有助于保持代码的可读性，使得在较小的显示设备上查看代码时也能保持清晰。

(2) 换行符

Python 使用换行符(\n)来表示一行的结束。在 Windows 上，换行符通常是\r\n，而在 Unix 和 Linux 上是\n。Python 脚本在不同的操作系统上运行时，会自动处理这些换行符的差异。

(3) 续行

如果一行代码太长，需要分成多行来书写，可以使用圆括号、方括号或花括号内的反斜杠(\)来表示代码行的延续。不过，在圆括号、方括号或花括号外部，反斜杠是无效的。

```
result = 1 + 2 + 3 + \
         4 + 5 + 6

my_list = [
    1, 2, 3,
    4, 5, 6
]
```

2. 缩进

(1) 块结构

Python 使用缩进来定义代码块。缩进相同的行属于同一个代码块。这种通过缩进组织代码块的方式是 Python 语法的一部分，而在其他许多编程语言中，代码块是通过大括号{}来定义的。

(2) 强制性

在 Python 中，缩进是强制性的，而不是可选的。这意味着如果忘记缩进，或者缩进不正确，代码将无法正常运行，并会抛出 IndentationError 错误。

(3) 一致性

PEP 8 建议在整个项目中保持一致的缩进风格。通常，每个缩进级别使用 4 个空格。虽然也可以使用制表符(Tab)进行缩进，但不建议这样做，因为它可能会导致在不同编辑器

或 IDE 之间显示不一致。

（4）逻辑结构

缩进不仅用于表示代码块的开始和结束，还用于表示代码之间的逻辑关系。例如，在循环、条件语句和函数定义中，缩进用于表明哪些语句属于这些结构。

```
if x > 0:
    print("x 是正数")
else:
    print("x 不是正数")
```

在上面的例子中，print 语句通过缩进表明它们分别属于 if 和 else 代码块。

（5）自动管理

许多现代文本编辑器和 IDE（如 PyCharm、VS Code 等）都提供了自动缩进管理功能，可以帮助开发者保持代码的一致性和可读性。

通过遵循 Python 的缩进规则，可以编写出结构清晰、易于理解的代码，这对于团队合作和代码维护非常重要。

2.1.3　语句换行

在 Python 中，如果想要将一个语句分成多行来书写，有几种方法可以实现。这不仅可以提高代码的可读性，还有助于遵守 PEP 8 风格指南中关于每行不超过 79 个字符的建议。以下是一些在 Python 中换行语句的方法。

1. 使用圆括号、方括号或花括号

当在圆括号、方括号或花括号内部编写代码时，可以自然地将语句延续到下一行，而无需使用任何特殊的符号。

```
result = (1 + 2 + 3 +
              4 + 5 + 6)

my_list = [
    1, 2, 3,
    4, 5, 6
]

my_dict = {
    'key1': 'value1',
    'key2': 'value2',
    'key3': 'value3',
}
```

2. 使用反斜杠(\)

在圆括号、方括号或花括号外部,可以使用反斜杠(\)来将语句延续到下一行。注意,在圆括号、方括号或花括号内部,反斜杠是无效的,并且不需要使用。

```
result = 1 + 2 + 3 + \
           4 + 5 + 6

#  注意:在列表、字典或函数调用中不需要使用反斜杠
#  下面这种方式是不正确的
#  my_list = [1, 2, 3, \
#                 4, 5, 6]
```

3. 使用括号来分组

即使不在圆括号、方括号或花括号内部,也可以通过添加额外的圆括号来将表达式分成多行。这通常用于提高可读性,尤其是在复杂的表达式中。

```
result = (1 + 2 + 3
         + 4 + 5 + 6)
```

4. 自动换行

许多文本编辑器和 IDE(如 PyCharm、VS Code 等)都支持自动换行功能,可以根据设置的行宽自动将长语句拆分成多行。这可以帮助遵守 PEP 8 风格指南,并减少手动换行的需要。

5. 字符串换行

对于多行字符串,可以使用三重引号('''或""")来定义,字符串内容会按原样保留换行符。

```
multi_line_string = """This is a
multiline string."""
```

在编写多行语句时,应确保缩进是一致的,并且遵循 PEP 8 风格指南的建议,以提高代码的可读性和可维护性。

任务 2.2　标识符和关键字

2.2.1　标识符

在 Python 中，标识符（identifier）是用来标识变量、函数、类、模块或其他对象的名称。选择清晰、具有描述性且符合命名约定的标识符对于编写易于理解和维护的 Python 代码非常重要。Python 标识符的命名规则如下。

（1）字母、数字和下划线：标识符可以包含字母（大写和小写）、数字和下划线。

（2）不能以数字开头：标识符的第一个字符不能是数字，必须是字母或下划线。

（3）区分大小写：Python 是区分大小写的，因此 myVariable 和 myvariable 是两个不同的标识符。

（4）不能使用关键字：标识符不能是 Python 的保留关键字，如 if、else、for、while 等。

（5）避免与内置函数和类冲突：尽量不要使用 Python 的内置函数和类的名称作为标识符，以免产生命名冲突。

（6）命名约定

① 常量：通常全部大写，多个单词之间用下划线分隔，如 MAX_SPEED。

② 类名：通常首字母大写，多个单词之间不分隔，使用驼峰命名法，如 MyClass。

③ 函数和方法名：全部小写，多个单词之间用下划线分隔，如 my_function。

④ 变量名：同函数和方法名，也可以使用下划线分隔单词，如 my_variable。

（7）特殊标识符

① 单下划线前缀：如_variable，通常表示该变量是“受保护的”或“私有的”，不应在模块外部直接使用。

② 双下划线前缀：如__variable，表示该变量是类的私有成员，Python 会对其名称进行“名称修饰”以防止外部直接访问。

③ 双下划线前缀和后缀：如__init__，是 Python 中的特殊方法，用于初始化对象等。

（8）国际性：在 Python 3 中，可以使用非 ASCII 字符作为标识符的一部分，但通常不推荐，因为可能导致跨平台或跨环境的问题。

（9）简洁性：尽量保持标识符简短且具有描述性。过长的标识符会使代码难以阅读和理解。

遵循上述规则可以使 Python 代码更加清晰、易于理解和维护。

下面是一些有效的 Python 标识符示例。

```
_private_var
__double_underscore_var
my_variable_1
ClassName
function_name
```

