
金融科技—Python 编程基础

第 0.61 版

周立刚 编著

二零二四年夏

目录

内容结构	1
如何使用本书	2
特别说明	2
致谢	3
1 计算机基础	4
1.1 计算的本质	4
1.2 计算机硬件基础	7
1.3 计算机软件基础	12
1.4 操作系统与文件管理	14
1.5 基本编程概念	21
1.6 Python 语言的特点	25
2 Python 安装与运行	28
2.1 安装 Python 及其开发环境	28
2.2 第一个 Python 程序	36
2.3 运行 Python 程序	37
3 状态表示 – 数据类型	49
3.1 信息形态与数据类型	49
3.2 变量与数据	51
3.3 Python 的数据类型	55
3.4 复合数据类型	72

4	状态变化控制	85
4.1	条件判断语句	85
4.2	循环语句	90
4.3	列表、字典推导与生成器表达式	101
5	程序结构	105
5.1	函数	106
5.2	类	118
5.3	模块与包	129
6	文件读写	137
6.1	文本文件的读写	138
6.2	Python 对象文件	143
6.3	Excel 文件	146
6.4	PDF 文件	150
7	异常与错误处理	154
7.1	Python 异常类型	154
7.2	异常处理机制	157
7.3	错误调试技巧	159

前言

自 2018 年秋季起，我在商学院先后为本科生开设《Python 在量化金融中的应用》，为金融专业硕士研究生讲授《金融科技》。在这段教学经历中，我发现很难为学生推荐一本既全面又合适的教材。尽管我曾精心编纂了一份资源清单，涵盖了参考书籍、在线视频课程、技术博客等多样化的经典参考材料，希望这些学习材料能够提高学生们的学习效率并拓宽他们的知识视野，但实际上太多的参考资料并没有起到预期的作用。主要原因在于，这些资料很少考虑读者可能没有计算机基础，更没有任何程序设计知识。专业术语缺乏通俗易懂的解释，内容的组织也缺乏系统的说明。

针对这一问题，为了帮助商学院同学，特别是金融专业的同学更好地掌握 Python 在金融数据分析中的应用，我决定编写此书。本书旨在为没有编程经验的读者提供一个简明扼要、循序渐进的 Python 入门指南。书中的解释力求通俗易懂，避免过多专业术语的使用，同时注重内容的逻辑组织，使读者能够循序渐进地学习。通过本书的学习，相信读者能够掌握 Python 编程的基础知识，并将其应用于金融数据分析和金融科技领域。

因为教学需要，本书在没有完全完成的情况下，分享给本学期正在学习相关课程的同学，方便他们在学习中使用。因此，本书设定为 0.6 版，是大致描述本书只完成了设想内容的 60%。

本书中同时介绍了 Windows 环境和 MacOS 下运行环境的安装和设置，主要是考虑到同学使用的电脑主要使用这两种操作系统，两种的系统使用的比例都比较高。

内容结构

本书是一本面向金融专业学生的 Python 编程入门教材，也可以为没有编程基础的读者学习 Python 编程提供参考。全书共 12 章，内容简介如下：

第 1 章介绍计算机的基础知识，为后续学习 Python 编程打下基础。第 2 章介绍 Python 的运行环境，教读者如何安装和运行 Python 程序。

第 3 章至第 5 章介绍 Python 编程的基础知识，包括数据类型、条件语句、循环语句、函数、类等概念。第 6 章和第 7 章分别介绍文件读写和异常和错误处理。

第 8 章至第 9 章介绍 Python 在数据处理和可视化方面的常用库，如 Numpy、Matplotlib、Plotly 和 Pandas。通过学习这些库，读者可以掌握数据处理、分析和可视化的基本方法。

第 10 章和第 11 章是两个实战演练章节，分别介绍如何使用 Python 构建信用风险评估模型和开发量化交易策略。这两章将前面学到的知识应用到实际问题中，帮助读者巩固所学并提高实践能力。

如何使用本书

1. 理论学习：通过阅读书中的理论部分，学习 Python 编程的基础知识和在金融领域应用的相关库。书中的解释力求通俗易懂，适合没有编程基础的读者。
2. 代码实践：每章都有对应的练习题，读者可以通过完成这些练习来巩固所学知识。
3. 实战演练：本书的最后两章提供了实际项目，读者可以通过完成这些项目来应用所学知识，提高实践能力。

特别说明

本书采用知识共享署名-非商业性使用 4.0 国际许可协议 (Creative Commons Attribution-NonCommercial 4.0 International License) 进行许可。

您可以自由地：

- 共享：在任何媒介以任何形式复制、发行本作品。
- 演绎：修改、转换或以本作品为基础进行创作。

惟须遵守下列条件：

- 署名: 您必须给出适当的署名, 提供指向本许可协议的链接, 同时标明是否(对原始作品)作了修改。
- 非商业性使用: 您不得将本作品用于商业目的。

本书在编写过程中使用了大型语言模型技术帮助提升表述质量和知识信息检索。尽管我们已努力确保内容的准确性和可靠性, 但本书中的信息可能仍存在不足或错误。因此, 本书内容仅供学习参考, 我们不对由于使用本书中信息所造成的任何形式的损失或损害承担责任。读者在依赖本书内容做出决策时, 应进行独立的验证和审慎考虑。

致谢

在这本书的编写过程中, 我深深体会到了开源社区的力量。特别要感谢那些辛勤工作的开发者们, 他们不仅创造了出色的编程工具和丰富的工具包, 还将它们无偿地公开给全世界的用户。这种开放和分享的精神极大地丰富了这本书的内容, 使我们能够站在巨人的肩膀上前行。

我也要特别感谢使用和推广这些开源工具的广大用户和开发者, 他们提出的宝贵意见和反馈帮助这些工具不断改进和完善。他们的参与确保了技术的持续进步和生态系统的繁荣。

最后, 对那些教育和推广开源精神的组织和个人表示衷心的感谢。是他们种下了合作与分享的种子, 让更多的人受益于开源技术的繁荣发展。这种精神不仅促进了学术交流, 更是推动了科技和社会的进步。

感谢这样一个充满活力与创新的社区, 让我们能够共同创造一个更加开放和交融的世界。

1

计算机基础

掌握计算机基础知识有助于理解和学习计算机编程语言。因此，本章简要的介绍计算的基本思想，以及计算机软硬件基础知识。

1.1 计算的本质

计算，从古老的算盘到现代的电子和量子计算机，其核心思想始终未变—**状态的表示以及状态变化的控制**。这一本质贯穿了各个时代的计算工具和理论，融入计算技术发展的所有历程。

1.1.1 从算盘到图灵机

算盘 (abacus)，如图 1.1 所示，是一种原始的计算工具。它通过珠子在不同位置的排列组合来表示数字状态。用户手动拨动这些珠子，按照算术规则进行状态的变换，从而实现加减乘除等算术运算。算盘的珠子和它们的位置直观地体现了“状态的表示”，用户依照运算规则对珠子位置进行改变实现了”状态变化的控制“。

图灵机 (Turing machine) 是一个理论上的计算模型，如图 1.2 所示，它是计算机科学理论上的重大突破。图灵机模型包括以下几个部分：

1. **无限长的纸带 (tape)**：纸带理论上是无限长的，被分割成连续的格子，每个格子可以记录一个符号。这些符号通常来自一个有限的字母表，例如 $\{0, 1\}$ 。

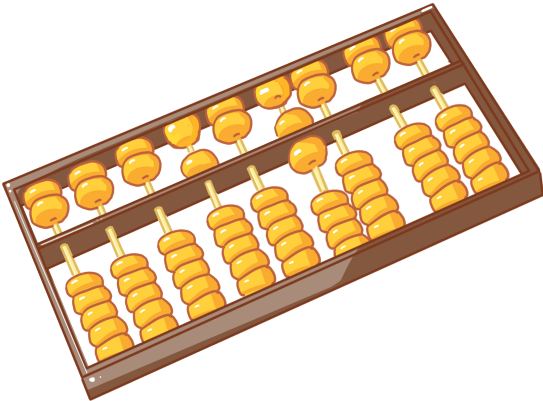


图 1.1: 算盘

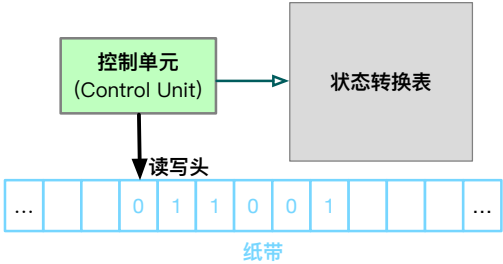


图 1.2: 图灵机模型

2. **读写头 (read/write head)**: 读写头负责读取和修改纸带上的符号, 并可以在纸带上左右移动。
3. **控制单元**: 负责管理读写头的行为, 根据当前的机器状态和读写头读到的符号决定下一步的操作, 如向左或右移动一格, 读取或改写格子中的符号。
4. **状态转换表**: 定义了在不同情况下读写头应该如何操作, 以及如何改变状态。

纸带上的符号和机器的内部状态共同记录了计算的状态。图灵机的读写头根据状态转换表控制状态的变化, 以此模拟任何算术或逻辑运算。换句话说, 图灵机通过不断读取纸带上的数据, 根据状态转换表决定如何改变状态和纸带上的符号, 来完成计算任务。

图灵机的强大之处在于其通用性。阿兰·图灵 (Alan Turing) 证明了图灵机能够执行任何可以通过**算法 (algorithm)** 实现的计算任务, 这就是著名的“**图灵完备性 (Turing completeness)**”。这意味着图灵机的计算能力与所有现有的计算机等价, 无论是传统的数字计算机还是未来可能发明的计算机。

算法是一组有序的、明确的步骤, 用于解决一个问题或执行某个任务。而将算法中的每个步骤按照某种语言的语法写成具体的、可执行的指令, 这些指令序列就构成了程序。算法是解决问题的抽象步骤和逻辑, 而程序是这些步骤在机器上的具体表述。对于图灵机, **程序 (program)** 是图灵机的状态转换表, 描述了图灵机在不同状态下的操作规则。这是静态的, 类似于一个说明书或指南。而**程序运行 (program execution)** 是图灵机按照程序规则一步步执行操作的动态过程。这是程序的实际执行, 是时间维度上的活动。

1.1.2 现代计算机: 电子、量子 and 生物计算机

晶体管 (Transistor) 是现代**电子计算机 (Electronic Computer)** 的基本构成单元, 它们可以处于开启或关闭两种状态, 这两种状态分别表示为 **1** 和 **0**, 因此它可以自然的表示二进制信息。电子计算机通过控制数以亿计的晶体管的开关状态, 以存储和表示复杂的数据和程序状态。电子计算机通过执行程序中的指令来改变晶体管的开或关状态以实现状态变化的控制。每执行一个操作, 计算机内部的寄存器、内存单元或者其他电子组件的状态就会按照指令的要求发生变化。

现代电子计算机通过使用电子元件的二进制来表示状态并使用程序指令控制状态的变化, 完美地体现了“状态的表示以及状态变化的控制”的计算本质。每一次计算都是在对数据状态进行精确的控制和转换, 从而实现复杂的数据处理和任务执行。

量子计算机 (quantum computer) 采用量子比特作为信息的基本单位, 它们可以处于 0、1 或者这两者的叠加态, 展现了比传统电子计算机元件更丰富的状态表示能力。量子计算机通过量子门来操纵这些状态, 可以并行处理大量的叠加状态, 极大地提高了计算效率。量子纠缠更是赋予量子计算机解决特定问题的超常能力, 比如整数分解和搜索问题。

生物计算机 (biocomputer) 则是利用生物分子的特性来进行计算。**DNA 计算机**使用 DNA 分子的序列来表示状态, 并通过分子生物学技术控制这些状态的变化。生物计算机可在模拟和处理生物学相关问题上展现出巨大的潜力, 例如在药物开发和复杂网络的分析中。

不论是物理形态如算盘, 理论模型如图灵机, 还是利用电子、量子或生物特性的现代计算机, 它们都体现了计算的两个基本要素: 状态的记录和状态变化的控制。这一本质揭示了计算机科学的根本, 也指引了未来计算技术的发展方向。量子 and 生物计算机的出现并没有改变这一计算的根本本质, 而是以新的物理实现和计算机制, 拓展了其表达和控制状态的边界, 为解决特定类型的问题提供了新的可能性。这些进展不断验证着图灵机作为计算模型的普适性, 即所有可计算问题都可以在图灵机上计算。这一点在现代计算机设计和理论中仍然占据中心地位。

1.2 计算机硬件基础

图灵机是理论计算模型, 侧重于理论计算能力的定义。约翰·冯·诺依曼 (John Von Neumann) 在 1945 年提出了将这一理论模型应用于实际可构建的计算机系统, 这种系统被称为冯诺依曼机, 也称冯诺依曼架构。冯诺依曼架构成为了现代计算机设计的基础, 几乎所有现代电子计算机都是在这—架构的指导下构建的。

冯诺依曼架构为现代电子计算机的设计提供了基础框架, 其核心概念是将程序指令和数据存储在同一内存系统中, 并通过中央处理单元 (CPU) 执行这些程序。以下是现代电子计算机硬件的几个主要组成部分, 这些部分在冯诺依曼架构的基础上进行了适配和发展:

1.2.1 中央处理器 (CPU, Central Processing Unit)

CPU 可以被看作是计算机的“大脑”。它负责执行程序的命令, 处理数据, 以及确保其他硬件部件协调工作。**CPU** 的性能直接影响计算机的运行速度和效率。**CPU**

性能衡量的主要指标：

- **时钟速度 (Clock Speed)**：通常用千兆赫兹 (GHz) 表示，指 CPU 每秒能够执行的指令周期数。高时钟速度通常意味着更快的处理能力。目前广泛使用的个人计算机的时钟速度在 1–5GHz。
- **核心数量 (Number of Cores)**：CPU 内部包含的独立执行单元的数量。多核心处理器能够同时处理多个任务，提高多任务处理能力和整体性能。
- **线程数 (Number of Threads)**：CPU 能够同时处理的线程数。超线程技术 (Hyper-Threading) 可以使每个核心同时处理多个线程，提高并行处理能力。

1.2 .2 主存储器 (RAM, Random Access Memory)

RAM 是计算机的“短期记忆”。它用于存储当前正在运行的程序和数据。RAM 是短期存储，这意味着一旦电源关闭，其中的数据就会消失。如果 CPU 需要快速获取信息，就会从 RAM 中读取。可以把 RAM 想象成大脑记忆，方便随时使用。RAM 性能衡量的主要指标：

- **容量 (Capacity)**：以字节为单位（如 GB，千兆字节），表示 RAM 能够存储的数据量。更大的容量允许同时运行更多和更大型的程序。
- **数据传输速率 (Data Transfer Rate)**：通常用每秒兆字节 (MB/s) 或每秒千兆传输率 (GT/s) 表示，指 RAM 能够传输数据的速度。高传输速率意味着更快的数据读写能力。
- **通道数 (Number of Channels)**：如单通道、双通道和四通道等，表示内存控制器与 RAM 之间的数据通道数量。多通道配置可以提升数据传输性能。
- **类型 (Type)**：常见的类型包括 DDR (Double Data Rate) 系列，如 DDR3、DDR4、DDR5 等。不同类型的 RAM 在性能、功耗和兼容性等方面有所不同。