下面是一些无效的 Python 标识符示例。

```
123_variable  # 以数字开头

my-variable   # 包含连字符

class         # 保留字
```

2.2.2 关键字

Python 中的关键字(或称为保留字)是 Python 语言中具有特殊含义的预定义标识符。这些关键字不能用作变量名、函数名、类名或其他标识符的名称。以下是 Python 3.x 版本中的全部关键字列表(注意,随着 Python 版本的更新,这个列表可能会有所变化)。

```
import keyword
print(keyword.kwlist)
```

执行上述代码将输出 Python 3 当前版本的所有关键字。以下是一个 Python 3.x 版本的关键字列表。

['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']

这个列表包括了从 Python 的早期版本到 Python 3.x 的所有关键字。一些关键字(如 async 和 await)是在 Python 3.7 及更高版本中引入的,用于支持异步编程。

关键字的作用是定义 Python 语言的语法和语义,它们有特殊的含义和用途。例如,def 关键字用于定义函数,if 关键字用于条件语句,class 关键字用于定义类,等等。

关键字在 Python 中具有特殊含义,因此不应将它们用作变量名、函数名或其他标识符。尝试使用关键字作为标识符将导致语法错误。

在 Python 中,可以使用 keyword 模块来查看关键字列表。这个模块提供了两个主要的函数:keyword.kwlist 和 keyword.iskeyword(word)。

keyword.kwlist 函数返回当前 Python 版本的所有关键字列表。这些关键字是 Python 语言中具有特殊含义的预定义标识符。

keyword.iskeyword(word)函数接受一个字符串参数 word,并返回一个布尔值,指示该字符串是否是一个关键字。

以下是如何使用这两个函数来查看关键字列表及其说明的示例。

```
import keyword

# 查看所有关键字列表
print("Python 关键字列表:")
```

```
print(keyword.kwlist)

#  查看关键字说明(这里没有直接的函数来获取说明,但可以通过文档或其他资源查找)
print("\nPython 关键字说明:")
for keyword in keyword.kwlist:
    #  这里使用 help 函数来获取每个关键字的简短说明
    #  注意:help 函数在某些情况下可能无法提供详细的说明
    print(f"{keyword}: {help(keyword)}")
```

上述代码将输出 Python 的所有关键字,并尝试使用 help 函数来获取每个关键字的简短说明。然而,需要注意的是,help 函数并不总是能提供关键字的详细说明,特别是对于那些没有附加文档字符串的关键字。

任务 2.3　变量和数据类型

【微信扫码】
数据类型与变量定义

2.3.1　变量和赋值

在 Python 中,变量是一种用于存储数据的容器。它们可以是整数、浮点数、字符串、列表、元组、字典等类型,或者是这些类型的组合。变量允许在程序执行期间存储和引用数据。

赋值是 Python 中的一个基本操作,它用于将一个值(可以是任何数据类型)存储在一个变量中。在 Python 中,变量在第一次赋值时会被创建,并且可以随时改变其存储的值或类型。

以下是变量和赋值的详细解释。

1. 变量

变量是一个标识符,用于存储数据值。在 Python 中,变量不需要事先声明其类型,因为 Python 是一种动态类型语言。变量的类型是在赋值时根据所赋值的对象自动确定的。例如:

```
#  创建一个整数类型的变量
age = 30

#  创建一个字符串类型的变量
name = "Alice"

#  创建一个列表类型的变量
fruits = ["apple", "banana", "cherry"]
```

在上面的例子中，age、name 和 fruits 都是变量，它们分别存储了整数、字符串和列表类型的值。

2. 赋值

赋值是将一个值（可以是任何类型的数据）与变量名关联起来的过程。在 Python 中，赋值操作符是 = 。使用 = 操作符可以将一个值赋给一个变量。例如：

```
# 将整数 30 赋给变量 age
age = 30

# 将字符串"Alice"赋给变量 name
name = "Alice"

# 将列表["apple", "banana", "cherry"]赋给变量 fruits
fruits = ["apple", "banana", "cherry"]
```

每次使用赋值操作符（=）时，都会创建一个新的变量（如果它之前不存在）或将现有变量的值更新为新赋的值。

3. 变量和赋值的关系

变量和赋值是紧密相关的。变量提供了一个存储数据的容器，而赋值操作则负责将数据放入这个容器中。通过赋值，我们可以改变变量的值或类型，或者创建新的变量。例如：

```
# 创建一个变量并赋值
x = 5

# 改变变量的值
x = 10

# 创建一个新的变量并赋值
y = "Hello, World!"

# 变量可以引用其他类型的值
z = [x, y]  # z 现在是一个列表，包含整数和字符串
```

在这个例子中，我们首先创建了一个变量 x 并将其赋值为 5。然后，我们改变了 x 的值，将其赋为 10。接着，我们创建了一个新的变量 y 并将其赋值为一个字符串。最后，我们创建了一个新的变量 z，它是一个列表，包含了整数和字符串类型的值。

总之，变量和赋值是 Python 编程中非常重要的概念，它们允许存储和操作程序执行期间需要使用的数据。

2.3.2　变量的类型

在 Python 中，变量的类型（或称为数据类型）决定了变量可以存储哪种类型的数据。Python 是一种动态类型语言，这意味着不需要在声明变量时指定其类型，Python 会根据赋给变量的值自动推断其类型。以下是一些 Python 中常见的变量类型。