1.2 .3 外部存储设备

外部存储设备主要是硬盘驱动器 (HDD) 和固态驱动器 (SSD)。硬盘驱动器使用磁盘存储数据，而固态驱动器使用闪存芯片，速度更快。这部分可以看作是计算机的“长期记忆”，用于存储不常用的文件、照片、软件等。即使计算机关机，数据也不会丢失。可以把外部存储设备理解为笔记本。外部存储设备性能衡量的主要指标：

- **存储容量 (Storage Capacity)**: 表示设备能够存储数据的总量, 通常以字节 (B)、兆字节 (MB)、千兆字节 (GB) 或兆兆字节 (TB) 为单位。
- **数据传输速率 (Data Transfer Rate)**: 表示设备读写数据的速度, 通常以每秒兆字节 (MB/s) 或每秒千兆字节 (GB/s) 为单位。
- **接口类型 (Interface Type)**: 表示设备与计算机连接的方式, 如 SATA、NVMe、USB、Thunderbolt 等。接口类型决定了设备的兼容性和数据传输速率。高性能接口 (如 NVMe 和 Thunderbolt) 通常提供更高的数据传输速率。
- **耐久性 (Durability)**: 表示设备在各种环境条件下的可靠性和使用寿命, 包括抗震性、耐温性和数据擦写次数 (特别是对于 SSD)。对于需要长时间稳定运行和频繁数据读写的场景, 较高的耐久性是有必要的。

1.2 .4 主板 (Motherboard)

主板是计算机的核心电路板, 连接并协调计算机各个组件的工作, 它负责数据和电力在各硬件部件之间的传输。主板性能的主要衡量指标:

- **芯片组 (Chipset)**: 主板上的一组集成电路, 负责管理数据流和控制硬件组件的工作, 通常包括北桥 (或内存控制器) 和南桥 (或 I/O 控制器)。
- **CPU 插槽 (CPU Socket)**: 用于安装 CPU 的接口类型, 如 LGA、AM4 等。
- **内存插槽 (Memory Slots)**: 用于安装 RAM 的插槽数量和类型, 如 DDR4、DDR5 等。
- **扩展插槽 (Expansion Slots)**: 用于安装扩展卡的插槽类型和数量, 决定了主板可以扩展的显卡、声卡、网卡等外设的种类和数量。
- **存储接口 (Storage Interfaces)**: 用于连接存储设备的接口类型和数量, 如 SATA、M.2 等。
- **I/O 接口 (I/O Ports)**: 主板后部和前部提供的输入输出接口, 如 USB、HDMI、音频接口、以太网接口等, 决定了主板可以连接的外围设备种类和数量。

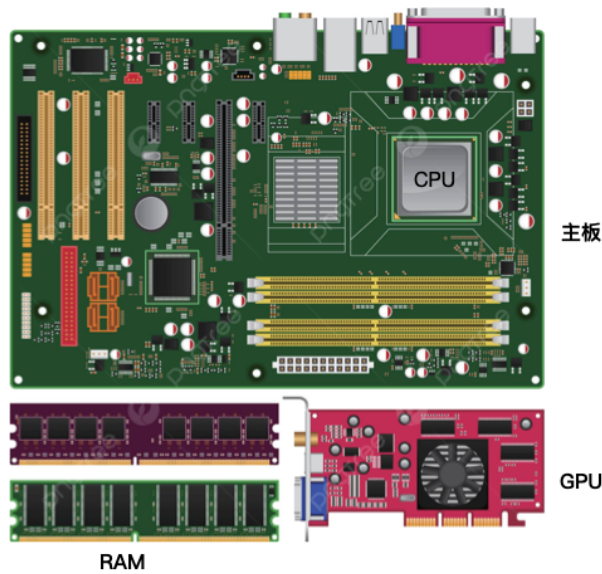


图 1.3: 个人计算机主要硬件

1.2 .5 显卡 (Graphics Processing Unit, GPU)

GPU 最初设计用于处理图形渲染任务，即将计算机生成的图形数据转换为可视图像的过程。具体而言，它包括将 3D 模型、纹理和光照等信息转换为 2D 图像，显示在屏幕上。渲染任务涉及多种计算，如几何计算、光线跟踪、着色等，以生成逼真的图像和动画。GPU 在图形渲染中发挥关键作用，因为它能够高效地处理大量并行计算，快速生成复杂的视觉效果。现代显卡因其强大的并行处理能力使其在现代计算中扮演了越来越重要的角色。特别是在机器学习领域，GPU 能够加速大规模矩阵运算和深度学习模型的训练，显著提高计算效率和速度。其大规模并行处理能力使其成为深度学习算法训练和推理的理想选择。

显卡主要性能衡量指标：

- **GPU 核心架构 (GPU Core Architecture)**：指 GPU 内部的设计和架构，如 NVIDIA 的 Ampere、Turing 等。不同的架构在性能、能效和功能上有显著差异，影响整体图形处理能力和计算性能。
- **CUDA 核心/流处理器数量 (CUDA Cores/Stream Processors)**：CUDA 核心（用于 NVIDIA GPU）或流处理器（用于 AMD GPU）的数量。更多的核心意味着更强的并行处理能力，适用于图形渲染和计算密集型任务。

- **显存容量 (VRAM Capacity):** 显卡上配备的显存容量, 通常以 **GB** 为单位。更大的显存容量允许处理更高分辨率的图形和更多的纹理数据, 特别是在游戏和专业图形应用中。
- **核心频率 (Core Clock Speed):** GPU 核心的工作频率, 通常以 **GHz** 为单位。更高的核心频率通常意味着更高的计算性能, 但也伴随更高的功耗和热量。
- **显存带宽 (Memory Bandwidth):** 显存与 GPU 之间的数据传输速率, 通常以 **GB/s** 为单位。
- **浮点运算能力 (Floating Point Performance):** 通常以每秒万亿次浮点运算 (TFLOPS) 为单位, 表示 GPU 的计算能力。对于科学计算、机器学习和 3D 渲染等应用, 浮点运算能力是关键性能指标。
- **显示输出接口 (Display Output Ports):** 显卡上提供的显示输出接口种类和数量, 如 HDMI、DisplayPort、DVI 等。该指标决定了显卡能够连接的显示器数量和类型。

1.2 .6 输入/输出设备 (I/O)

输入/输出设备允许用户与计算机系统交互, 并与外部世界通信。

- **输入设备:** 键盘、鼠标、触控板等。
- **输出设备:** 显示器、打印机、扬声器等。

显示器是计算机系统最重要的输出设备, 其主要性能衡量指标:

- **分辨率 (Resolution):** 显示器屏幕上能够显示的像素数量, 通常以宽度 x 高度表示, 如 1920x1080 (全高清, FHD)、2560x1440 (四倍高清, QHD)、3840x2160 (超高清, UHD 或 4K)。高分辨率提供更清晰和细腻的图像质量, 适合高精度图形工作和高分辨率内容观看。
- **刷新率 (Refresh Rate):** 显示器每秒能够刷新图像的次数, 通常以赫兹 (Hz) 为单位, 如 60Hz、120Hz、144Hz、240Hz。高刷新率提供更流畅的视觉体验, 特别是在快速运动的场景中, 如游戏。

- **面板类型 (Panel Type):** 显示器使用的面板技术, 如 **IPS (In-Plane Switching)**、**TN (Twisted Nematic)**、**VA (Vertical Alignment)**、**OLED (Organic Light-Emitting Diode)**。不同面板类型在色彩准确性、视角、响应时间和对比度等方面有不同表现。**IPS** 面板通常提供更好的色彩和视角, 而 **TN** 面板响应时间更短。
- **色域 (Color Gamut):** 显示器能够显示的颜色范围, 通常表示为 **sRGB**、**Adobe RGB** 等标准的覆盖百分比。较广的色域提供更丰富和准确的色彩表现, 适用于专业图形设计和视频编辑。
- **连接接口 (Connectivity Ports):** 显示器提供的输入输出接口种类和数量, 如 **HDMI**、**DisplayPort**、**USB-C**、**VGA**、**DVI** 等。丰富的接口支持多种设备连接和扩展功能, 提高显示器的兼容性和实用性。

1.3 计算机软件基础

软件是计算机系统中的“指令集”, 它告诉硬件应该如何工作和执行指定任务。软件和硬件是计算机系统中密不可分的两个组成部分, 它们之间的关系相互依存, 共同协作以实现计算机的各种功能。计算机中主要软件的结构如下图所示:

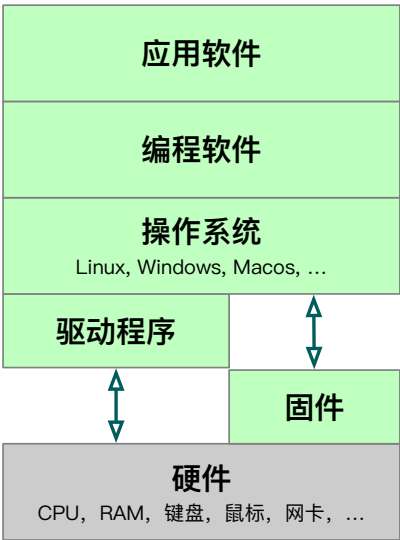


图 1.4: 计算机系统软件结构

1.3 .1 固件 (Firmware)

固件是一种嵌入在硬件设备中的特殊软件，用于控制和管理设备的底层功能。它通常存储在只读存储器（ROM）或其他非易失性存储介质中，并在设备启动时由硬件直接加载。例如，BIOS（基本输入输出系统）或 UEFI（统一可扩展固件接口）在计算机启动时初始化和测试硬件。其主要特点有：

- **持久性**：固件存储在非易失性存储器中，即使断电也不会丢失。
- **专用性**：固件针对特定硬件进行优化，确保设备运行的正确性和高效性。
- **更新频率**：固件的更新频率较低，通常只有在需要修复错误或改进性能时才进行更新。

1.3 .2 驱动程序 (Driver)

驱动程序是一种使操作系统和硬件设备能够交互的软件。它提供了一个必要的接口，通过这个接口，操作系统能够发送指令给硬件设备，并从设备接收数据。例如打印机驱动程序使电脑能够与打印机通信，执行打印任务。为确保硬件与各种操作系统的兼容，每种硬件通常需要特定的驱动程序。更新较固件更频繁，以适应新的操作系统和硬件的改进。

1.3 .3 操作系统 (Operating System, OS)

操作系统是一种用于管理计算机硬件和软件资源的关键软件。它不仅为用户与计算机的交互提供了基本平台，还负责管理文件系统、设备和正在运行的程序。常见的操作系统包括 Windows、macOS 和 Linux，它们在全球范围内被广泛使用。操作系统的主要特点有：

- **多任务**：允许同时运行多个应用程序。
- **用户界面**：提供图形用户界面 (GUI) 和命令行界面 (CLI) 以供用户操作。GUI 适合一般用户和需要图形化操作的应用场景，提供直观、易用的交互方式。CLI 适合开发者、系统管理员和高级用户，提供高效、灵活的操作方式快速执行复杂任务，效率高于 GUI，但需要用户具备一定的专业知识和经验。
- **资源管理**：高效管理处理器、内存和存储等资源。

1.3.4 编程软件 (Programming Software)

编程软件是帮助程序员开发其他软件的工具，包括编译器、解释器、文本编辑器、集成开发环境（IDE）等。Pycharm、Visual Studio 和 Eclipse 等 IDE 提供了多种编程语言代码编写、调试和版本控制的多项功能。

1.3.5 应用软件 (Application Software)

应用软件是为用户执行特定任务而设计的软件。这些任务可以是文档编辑、数据管理、图像设计等。广泛使用的应用软件有：

- Microsoft Office 包括 Word、Excel、PowerPoint 等应用，专注于办公自动化。
- Adobe Photoshop 专用于图像编辑和设计。
- Stata、Eviews、Spss 专用于统计和计量分析。
- MySQL、Oracle、SQL Server 等数据库软件帮助企业管理大量的用户数据、交易记录、产品信息等。
- AutoCAD、Ansys、Abaqus 等计算机辅助设计 (Computer Aided Design CAD) 和计算机辅助工程软件 (Computer Aided Engineering CAE)，主要用于工程项目的设计和分析。

众多的软件类型共同构成了现代计算设备丰富的软件生态系统。

1.4 操作系统与文件管理

1.4.1 Windows 系统

Windows 操作系统是由美国微软公司 (Microsoft Corporation) 开发的一系列操作系统。它是全球最广泛使用的桌面操作系统之一，广泛应用于个人电脑、办公环境、教育场所以及企业中。

A 主要特点

- 图形用户界面 (GUI)：Windows 提供了一个直观的、基于窗口的图形用户界面，用户可以通过鼠标和键盘与之交互。这种界面包括开始菜单、任务栏、以及文件资源管理器等元素。

- **兼容性**: Windows 支持广泛的硬件和软件产品, 这是通过与硬件制造商和软件开发商的合作实现的。这意味着用户可以在 Windows 系统上运行各种应用程序, 包括游戏、办公软件、图形设计软件等。
- **多任务处理**: Windows 允许用户同时运行多个应用程序。系统通过任务栏和多窗口管理功能, 使得切换和管理多个应用程序变得容易。
- **安全性**: 自 Windows XP SP2 起, 微软增加了许多安全功能, 如 Windows Defender 防病毒软件、防火墙、以及定期的安全更新, 以保护用户免受恶意软件和网络攻击的侵害。
- **支持网络**: Windows 包含了广泛的网络功能, 支持各种类型的网络连接 (包括有线和无线连接), 并提供了共享文件和打印机等网络服务。

B 文件操作

在 Windows 系统中, 用户主要通过“文件资源管理器”来进行文件操作。常见的操作包括:

- **创建文件/文件夹**: 右键点击然后从菜单中选择“新建”, 选择需要新建的对象。
- **复制和粘贴**: 使用右键菜单或者快捷键 **Ctrl+C** (复制) 和 **Ctrl+V** (粘贴)。
- **删除文件/文件夹**: 选择文件后按 **Delete** 键, 或使用右键菜单中的“删除”选项。
- **重命名**: 右键点击文件或文件夹后选择“重命名”, 或直接点击已选中的文件或文件夹名称。

C 文件路径

在 Windows 中, 文件路径通常使用反斜杠 (\) 作为目录分隔符。例如, 一个典型的文件路径可能看起来像这样:

```
1 | C:\Users\Username\Documents\file.txt
```

这表明文件 `file.txt` 位于系统盘 (C 盘) 下的“Users”文件夹内, 具体在“Username”用户的“Documents”目录中。

D 命令行界面工具

Windows 提供了“命令提示符”和“PowerShell”两种命令行界面工具。命令提示符是一个较为基础的命令行工具, 可以执行文件操作、系统管理等任务。**Power-**