（1）整型（integer）：正或负整数，不带小数点。例如：100，-786，0。

（2）浮点型（floating point）：带有小数点的数字。例如：15.20，0.0，-21.9，32.3 + e18。

（3）复数型（complex）：包含实部和虚部的数字。例如：3.14j，45.j，9.322e - 36j。

（4）布尔型（boolean）：有两个值，True 或 False。常用于条件判断。

（5）字符串型（string）：由零个或多个字符组成的有序字符序列。例如：'hello'，"world"。

（6）列表（list）：有序的集合，可以包含不同类型的数据项。使用方括号[]表示。例如：[1，'a'，2.3]。

（7）元组（tuple）：与列表类似，但元组是不可变的，即不能修改其内容。使用圆括号()表示。例如：(1，'a'，2.3)。

（8）集合（set）：无序且不包含重复元素的集合。使用大括号{}（注意，空集合必须用 set()表示，因为{}用于表示空字典）。例如：{1，2，3}。

（9）字典（dictionary）：无序的键值对集合。使用大括号{}表示，键和值之间用冒号：分隔。例如：{'name'：'Alice'，'age'：25}。

在 Python 中，可以使用 type()函数来查看变量的类型，或者使用 isinstance()函数来检查变量是否属于某个特定的类型。

```
x = 10
print(type(x))  # 输出: <class 'int'>

y = "Hello"
print(type(y))  # 输出: <class 'str'>

z = [1, 2, 3]
print(type(z))  # 输出: <class 'list'>

print(isinstance(x, int))  # 输出: True
print(isinstance(y, str))  # 输出: True
print(isinstance(z, list))  # 输出: True
```

通过这些类型，Python 提供了灵活的数据结构来处理各种编程任务。

任务 2.4　数字类型

Python 中的简单数值类型主要包括整数(integer)、浮点数(floating point)和复数(complex)。这些类型用于存储和操作数值数据。

2.4.1　整数

整数是没有小数部分的数字,可以是正数、负数或零。在 Python 中,整数类型没有固定的上限或下限,它的大小只受限于可用内存。

```
x = 100  # 正整数
y = -786  # 负整数
z = 0  # 零
```

2.4.2　浮点数

浮点数是有小数部分的数字。它们用于表示实数,即小数或分数的数字。浮点数类型在 Python 中用于执行涉及小数点的数学运算。

```
pi = 3.14159  # 正浮点数
negative_pi = -2.71828  # 负浮点数
zero_float = 0.0  # 零浮点数
```

2.4.3　复数

复数是包含实部和虚部的数字,通常用于表示数学和工程中的复杂数值。在 Python 中,复数类型用于执行涉及复数的数学运算。

```
c = 1j  # 虚数单位
d = 3.14j  # 带有实部的复数
e = 45.j  # 带有虚部的复数
```

可以使用 type() 函数来检查变量的类型。

```
print(type(100))        # 输出: <class 'int'>
print(type(3.14159))    # 输出: <class 'float'>
print(type(1j))         # 输出: <class 'complex'>
```

2.4.4　数字类型转换

在 Python 中，可以使用内置的函数来转换数字类型。以下是一些常见的数字类型转换。

(1) int()：将数字或具有数字值的字符串转换为整数。

```
num = int(42.9)  # 结果是 42
num_str = "123"
num = int(num_str)  # 结果是 123
```

(2) float()：将数字或字符串转换为浮点数。

```
num = float(42)  # 结果是 42.0
num_str = "3.14"
num = float(num_str)  # 结果是 3.14
```

(3) complex()：将两个数字或字符串转换为复数。

```
num = complex(42, 24)  # 结果是 (42 + 24j)
num_str = "3.14j"
num = complex(num_str)  # 结果是 (0 + 3.14j)
```

需要注意的是，当尝试将一个不能转换为数字的字符串传递给这些函数时，Python 会抛出一个 ValueError。例如，int("abc")会引发一个错误，因为"abc"不能转换为整数。

此外，Python 还提供了 bin()，oct()和 hex()函数，这些函数可以将整数转换为二进制、八进制和十六进制的字符串表示。

```
num = 10
bin_str = bin(num)  # 结果是'0b1010'
oct_str = oct(num)  # 结果是'0o12'
hex_str = hex(num)  # 结果是'0xa'
```


任务 2.5　运算符

【微信扫码】
运算符与表达式

Python 支持多种类型的运算符，包括算术运算符、比较(关系)运算符、赋值运算符、逻辑运算符、位运算符、成员运算符和身份运算符。以下对这些运算符作简要介绍。

2.5.1　算术运算符

算术运算符用于执行基本的数学运算，如加(+)、减(-)、乘(*)、除(/)、取模(%)、指数

运算(**)和整除(//)。

(1) a + b:加法。

(2) a - b:减法。

(3) a *b:乘法。

(4) a / b:除法。

(5) a % b:取模(余数)。

(6) a **b:指数运算。

(7) a // b:整除(向下取整)。

算术运算符用于执行基本的数学运算,如加法、减法、乘法、除法等。以下是常见的算术运算符及其用法和案例。

```
a = 5
b = 3
result = a + b
print(result)  # 输出: 8

a = 5
b = 3
result = a - b
print(result)  # 输出: 2

a = 5
b = 3
result = a *b
print(result)  # 输出: 15

a = 5
b = 3
result = a / b

print(result)  # 输出: 1.6666666666666667

a = 5
b = 3
result = a // b
print(result)  # 输出: 1

a = 5
b = 3
result = a % b
print(result)  # 输出: 2
```

```
a = 2
b = 3
result = a **b
print(result)  # 输出: 8

a = - 5
result = - a
print(result)  # 输出: 5
```