Shell 是一个更强大的脚本环境，支持复杂的脚本编写和任务自动化。**Windows** 命令行界面的主要功能：执行系统管理任务，文件操作，网络监控等。

- 启动 **Windows** 命令提示符：

- 通过开始菜单

1. 点击“开始”按钮：位于屏幕左下角的 **Windows** 图标。
2. 搜索“**cmd**”或“命令提示符”：输入“**cmd**”或“命令提示符”。
3. 启动命令提示符：在搜索结果中，点击“命令提示符”应用程序。

- 通过运行对话框

1. 按下 **Win + R** 键：打开运行对话框。
2. 输入“**cmd**”：在文本框中输入“**cmd**”。
3. 按下“确定”或 **Enter** 键：启动命令提示符。

”命令提示符“成功启动后的图形窗口如下图：

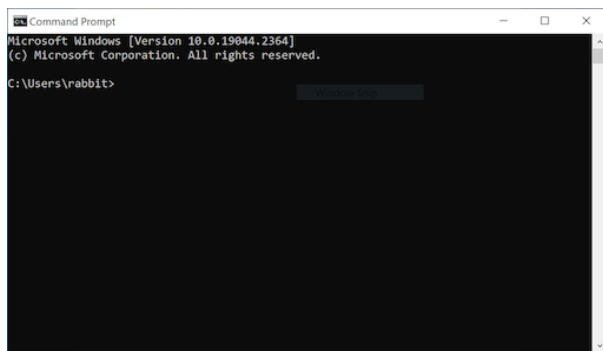


图 1.5: Windows 命令提示符窗口

1.4.2 MacOS 系统

macOS 是苹果公司（Apple Inc.）专为其 **Mac** 计算机系列开发的操作系统。它最早在 2001 年发布，当时名为 **Mac OS X**，后来简化为 **macOS**。这是一个基于 **Unix** 的操作系统，以其稳定性、效率和出色的图形用户界面而受到广泛赞誉。**macOS** 和苹果的其他操作系统，如 **iOS**、**watchOS**、**tvOS**，共享了许多设计理念和功能，使得苹果设备间的协同工作更加流畅。

A 主要特点

- **图形用户界面**: macOS 提供了一个清晰、现代的图形用户界面, 包括一个特色的 Dock 栏、Finder 文件管理器以及 Mission Control 等多任务管理特性。
- **集成性与互操作性**: macOS 与苹果的其他设备和服务高度集成, 例如, 而 Air-Drop 则允许用户快速在 Mac 和其他苹果设备之间传输文件。
- **安全性与隐私**: macOS 在安全性方面具有天然优势, 其 Unix 基础提供了良好的安全架构。苹果还增加了多项安全措施, 如 Sandboxing (沙盒化应用程序)。
- **兼容性**: 虽然 macOS 主要支持苹果硬件, 但它也支持广泛的第三方软件, 包括专业级的视频编辑软件、音乐制作工具和图形设计软件等。

B 文件操作

在 macOS 中, 文件操作主要通过“访达”(Finder) 来执行。其界面和功能与 Windows 的文件资源管理器类似, 但也有一些独特的特性如 Quick Look (快速预览文件而不实际打开它们)。

- **创建文件/文件夹**: 在 Finder 中, 通过“文件”菜单选择“新建文件夹”, 或使用快捷键 **Command+Shift+N**。
- **复制和粘贴**: 使用 **Command+C** 和 **Command+V** 快捷键。
- **删除文件/文件夹**: 将文件拖到废纸篓, 或使用 **Command+Delete** 快捷键。
- **重命名**: 选中文件后按回车键, 然后输入新的名称。

C 文件路径

在 macOS 中, 文件路径使用正斜杠 (/) 作为目录分隔符。例如, 一个典型的文件路径可能是:

```
1 | /Users/Username/Documents/file.txt
```

这表示文件位于硬盘的“Users”目录下的“Username”文件夹里的“Documents”中。

D 终端 (Terminal)

MacOS 的终端是一个基于 Unix 的命令行界面, 它允许用户通过键入命令来执行各种任务。终端的主要功能有:

- **命令执行**：用户可以输入各种命令，如文件操作、程序启动、系统监控等。
- **脚本运行**：用户可以编写并运行 **Shell** 脚本，这些脚本可以自动化复杂的任务序列。
- **系统管理**：通过终端，用户可以访问全套的系统管理工具，进行软件安装、更新、网络配置等。
- **环境配置**：用户可以通过编辑配置文件来定制和控制他们的工作环境，例如设置环境变量。

启动 Terminal:

1. 按下 **Command () + Space** 键：打开 Spotlight 搜索。
2. 输入 **“Terminal”**：在搜索结果中选择 **“Terminal”** 应用程序。
3. 按下 **Enter** 键

Terminal 启动成功后的图形窗口如图：

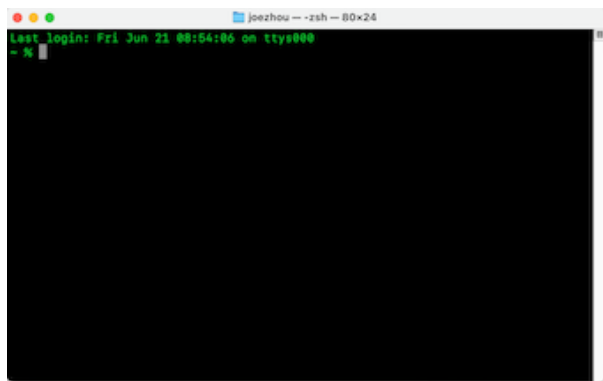


图 1.6: Terminal 界面

1.4 .3 Linux 系统

Linux 是一种强大的开源操作系统，它基于 1991 年由林纳斯·托瓦兹（Linus Torvalds）创建的 Linux 内核。Linux 系统是 Unix-like 操作系统，其设计和实现遵循了 Unix 的设计哲学。由于其开源本质，Linux 已经被开发成多种不同的版本，这些版本通常被称为“发行版”（distributions，简称“distros”）。流行的发行版有 Ubuntu, Fedora, Debian, CentOS, Arch Linux 等。

A 主要特点

- **开源性**: **Linux** 的最大特点是它的开源性。源代码可以被任何人自由地查看、修改和重新分发。
- **安全性**: **Linux** 被认为是非常安全的操作系统, 它的权限和用户管理系统可以有效防止恶意软件和病毒。
- **稳定性**: **Linux** 系统非常稳定, 适用于长时间运行而无需重启的服务器和系统。
- **灵活性和可配置性**: **Linux** 系统可以高度自定义。用户可以控制几乎所有的系统方面, 包括使用的桌面环境、安装的软件以及系统的行为。
- **免费使用**: 大多数 **Linux** 发行版都是免费提供的, 这使得它们成为预算有限的用户和教育机构的理想选择。

B 文件操作

Linux 系统中的文件操作通常通过命令行 (使用终端) 或图形用户界面 (GUI) 完成。在命令行中, 常用的文件操作命令包括:

- **ls**: 列出目录内容。
- **cd**: 改变当前目录。
- **cp**: 复制文件或目录。
- **mv**: 移动或重命名文件或目录。
- **rm**: 删除文件或目录。
- **mkdir**: 创建新目录。
- **touch**: 创建新文件或更新已存在文件的时间戳。
- **chmod**: 改变文件或目录的权限。
- **chown**: 改变文件或目录的所有者。

这些命令提供了管理文件和目录的基本方法, 实用且功能强大。可以使用 **man** 命令获得其他命令的使用帮助文档。如 **man ls** 就可以查看 **ls** 命令的使用方法。

C 文件路径的定义

在 **Linux** 中, 文件路径可以是绝对路径或相对路径:

- **绝对路径**: 始于根目录 (**/**), 例如 **/usr/local/bin**。不管当前位置在哪里, 绝对路径总是指向同一个文件或目录。

- **相对路径**: 相对于当前工作目录的路径。例如, 如果当前目录是 `/home/user`, 则相对路径 `documents/report.txt` 指向 `/home/user/documents/report.txt`。

了解和正确使用文件路径对于高效地使用 **Linux** 系统至关重要。

D 终端

Linux 的终端 (**Terminal**) 是 **Linux** 系统中用于与操作系统交互的命令行界面, 它的功能和界面与 **MacOS** 基本一致。

1.4 .4 金融科技中的操作系统

在金融科技 (**FinTech**) 领域, 不同的系统根据其独特的特性和优势被应用于不同的场景。下面分别介绍这三种系统在金融科技中的应用。

A Linux

Linux 因其开源性、高度的可定制性及其在服务器和云基础设施中的广泛使用, 成为金融科技领域的重要操作系统。**Linux** 的稳定性和安全性使得它成为运行关键金融服务器和后端系统的理想选择。

许多金融服务提供商使用基于 **Linux** 的服务器来托管交易平台、数据库和应用程序接口 (**APIs**)。此外, **Linux** 是许多区块链平台和加密货币挖矿操作的首选操作系统, 因为它的安全性、可靠性和高度可配置性。

B MacOS

macOS 由于其高级的安全特性和稳定性, 被许多金融科技初创公司和软件开发者采用。特别是在移动应用和网页设计领域, **macOS** 是首选系统之一, 因为它提供了优秀的开发工具如 **Xcode**, 并且与 **iOS** 设备的高度整合性使得测试和部署 **iOS** 应用更为便捷。金融科技开发者常使用 **MacOS** 设备进行软件开发, 尤其是针对苹果产品平台的应用程序。

C Windows

Windows 在企业环境中的普及使其在金融科技行业中也占有一席之地。**Windows** 操作系统的广泛使用和对各种商业软件的兼容性, 使其成为许多金融机构的标准桌面环境。

Windows 广泛用于运行企业级金融软件，如 ERP 系统、客户关系管理 (CRM) 系统和各种财务管理工具。许多专业的交易和分析软件都是为 Windows 系统设计的，包括股票和外汇交易平台，金融数据平台，如 Bloomberg, Eikon。

1.5 基本编程概念

当我们谈论计算机编程时，我们实际上是在讨论如何让计算机帮助我们自动化执行各种任务。编程主要涉及如下几个基本概念：

1.5.1 什么是程序？

程序就是这些具体指令的集合，用来告诉计算机如何执行特定的任务。你可以把程序想象成一个食谱，每一个步骤都详细说明了如何从原料制作出最终的菜肴。在编程中，这些“原料”可以是数据，而“菜肴”就是用户需要的输出结果，比如计算结果、信息列表或者是游戏中的一个动作。

1.5.2 为什么需要程序？

计算机本质上是一台非常快速和精确的机器，但它需要详细的指令来执行任务。这些指令需要非常具体和详尽，因为计算机虽然处理信息速度快，但它并不理解模糊的指令或假设。因此，编程就是创建一系列指令，告诉计算机如何完成一个特定的任务，比如计算账单、编辑照片，或者甚至是控制机器人。

1.5.3 如何编写程序？

编程语言是一种用于编写指令，让计算机执行特定任务的形式语言。想象一下，如果你想和一个不会说你的语言的人沟通，你需要通过一个翻译来帮助你们理解彼此。在编程的世界里，编程语言就是那个“翻译”，它帮助我们（人类）和计算机沟通。

A 编程语言的角色

编程语言的主要作用是提供一种结构化和可管理的方式表达计算逻辑和数据操作。通过编程语言，我们可以：

- 定义数据存储方式（例如变量和数据结构）。
- 描述数据之间的操作（例如算术运算或逻辑运算）。
- 控制程序流程（例如循环、条件判断和函数调用）。

B 编程语言的选择

不同的编程语言设计有不同的目标和专长。例如，有些语言非常适合于开发复杂的操作系统（如 **C**），而其他语言可能更适合于快速开发网络应用（如 **JavaScript**）。选择哪种编程语言往往取决于任务的具体需求、目标平台、执行效率和开发团队的熟悉程度。

C 编程的主要步骤

1. 定义问题

在开始编程之前，首先需要明确你想要解决的问题是什么。这可能涉及到数据处理、用户输入、计算或其他任何任务。对问题的清晰理解是成功编程的基础。

2. 规划解决方案

一旦问题定义清楚，下一步是规划如何解决这个问题。这通常涉及到：

- **算法设计**：设计一系列步骤来解决问题。算法是独立于任何编程语言的，它仅仅描述了如何通过一系列逻辑步骤来解决问题。
- **数据结构选择**：确定最适合用于实现算法的数据结构，比如数组、链表、树、哈希表等。

3. 选择编程语言

基于问题的性质和特定需求（如性能要求、平台依赖性、开发时间等），你需要选择一个合适的编程语言。例如，如果是需要高性能的应用，可能会选择 **C** 或 **C++**；如果是数据分析或机器学习，可能会选择 **Python**。

4. 编写代码

使用所选择的编程语言，根据设计的算法和数据结构开始编写代码。这个阶段包括：

- **编码**：将算法转换成具体的编程语句。

- **使用库和框架**：许多编程语言提供标准库或第三方库，这些库包含了可重用的代码，可以帮助你更快地开发程序。

5. 测试和调试

编写代码后，需要测试程序以确保它按预期工作。这个阶段可能涉及：

- **单元测试**：测试程序的独立部分是否正确。
- **集成测试**：确保多个部分协同工作时程序的正确性。
- **调试**：在发现代码中的错误后，使用调试工具来帮助定位和修复这些错误。

6. 优化和维护

程序完成基本功能后，可能需要根据反馈进行优化和调整，提高效率，或者修复在使用过程中发现的问题。程序的维护是一个持续的过程，可能包括升级功能、改进用户界面和修复安全漏洞。

7. 部署和运行

最后，程序需要在目标环境中部署并运行。这可能涉及配置服务器、设置用户访问权限等。

1.5 .4 程序是如何执行的？

当编写好程序后，它通常是以一种高级编程语言的形式存在，如 **Python**、**Java** 或 **C++**。这种形式的程序对人类来说相对容易理解，但对计算机来说还不能直接执行。因此，程序需要先被翻译成计算机可以理解的语言——这种语言被称为机器语言，它由一系列的 **0** 和 **1** 组成。

这个翻译过程可以通过编译或解释来完成：

- **编译**：编译器将整个程序转换为机器语言，生成一个可执行文件，之后可以直接在计算机上运行，无需再次转换。
- **解释**：解释器逐行读取源代码，每读取一行就现场转换并执行这行代码。这意味着不生成可执行文件，每次运行程序时都需要源代码和解释器。

程序的执行是一个涉及多个复杂步骤的过程，主要包括以下关键阶段：

1. 加载程序

- **读取可执行文件**：操作系统从磁盘读取可执行文件的内容到主内存中。
- **准备执行环境**：操作系统为程序准备必要的执行环境，如分配内存空间、初始化程序需要的资源等。

2. 解析和执行

- **CPU 调度**：操作系统的调度器决定程序何时获得 CPU 的执行时间。
- **指令解码**：CPU 从内存中读取指令，解码指令以确定需要执行的操作。
- **执行指令**：CPU 执行解码后的指令，包括算术运算、数据传输、条件分支等操作。