2.5.2　比较(关系)运算符

比较(关系)运算符用于比较两个值的大小关系,如等于(==)、不等于(!=)、大于(>)、小于(<)、大于等于(>=)和小于等于(<=)。

(1) a == b:检查 a 是否等于 b。

(2) a != b:检查 a 是否不等于 b。

(3) a> b:检查 a 是否大于 b。

(4) a <b:检查 a 是否小于 b。

(5) a>= b:检查 a 是否大于或等于 b。

(6) a<= b:检查 a 是否小于或等于 b。

上述运算符通常用于条件判断和控制流语句(如 if 语句)。以下是 Python 中的比较运算符及其用法和案例。

```
a = 5
b = 5
result = a == b
print(result)  # 输出: True

a = 5
b = 3
result = a != b
print(result)  # 输出: True

a = 5
b = 3
result = a > b
print(result)  # 输出: True
```

2.5.3　赋值运算符

赋值运算符用于将值赋给变量,如等于(=)、加等于(+=)、减等于(-=)、乘等于(*=)、除等

于(/=)、取模等于(%=)和指数等于(**=)。

(1) a = b:将 b 的值赋给 a。

(2) a += b:将 a 和 b 的和赋给 a。

(3) a -= b:将 a 减去 b 的结果赋给 a。

```
# 初始化变量
a = 10
b = 5

# 使用加法赋值运算符
a += b  # 相当于 a = a + b
print(f"After a += b, a = {a}")  # 输出: After a += b, a = 15

# 使用减法赋值运算符
a -= b  # 相当于 a = a - b
print(f"After a -= b, a = {a}")  # 输出: After a -= b, a = 10

# 使用乘法赋值运算符
a *= b  # 相当于 a = a * b
print(f"After a *= b, a = {a}")  # 输出: After a *= b, a = 50

# 使用除法赋值运算符
a /= b  # 相当于 a = a / b
print(f"After a /= b, a = {a}")  # 输出: After a /= b, a = 10.0(注意,结果是浮点数)
```

2.5.4 逻辑运算符

逻辑运算符用于组合或修改条件语句的结果,如与(and)、或(or)和非(not)。

(1) a and b:如果 a 和 b 都为真,则结果为真。

(2) a or b:如果 a 或 b 至少有一个为真,则结果为真。

(3) not a:如果 a 为假,则结果为真;如果 a 为真,则结果为假。

2.5.5 位运算符

位运算符用于执行二进制位级别的运算,如与(&)、或(|)、异或(^)、取反(~)、左移(<<)和右移(>>)。

(1) a & b:对 a 和 b 的二进制表示进行按位与运算。

(2) a | b:对 a 和 b 的二进制表示进行按位或运算。

(3) a ^ b:对 a 和 b 的二进制表示进行按位异或运算。

(4) ~a:对 a 的二进制表示进行按位取反运算。

(5) a << b:将 a 的二进制表示向左移动 b 位。

(6) a >> b:将 a 的二进制表示向右移动 b 位。

2.5.6　成员运算符

成员运算符用于检查一个值是否存在于某个序列(如列表、元组或字符串)中,如 in 和 not in。

(1) a in b:检查 a 是否是 b 的一个成员。

(2) a not in b:检查 a 是否不是 b 的一个成员。

```
    # 定义一个包含水果名称的列表
    fruits = ['apple', 'banana', 'cherry', 'date']
    # 获取用户输入
    user_input = input("请输入一个水果名称: ").lower()  # 转换为小写以进行不区分大小写
的比较
    # 使用 if 语句和成员运算符检查用户输入是否在列表中
    if user_input in fruits:
        print(f"你输入的水果 '{user_input}'在列表中。")
    else:
        print(f"你输入的水果 '{user_input}'不在列表中。")

    # 定义一个包含数字的集合
    numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    # 定义要检查的数字
    number_to_check = 7
    # 使用 if 语句和成员运算符检查数字是否在集合中
    if number_to_check in numbers:
        print(f"数字 {number_to_check} 在集合中。")
    else:
        print(f"数字 {number_to_check} 不在集合中。")
```

上述案例展示了如何使用 if 语句配合 Python 的成员运算符(in 和 not in)来检查值是否存在于不同的数据结构(如列表、字符串和集合)中。这种检查通常用于条件判断,以决定程序接下来应该执行哪些操作。

2.5.7　身份运算符

身份运算符用于检查两个变量是否引用同一个对象,如 is 和 is not。

(1) a is b:如果 a 和 b 引用同一个对象,则结果为真。

(2) a is not b:如果 a 和 b 不引用同一个对象,则结果为真。

运算符的优先级决定了在表达式中先执行哪个运算符。在 Python 中,通常遵循算术运算符、比较运算符、赋值运算符、逻辑运算符和位运算符的顺序,其中括号可以用来改变运算顺序。

【微信扫码】
程序基本编写方法

小　结

本模块主要讲解了 Python 中的基本语法、变量、数据类型和运算符，这些都是 Python 中最基础、最容易理解的知识，在学习的过程中，需要进行反复多遍的手写练习，加深印象，为后面的深入学习扎实基本功。

习　题

一、填空题

1. float()函数用于将数据转换为_____类型的数据。
2. 使用_____函数可查看数据的类型。
3. 布尔类型的取值包括_____和_____。
4. Python 中建议使用_____个空格表示一级缩进。

二、判断题

1. 布尔类型是特殊的浮点型。(　　)
2. Python 标识符不区分大小写。(　　)
3. 变量名可以以数字开头。(　　)
4. Python 中可以使用关键字作为变量名。(　　)
5. 复数类型的实数部分可以为 0。(　　)

三、选择题

1. 下列选项中，属于数值类型的是(　　)。

A. 0　　B. 1.0
C. 1+2j　　D. 以上全部

2. Python 中使用()符号表示单行注释。(　　)。

A. #　　B. /
C. //　　D. <!-- -->

3. 下列选项中，不属于 Python 关键字的是(　　)。

A. name　　B. if
C. is　　D. and

四、简答题

1. 请简单介绍 Python 中的运算符。
2. 请简单介绍 Python 中的数据类型和数字类型。

五、编程题

1. 已知某煤场有 30 吨煤，先用一辆载重 4 吨的汽车运 3 次，剩下的用一辆载重为 3 吨的汽车运送，请问还需要运送几次才能送完？编写程序，解答此问题。

2. 编写程序，要求程序能根据用户输入的数据计算圆的面积（圆的面积公式：$S=\pi r^2$，π 取值为 3.14），并分别输出圆的直径和面积。

模块 3

Python 常用语句

知识目标

1. 掌握判断语句的使用
2. 掌握循环语句的使用
3. 掌握 break、continue、pass 和 else 语句的作用

技能要求

1. 能够熟练使用 if 判断语句
2. 能够熟练使用 for、while 循环语句
3. 能够熟练使用 break、continue、pass 和 else 语句