3. 系统调用和外围设备管理

- **系统调用**：程序执行过程中可能需要进行输入输出操作、文件操作等，这些通过系统调用实现，由操作系统管理。
- **资源管理**：CPU、内存和输入输出设备的使用由操作系统统一调度和管理，确保系统稳定运行。

4. 中断处理

- **外部中断**：如输入设备的数据到达或网络通信等，操作系统响应中断，暂停当前程序执行，处理中断请求。
- **内部中断**：由程序错误或异常情况引起，如除零错误、访问非法内存地址等。

5. 结束执行

- **程序终止**：程序执行完毕后，操作系统进行清理工作，如释放分配的资源，并结束程序的执行。
- **返回控制**：控制权返回给操作系统，可能返回到一个用户界面或终端，或者操作系统继续执行其他任务。

1.6 Python 语言的特点

Python 是一种高级编程语言，由 Guido van Rossum 在 1991 年首次发布。它是一种解释型语言，以其清晰、简洁和可读性高的语法而闻名，适用于多种编程范式，包括面向对象、命令式、函数式和过程式编程。Python 的主要特点包括：

- 易于学习和使用

Python 的语法非常接近英语，这使得 Python 特别容易学习和理解。它的结构和命名约定都旨在使代码尽可能清晰明了。

- 可读性强

Python 有一个强制的缩进规则，要求代码块必须正确缩进，这提高了代码的可读性和一致性。

- 广泛的标准库

Python 自带了一个庞大的标准库，这些库支持多种程序任务，如文件 I/O、系统调用、网络通信等，几乎可以在不使用外部库的情况下编写任何普通程序。

- 跨平台

Python 可以在多种操作系统上运行，包括 Windows、Mac OS X 和 Linux，代码通常不需要修改就能在这些平台上运行。

- 丰富的第三方库

除了丰富的标准库之外，Python 还有一个活跃的开源社区，开发了大量的第三方库，支持从网页开发到数据科学再到机器学习等各种高级应用。

- 多用途

Python 被广泛用于网络开发、自动化脚本、科学计算、数据分析、人工智能、机器学习等领域。它的灵活性使其成为初学者和专业开发者的热门选择。

- 支持多种编程范式

Python 支持面向对象、命令式以及函数式编程，使得开发者可以选择最适合问题的方法来选择编程范式。

- **动态类型**

Python 是动态类型的语言，这意味着不需要在代码中声明变量的类型。系统会在运行时自动确定类型，这使得 **Python** 在编写代码时更加快速和灵活。

- **易于集成**

Python 可以轻松集成 **C**、**C++**、**Java** 等其他语言编写的代码。这使得 **Python** 在需要优化性能的场所，可以用这些语言编写关键部分。

1.6 .1 **Python** 与金融科技

Python 在金融科技（**FinTech**）领域的广泛应用可以归因于它的多种特性，使其非常适合满足金融行业的多种需求。**Python** 在金融科技中被广泛使用的主要原因有：

1. 处理数据的能力

金融行业依赖于大量的数据分析和处理，**Python** 提供了强大的数据处理能力。它拥有如 **Pandas**、**NumPy** 等数据分析库，这些库能够高效地处理和分析大规模的数据集，非常适合金融数据分析、量化交易、风险管理等领域。

2. 金融建模和数学工具

Python 支持多种用于数学和统计分析的库，如 **SciPy** 和 **StatsModels**。这些工具使得金融机构能够执行复杂的算术计算、统计分析和金融建模，非常适用于定价模型、风险评估和其他金融研究。

3. 易于学习和使用

Python 语法简洁明了，易于学习和编写，这使得非程序员背景的金融专业人士（如分析师和交易员）也能快速学习并使用 **Python** 来进行数据分析和自动化任务。

4. 强大的金融分析相关的库生态系统

Python 拥有丰富的工具库生态系统，涵盖了从数据获取、数据清洗、数学运算到数据可视化等几乎所有步骤。特别是在金融领域，有许多专门的库（如 QuantLib、PyAlgoTrade、FinRL 等），专注于金融工程、算法交易和投资组合管理。

5. 高效的开发速度

Python 的开发效率高，能够快速地从概念验证到实际部署。在金融科技领域，市场环境变化迅速，快速响应市场变化和调整策略是非常重要的。Python 的这一特性使得金融机构能够迅速开发和部署新的工具和应用。

6. 适合机器学习和人工智能

Python 是机器学习和人工智能领域的主流语言之一。它拥有 TensorFlow、Scikit-learn、Pytorch 等强大的机器学习库，这些都是金融科技领域中进行信用评级、欺诈检测、算法交易等应用的关键技术。

综上所述，Python 的这些特点使其在金融科技领域成为一种极具吸引力的选择，能够满足该领域快速增长和不断变化的需求。

2

Python 安装与运行

2.1 安装 Python 及其开发环境

2.1.1 安装 Python

A 安装方法

Python 的安装方式可以根据用户具体需求和使用场景来选择。主流的安装方法有两种：（1）使用 🐍Python 官网提供的安装程序，（2）使用 🐍Conda（特别是 Anaconda 或 Miniconda）进行安装。其中 Conda 方式安装具有如下优点：

- **包管理和环境管理：**Conda 不仅可以管理 Python 包，还可以管理非 Python 包（如库或应用程序），并且可以非常容易地创建、导出、复制和恢复整个环境。
- **简化复杂依赖：**Conda 能够自动处理包之间的依赖关系，特别是涉及到数据科学和机器学习的复杂环境。

关于“包依赖”的概念，我们可以用车间加工产品的例子来类比理解。假设你的车间负责生产一种精密的机械零件，比如一个齿轮。为了制造这个齿轮，你可能需要铁块、切削液、磨具和一些特定的机床。在这个例子中，齿轮的生产就“依赖”于这些原材料和设备。如果缺少了其中任何一个关键组件（比如没有磨具），你就无法正常完成这个齿轮的生产。

这种依赖关系意味着，为了顺利完成生产，车间管理者必须确保所有必需的材料和设备都已准备就绪。这与软件开发中的包依赖非常相似，开发者需要确保所有必要的库和工具都在项目开始前就已经安装和配置好，以确保软件能够顺利运行和构建。

在编程中，当我们开发一个软件或应用时，通常会用到一些已经写好的代码库，这些代码库我们称之为“包”。这些包可以帮助我们快速实现某些功能，比如处理网页数据、进行数学计算等，而不需要我们从头编写所有的代码。

“包依赖”就是指我们的程序为了能正常运行，需要依赖（即必须先安装）某些特定的包。例如，如果你的程序中需要对网页内容进行处理，你可能需要使用一个名为 **BeautifulSoup** 的包来帮助你解析网页。如果没有安装 **BeautifulSoup**，那么程序就会因为找不到需要的功能而无法运行，就像做蛋炒饭时没有鸡蛋一样。

总结来说，包依赖就是我们的程序为了执行某些特定功能，必须先有的一些外部代码包。如果没有这些必需的包，程序就无法正常工作。理解和管理好这些依赖关系，对于确保程序的稳定和高效运行非常重要。

- **广泛的第三方包支持**：Anaconda 提供了大量预编译的 Python 包，可以避免在安装时编译代码，特别是在 Windows 系统上尤为重要。

B Anaconda 还是 Miniconda?