输入输出是程序与外界对话的桥梁，教会我们开放包容的合作态度；条件判断如同人生岔路口的选择，锤炼着我们的决策智慧；循环结构则是效率革命的引擎，让我们领悟积跬步以至千里的坚持精神。这些语句不仅是代码世界的"工具箱"，更是培养我们辩证思维与责任担当的"思政课"。让我们在编写 if-else、while 循环时，既掌握改变世界的代码武器，更铸就服务社会的赤子之心，成为兼具技术硬实力与人文软实力的新时代创新者！

任务 3.1　判断语句

【微信扫码】
条件分支语句

在 Python 中，判断语句（也称为条件语句）用于根据特定条件执行不同的代码块。它们使程序能够根据条件为真（True）或假（False）来选择执行路径。Python 使用 if、elif（else if 的缩写）和 else 关键字来实现判断语句。

3.1.1　if 语句

if 语句是最基本的条件语句，它允许程序根据一个布尔表达式（条件）的真假来决定是

否执行一段代码。

语法结构：

```
if 条件:
#  当条件为真时执行的代码
```

示例如下。

```
x = 10
if  x > 5:
    print("x 大于 5")
```

在这个例子中，如果 x 的值大于 5，那么就会打印出"x 大于 5"。

3.1.2 if-else 语句

if-else 语句是 if 语句的扩展，它允许程序在条件为假时执行另一段代码。

语法结构：

```
if 条件:
    #  当条件为真时执行的代码
else:
    #  当条件为假时执行的代码
```

示例如下。

```
x = 3
if x > 5:
    print("x 大于 5")
else:
    print("x 不大于 5")
```

在这个例子中，如果 x 的值不大于 5，那么就会打印出"x 不大于 5"。

```
#  获取用户输入的年龄
age = int(input("请输入你的年龄:"))
#  使用 if - else 语句判断年龄是否大于 18 岁
if age > 18:
#  如果年龄大于 18 岁，打印成年信息
    print("你已经成年了。")
else:
#  如果年龄不大于 18 岁，打印未成年信息
    print("你还未成年。")
```

(1) 输入年龄

age = int(input("请输入你的年龄:")):这行代码会提示用户输入一个值,然后使用 int()函数将这个值转换成整数类型,并赋值给变量 age。

(2) 判断年龄

if age > 18::这行代码是一个条件判断语句。它检查变量 age 的值是否大于 18。

如果条件为真(即用户输入的年龄大于 18),则执行 if 语句块下的代码,打印出"你已经成年了。"。

(3) 处理其他情况

else::这行代码表示如果 if 语句中的条件不为真,则执行 else 语句块下的代码。

在这个例子中,如果用户的年龄不大于 18 岁,就会执行 else 语句块下的代码,打印出"你还未成年。"。

运行结果如下。

如果用户输入 20,程序会输出:"你已经成年了。"

如果用户输入 16,程序会输出:"你还未成年。"

这个简单的 if-else 语句示例展示了如何根据用户输入的条件(在这个例子中是年龄)来执行不同的代码块。这是编程中非常基础且常用的逻辑判断方式。

3.1.3 if-elif 语句

if-elif 语句实际上是 if-elif-else 的简化形式,因为可以有多个 elif 但没有 else 也是可行的。if-elif 语句允许程序检查多个条件,并根据第一个为真的条件执行相应的代码块。如果所有条件都不为真,并且存在 else 子句,则执行 else 子句中的代码。

语法结构:

```
if 条件 1:
    # 当条件 1 为真时执行的代码
elif 条件 2:
    # 当条件 1 为假且条件 2 为真时执行的代码
    # 可以有多个 elif 子句
else:
    # 当所有条件都为假时执行的代码(可选)
```

示例如下。

```
x = 7
if x > 10:
    print("x 大于 10")
elif x > 5:
    print("x 大于 5 但小于或等于 10")
else:
    print("x 小于或等于 5")
```

3.1.4　if 嵌套语句

在 Python 中，if 嵌套语句指的是在一个 if 语句（或 elif 语句）的代码块内部再包含一个或多个 if 语句。这种结构允许根据多个条件来执行不同的代码块，从而实现更复杂的逻辑判断。

语法结构：

```
if 条件 1:
    # 当条件 1 为真时执行的代码
    if 条件 2:
        # 当条件 1 和条件 2 都为真时执行的代码
    else:
        # 当条件 1 为真但条件 2 为假时执行的代码
else:
    # 当条件 1 为假时执行的代码
    if 条件 3:
        # 当条件 1 为假但条件 3 为真时执行的代码(这也是一种嵌套,尽管它不在最初的 if 块内)
    else:
        # 当条件 1 为假且条件 3 也为假时执行的代码(同样,这也是一种嵌套)
```

以下是一个具体的示例，用于判断一个数字是正数、负数还是零，并进一步判断正数是奇数还是偶数。

```
num = int(input("请输入一个整数:"))

if num > 0:
    print("这个数字是正数。")
    if num % 2 == 0:
        print("并且它是偶数。")
    else:
        print("并且它是奇数。")
elif num < 0:
    print("这个数字是负数。")
else:
    print("这个数字是零。")
```

运行结果如下。

如果用户输入 4，程序输出如下。

```
请输入一个整数:4
这个数字是正数。
并且它是偶数。
```

如果用户输入－3，程序输出如下。

```
请输入一个整数：- 3
这个数字是负数。
```

如果用户输入 0，程序输出如下。

```
请输入一个整数：0
这个数字是零。
```

注意事项：

（1）缩进：Python 使用缩进来定义代码块。在 if 嵌套语句中，每个 if 或 else 下面的代码块都必须正确缩进。

（2）可读性：过多的嵌套会使代码难以阅读和维护。尽量通过重构代码（如使用函数或更简洁的逻辑结构）来减少嵌套层次。

（3）逻辑清晰：确保每个 if 和 else 的条件都是清晰且互斥的，以避免逻辑错误或不必要的复杂性。

（4）避免“悬垂 else”：在嵌套 if 语句中，要注意 else 子句是与最近的未匹配的 if 子句相关联的。有时会导致逻辑错误，特别是当嵌套层次较多时。为了避免这种情况，可以使用 elif 来明确指定条件分支，或者通过适当的缩进和注释来提高代码的可读性。

任务 3.2　循环语句

【微信扫码】
循环控制语句

在 Python 编程中，循环语句用于重复执行代码块，直到满足某个条件为止。这种机制在现实生活中也有很多类似的场景，尽管它们可能不像编程语言中的循环那样精确和形式化。以下是一些生活中循环场景的示例。

1. 日常习惯

每天早晨的例行公事，如起床、洗漱、吃早餐等，可以看作是一个循环。

每周的工作日，人们通常遵循相似的日程安排，从早到晚进行各种活动。

2. 季节变化

一年四季的更迭，春、夏、秋、冬不断循环。

在农业中，播种、生长、收获和休耕的周期也可以看作是一种循环。

3. 自然界中的循环

水循环：水从海洋蒸发，形成云，然后以降水的形式返回地面，再流入河流和海洋。

碳循环：碳在生物体、大气和地壳之间不断循环。

4. 工作和学习

学期中的课程安排，每周或每天的课程表可能都是相似的，形成一个循环。
工作中的日常任务，如检查邮件、回复客户、处理数据等，可能每天都会重复。

5. 周期性事件

每月的账单支付，如房租、水电费、信用卡还款等。
每年的节日和纪念日，如春节、圣诞节、生日等，虽然具体内容可能不同，但每年都会发生。

6. 健身和锻炼

定期的健身计划，如每周三次的跑步或健身房锻炼。
日常的身体活动，如散步、做家务等，也可以看作是一种循环。

7. 社交活动

每周的聚会或社交活动，如与朋友共进晚餐、参加俱乐部会议等。
社交媒体的日常浏览和互动，如查看更新、点赞、评论等。

这些生活中的循环场景虽然不像 Python 中的循环语句那样具有严格的逻辑和控制结构，但它们都体现了重复和周期性的概念。在编程中，循环语句以一种高效和简洁的方式处理这些重复性的任务。

3.2.1　while 循环

while 循环是 Python 编程语言中的一种基本控制结构，它允许代码块根据给定的条件重复执行。以下是对 Python 中 while 循环的详细讲解。

语法结构：

```
while 条件:
    # 循环体:要重复执行的代码块
    # 可以是任意数量的语句
```

“条件”是一个表达式，它会在每次循环开始前被评估。如果“条件”为真（即它的值为 True），则循环体中的代码块会被执行。执行完循环体后，会再次评估“条件”。如果“条件”仍然为真，循环会继续。当“条件”为假（即它的值为 False）时，循环结束，程序会继续执行 while 循环之后的代码。

以下是一个简单的 while 循环示例，它打印数字 0 到 4。

```
count = 0  # 初始化计数器
# 当计数器小于 5 时,执行循环体内的代码
while count < 5:
    print(count)  # 打印当前计数器的值
    count += 1  # 计数器加 1,为下一次循环做准备
```

在这个例子中，count 初始化为 0，然后 while 循环检查 count <5 是否为真。由于初始时 count 是 0，所以条件为真，循环体中的 print(count)语句被执行，打印出 0。然后，count 被递增为 1，并再次检查条件。这个过程会一直重复，直到 count 达到 5，此时条件变为假，循环结束。

3.2.2 for 循环

for 循环语句在 Python 中是一种用于遍历序列(如列表、元组、字符串、字典、集合或任何可迭代对象)或其他可迭代对象并执行循环体中代码块的控制流语句。下面是关于 for 循环语句的详细讲解。

语法结构：

```
#  for 循环开始
for 变量 in 可迭代对象:
    #  循环体:要重复执行的代码块
        #  可以是任意数量的语句
```

“变量”在每次循环迭代时会被赋予“可迭代对象”中的下一个元素的值。“可迭代对象”可以是列表、元组、字符串、字典、集合、范围(range)对象等。循环体中的代码会对“可迭代对象”中的每个元素执行一次。

遍历列表：

```
fruits = ['apple', 'banana', 'cherry']
for fruit in fruits:
    print(fruit)
```

输出：

```
apple
banana
cherry
```

在这个例子中，for 循环遍历了列表 fruits 中的每个元素，并将每个元素的值赋给了变量 fruit，然后执行了 print(fruit)语句。

遍历字符串：

```
greeting = "Hello, world!"

for char in greeting:
    print(char)
```

输出：

```
H
e
```

```
l
l
o
,

w
o
r
l
d
!
```

在这个例子中，for循环遍历了字符串greeting中的每个字符，并将每个字符的值赋给了变量char，然后执行了print(char)语句。

使用range()函数：

```
for i in range(5):
    print(i)
```

输出：

```
0
1
2
3
4
```

在这个例子中，range(5)生成了一个从0到4的整数序列(包括0，不包括5)，for循环遍历了这个序列中的每个数字，并将每个数字的值赋给了变量i，然后执行了print(i)语句。

注意事项：

(1) 避免修改可迭代对象：在遍历可迭代对象时，通常不建议修改它(除非有明确的原因和了解可能产生的后果)。例如，在遍历列表时不要尝试删除或添加元素，因为这可能会导致未定义的行为或错误。

(2) enumerate()函数：如果需要在循环中同时获取元素的索引和值，可以使用enumerate()函数。

(3) break和continue：与while循环类似，for循环中也可以使用break语句提前退出循环，或使用continue语句跳过当前迭代并开始下一次迭代。

enumerate()示例如下。

```
fruits = ['apple', 'banana', 'cherry']

for index, fruit in enumerate(fruits):
    print(f"Index: {index}, Fruit: {fruit}")
```

输出：

```
Index: 0, Fruit: apple
Index: 1, Fruit: banana
Index: 2, Fruit: cherry
```

在这个例子中，enumerate(fruits)生成了一个包含元组的迭代器，每个元组包含两个元素：当前元素的索引和值。for 循环遍历了这个迭代器，并将每个元组的值赋给了变量 index 和 fruit，然后执行了 print 语句。

3.2.3 while 嵌套

在 Python 中，while 循环是一种基本的控制流语句，用于在特定条件为真时重复执行一段代码。while 循环可以嵌套使用，即在一个 while 循环的内部再包含另一个 while 循环。这种嵌套结构在处理多级条件或复杂逻辑时非常有用。

语法结构：

```
while 条件 1:
    # 代码块 1
    while 条件 2:
        # 代码块 2
        # 这里的代码在内部 while 循环结束后执行
```

假设要打印一个 4 行 5 列的矩形图案，每个位置都用星号(*)表示。

```
# 初始化行计数器
rows = 4
i = 1
while i <= rows:
    # 初始化列计数器
    cols = 5
    j = 1
    while j <= cols:
        # 打印星号
        print('*', end ='')
        # 列计数器递增
        j += 1
    # 打印完一行后换行
    print()
    # 行计数器递增
    i += 1
```

输出：

```
* * * * *
* * * * *
* * * * *
* * * * *
```

外层 while 循环：控制行数。变量 i 从 1 开始，直到 rows(包含 rows)，即 4。

内层 while 循环：控制列数。对于每一行，变量 j 从 1 开始，直到 cols(包含 cols)，即 5。

打印操作：在内层循环中，打印星号 *，并用空格 end =''来避免自动换行。

换行操作：内层循环结束后，使用 print()函数来换行，开始新的一行。

计数器递增：每次内层循环结束后，外层循环的计数器 i 递增，以开始新的一行。

这个简单的例子展示了如何使用嵌套的 while 循环来控制行和列，从而打印出所需的图案。

以下示例，演示如何使用嵌套的 while 循环来打印一个乘法表(如 5×5 的乘法表)。

```
# 打印 5×5 乘法表
i = 1  # 外层循环变量
while i <= 5:
    j = 1  # 内层循环变量,每次外层循环开始时重置
    while j <= 5:
        print(f"{i} * {j} = {i * j}", end ="\t")  # 打印乘法结果,并用制表符分隔
        j += 1  # 内层循环变量递增
    print()  # 内层循环结束后换行
    i += 1  # 外层循环变量递增
```

外层 while 循环：变量 i 从 1 开始，直到 5(包含 5)，用于控制乘法表的行数。

内层 while 循环：变量 j 从 1 开始，直到 5(包含 5)，用于控制每一行中的列数。

打印操作：在内层循环中，计算并打印 i 和 j 的乘积，结果用制表符 \ t 分隔，以便对齐。

变量递增：每次内层循环结束后，打印一个换行符以开始新的一行；每次外层循环结束后，递增 i 以开始新的一行计算。

输出：

```
1 * 1 = 1 1 * 2 = 2 1 * 3 = 3 1 * 4 = 4 1 * 5 = 5
2 * 1 = 2 2 * 2 = 4 2 * 3 = 6 2 * 4 = 8 2 * 5 = 10
3 * 1 = 3 3 * 2 = 6 3 * 3 = 9 3 * 4 = 12 3 * 5 = 15
4 * 1 = 4 4 * 2 = 8 4 * 3 = 12 4 * 4 = 16 4 * 5 = 20
5 * 1 = 5 5 * 2 = 10 5 * 3 = 15 5 * 4 = 20 5 * 5 = 25
```

注意事项：

(1) 嵌套层次：Python 对嵌套的层次没有硬性限制，但过深的嵌套可能会使代码难以阅读和维护。

(2) 终止条件：确保每个 while 循环都有明确的终止条件，以避免无限循环。

(3) 变量作用域:注意变量的作用域,尤其是在多层嵌套时,确保变量的正确初始化和更新。

通过使用嵌套的 while 循环,可以灵活地处理复杂的循环逻辑。在实际编程中,根据需要选择适合的循环结构来解决问题。

任务 3.3 Python 的其他语句

3.3.1 break 语句

break 语句用于立即终止当前所在的最内层循环(无论是 for 循环还是 while 循环),并跳出循环体,继续执行循环之后的代码。break 语句通常与某种条件判断一起使用,以便在满足特定条件时提前结束循环。

使用 break 语句的基本示例如下。

(1) 在 for 循环中使用 break

```
for i in range(1, 11):
    if i == 5:
        break  # 当 i 等于 5 时,终止循环
    print(i)

print("循环结束")
```

在这个例子中,循环从 1 到 10 进行迭代。当 i 等于 5 时,break 语句被执行,循环被终止,因此不会打印出 5 及之后的数字。

(2) 在 while 循环中使用 break

```
count = 0
while True:
    count += 1
    print(count)
    if count == 5:
        break  # 当 count 等于 5 时,终止循环

print("循环结束")
```

在这个例子中,while 循环会无限地进行下去,但由于 break 语句的存在,当 count 等于 5 时,循环被终止。

注意事项:

(1) 跳出多层循环:break 语句只能终止它所在的最内层循环。如果需要跳出多层循

环,可以考虑使用函数或者 flag 变量来控制。

(2) 与 continue 的区别:continue 语句会跳过当前循环的剩余部分,并继续下一次循环迭代,而 break 语句会完全终止循环。

3.3.2 continue 语句

continue 语句用于跳过当前循环的剩余部分,并立即开始下一次循环迭代。与 break 语句不同,continue 不会终止整个循环,而只是跳过当前迭代中 continue 之后的代码。它通常与某种条件判断一起使用,以便在满足特定条件时忽略当前迭代的剩余部分。

使用 continue 语句的基本示例如下。

(1) 在 for 循环中使用 continue

```
for i in range(1, 6):
    if i % 2 == 0:  # 如果 i 是偶数
        continue  # 跳过当前迭代,不执行后面的 print 语句
    print(i)  # 只打印奇数
```

在这个例子中,循环从 1 到 5 进行迭代。当 i 是偶数时(2,4),continue 语句被执行,因此 print(i)不会被执行。最终,只有奇数(1,3,5)被打印出来。

(2) 在 while 循环中使用 continue

```
count = 0
while count < 5:
    count += 1
    if count % 2 == 0:  # 如果 count 是偶数
        continue  # 跳过当前迭代,不执行后面的 print 语句
    print(count)  # 只打印奇数
```

在这个例子中,while 循环会一直执行,直到 count 达到 5。但是,当 count 是偶数时,continue 语句会跳过 print(count)语句。因此,只有奇数(1,3)被打印出来。

注意事项:

(1) 多层循环:continue 语句只能影响它所在的最内层循环。如果需要在多层循环中跳过迭代,每个循环都需要有自己的 continue 语句和条件判断。

(2) 与 break 的区别:break 语句会终止整个循环,而 continue 语句只是跳过当前迭代的剩余部分。

(3) 循环效率:使用 continue 语句可以提高循环的效率,特别是当循环体内有大量代码需要跳过时。但是,过度使用 continue 可能会使代码难以理解和维护。

3.3.3 pass 语句

pass 语句是一个空操作,也就是说,它什么也不做。它通常用作占位符或者当语法上需

要一个语句但程序逻辑上不需要任何操作时。pass 语句可以帮助保持程序结构的完整性，同时避免在编写代码时产生语法错误。

```
# 在 if 语句中使用 pass
if False:
    pass  # 当条件为 False 时,不执行任何操作

# 在 for 循环中使用 pass
for i in range(5):
    if i % 2 == 0:
        pass  # 当 i 是偶数时,不执行任何操作
    else:
        print(i)  # 当 i 是奇数时,打印 i
```

注意事项：

（1）pass 语句虽然是一个空操作，但它确实是一个有效的语句，可以被 Python 解释器识别和执行（尽管它什么也不做）。

（2）过度使用 pass 语句可能会使代码变得难以理解和维护。因此，在使用 pass 语句时，应该明确你的意图，并在后续的开发中尽快替换为实际的逻辑代码。

（3）在编写代码时，如果某个位置确实不需要任何操作，使用 pass 语句是一种简洁而清晰的方式来表达这一点。但是，如果某个函数或类完全没有实际功能，那么可能需要重新考虑其存在的必要性。

3.3.4 else 语句

else 语句通常与 if 语句一起使用，形成条件判断结构。在 Python 中，while 循环也可以使用 else 语句。这个 else 语句在循环正常结束时执行，也就是说，当循环条件不再满足（导致循环自然结束），而不是通过 break 语句跳出循环时，else 语句中的代码会被执行。

语法结构：

```
while 条件:
    # 循环体:满足条件时执行的代码
else:
    # 循环正常结束时执行的代码
```

示例如下。

```
count = 0
while count < 5:
    print(f"当前计数: {count}")
    count += 1
```

```
else:
    print("循环正常结束,计数达到 5")
```

在这个例子中,while循环会一直执行,直到count的值达到5,此时循环条件count <5不再满足,循环自然结束。随后,else子句中的代码被执行,打印出"循环正常结束,计数达到5"。

使用场景:

while循环中的else子句可以用于需要在循环正常结束时执行某些清理工作或后续处理的场景。例如,需要在成功读取完一个文件的所有内容后关闭文件,或者在成功处理完一系列数据项后发送一个确认消息。

注意事项:

(1) else子句是while循环的一个可选部分,不是必须的。

(2) else子句必须紧跟在while循环之后,并且它们之间不能有其他的语句。

(3) 如果while循环是通过break语句跳出的,那么else子句中的代码将不会被执行。

小　结

本模块深入探讨了Python编程中的核心语句,包括条件判断(if、elif、else)、循环控制(for、while及break、continue)。这些语句是构建Python程序逻辑和功能的基础,掌握它们对于编写高效、可维护的代码至关重要。通过实例解析和练习,可以熟悉这些语句的用法,为后续深入学习Python打下坚实基础。

习　题

一、填空题

1. _____语句是最简单的条件语句。
2. _____循环一般用于实现遍历循环。
3. Python中的循环语句有_____和_____循环。
4. 若循环条件的值变为_____,说明程序进入无限循环。

二、判断题

1. for循环只能遍历字符串。(　　)
2. elif可以单独使用。(　　)
3. if语句不支持嵌套使用。(　　)
4. break语句用于结束循环。(　　)
5. if-else语句可以处理多个分支条件。(　　)

三、选择题

1. 已知 x=10,y=20,z=30,以下代码执行后 x、y、z 的值分别为(　　)。

```
if x <y:
  z = x
  x = y
  y = z
```

A. 10,20,30　　B. 10,20,20

C. 20,10,10　　D. 20,10,30

2. 下列语句中,可以跳出循环结构的是(　　)。

A. continue　　B. break

C. if　　D. while

四、简答题

1. 简述 while 和 for 的区别。
2. 简述 break 和 continue 的区别。

五、编程题

1. 编写程序,实现判断用户输入的数是正数还是负数的功能。
2. 编写程序,实现利用 while 循环输出 100 以内偶数的功能。