- **Anaconda**：适合希望“开箱即用”的用户，尤其是在数据科学和机器学习领域。它预安装了许多常用科学计算和数据分析的库。
- **Miniconda**：提供了一个最小化的安装器，适合那些喜欢自定义安装内容的用户，安装后可以根据需要安装所需的包。

本书推荐选用 Miniconda 方式安装，主要原因如下：

1. 轻量级安装

Miniconda 提供了一个最小化的安装包，其中只包含了 Conda、Python 以及一些必要的库和工具。这使得 Miniconda 的安装过程快速且占用的磁盘空间较小，非常适合希望保持系统整洁的用户。

2. 高度可控的环境配置

由于 Miniconda 安装时不会自动安装一大批预设的库，用户可以完全控制哪些包被安装在环境中。这种灵活性特别适合需要精细管理其依赖的高级用户和开发者。

3. 强大的包和环境管理功能

Miniconda 继承了 Conda 的所有功能，包括包管理和虚拟环境管理。用户可以轻松创建、克隆、共享和管理复杂的项目环境。这对于确保项目在不同机器和团队成员之间的一致性非常有帮助。

4. 适合科学计算和数据分析

虽然 Miniconda 本身不预装科学包，但用户可以通过 Conda 轻松安装如 NumPy、Pandas、SciPy、Matplotlib 等数据科学相关的库。Conda 提供的包往往针对特定平台优化过，可以提供更好的性能。

C Miniconda 安装 Python 的步骤

下面是通过 Miniconda 安装 Python 的具体步骤：

1. 步骤 1: 下载 Miniconda 安装器

1. 访问 Miniconda 的官方网站：🐍Miniconda
2. 根据你的操作系统（Windows、macOS 或 Linux）选择合适的版本。下载适合你的操作系统的安装程序。各种系统对应的文件名形式如下：
 - Windows: Miniconda3-pyxxx_xx.x.x-x-Windows-x86_64.exe。
 - MacOS (Intel) : Miniconda3-pyxxx_xx.x.x-x-MacOSX-x86_64.sh
 - MacOS (Sillicon M1~4): Miniconda3-pyxxx_xx.x.x-x-MacOSX-arm64.sh
 - Linux: Miniconda3-pyxxx_xx.x.x-x-Linux-x86_64.sh

虽然通常建议用户使用最新 (latest) 的软件版本，因为它们包含最新的功能和安全更新。但考虑到许多 Python 库可能还没有更新以支持最新的 Python 版本，为了避免代码的兼容性问题，选择使用 Python 次新的版本可能会是最佳选择。因此，各系统平台推荐安装 Python3.11 版本的链接如下：

- 🐍Windows
- 🐍MacOS (Intel)
- 🐍MacOS (Sillicon)
- 🐍Linux

2. 步骤 2: 安装 Miniconda

Windows:

1. 双击下载的 `.exe` 文件开始安装。
2. 按照安装向导的指示操作。在安装过程中，可以选择“Add Miniconda to my PATH environment variable”，这样可以在任何命令行窗口中直接使用 Conda（虽然官方推荐不勾选此项，而是使用 Anaconda Prompt）。
3. 完成安装。

MacOS 和 Linux:

1. 打开终端 (Terminal)。
2. 使用 `cd` 命令切换到包含下载的 Miniconda 安装脚本的目录。
3. 运行安装脚本：
 - macOS: `bash Miniconda3-py311_24.4.7.1-0-MacOSX-x86_64.sh`
 - Linux: `bash Miniconda3-py311_24.4.7.1-0-Linux-x86_64.sh`
4. 按照终端中的指示完成安装，通常是阅读许可协议，同意许可协议，然后选择安装位置。
5. 如果在安装过程中选择将 Conda 初始化到您的 shell 中，您需要重新启动终端以激活安装。
6. 步骤 3: 设置并激活新的 Python 环境
7. 打开命令行界面（在 Windows 上使用 Anaconda Prompt，macOS 或 Linux 上使用标准终端）。
8. 创建一个新的 Conda 虚拟环境 (virtual environment)，并指定 Python 版本：

```
1 | conda create --name fintech python=3.11
```

这里 `fintech` 是你要创建的虚拟环境的名称，`python=3.11` 是指定的 Python 版本，你可以根据需要安装不同的版本。

想象一下，Python 的虚拟环境就像是一个可以随意装备和改造的小车间，专门用来进行特定的项目。在这个小车间里，你可以按照自己的需求安装各种工具和设备（库或工具），并且不会影响到其他车间。这样做的好处是，每个车间（项目）都可以有自己独特的设置，而这些设置不会互相干扰。

在更具体的技术层面上，Python 的虚拟环境是一个隔离的环境，它允许你为不同的项目安装不同版本的库或软件包，而这些安装不会影响到全局的 Python 安装（整个工厂）。这样做有几个好处：

1. **隔离性**：每个项目都在自己的虚拟车间中运行，这意味着它们不会共享安装的工具，从而避免了版本冲突。例如，如果一个项目需要版本 1.5 的某个工具，而另一个项目需要版本 2.2，这两个版本可以在各自的虚拟车间中安装和运行，而不会相互干扰。
2. **组织性**：虚拟车间帮助你保持工作区的整洁。每个项目都有自己的车间，只包含所需的工具，这使得项目更容易管理和理解。
3. **移植性**：因为每个虚拟车间都包含了项目所需的所有依赖，这使得将项目从一个机器移动到另一个机器变得更加简单。只需复制车间和代码，而不需要在新机器上重新配置所有东西。
4. **安全性**：在虚拟车间中实验新工具或库更加安全，不会影响全局安装，这样即使出现问题，也只限于那个特定的车间中。

创建和使用虚拟环境，就像是为每个项目搭建一个小车间，确保你在其中的活动不会打扰到别的项目，同时也使得你的工作更加有序。这对于维护多个项目或与他人合作时尤其有用。

3. 激活刚刚创建的环境：

```
1 | conda activate fintech
```

激活环境后，你将看到终端提示符前面出现环境名称。可以根据实际需要，创建自己命名的任意新环境。

4. 步骤 4：验证安装

检查 Python 是否正确安装在新环境中：

```
1 | python --version
```

这应该会显示你安装的 Python 版本。

2.1.2 安装 Python 开发环境

A 安装 Jupyter Notebook

激活新环境后，在该环境中安装 Jupyter Notebook，只需在命令行中输入以下命令：

```
1 | conda install notebook
```

安装成功后，在命令行输入如下命令以启动 Jupyter Notebook：

```
1 | jupyter notebook
```

这将在浏览器中打开 Jupyter Notebook 的界面如下：

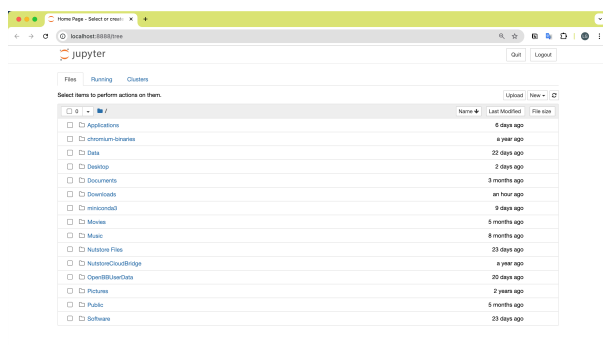


图 2.1: Jupyter Notebook 运行界面

B 安装 Visual Studio Code

Visual Studio Code (VS Code) 是一款非常受程序员欢迎的轻量级代码编辑器，支持多种编程语言，具有强大的功能扩展性。以下是在 Windows 和 macOS 上安装 VS Code 的步骤：

• Windows 上安装 VS Code

1. 下载安装程序：

- 访问 VS Code 官方网站：🐼 Visual Studio Code
- 点击首页的“Download for Windows”按钮，下载将自动开始。

2. 运行安装程序：

- 找到下载的 .exe 文件，并双击运行。
- 跟随安装向导，接受许可协议。
- 选择安装位置和其他安装选项（如添加“打开方式”快捷方式到右键菜单，注册 VS Code 为默认编辑器等）。
- 点击“安装”按钮，等待安装完成。

3. 启动 VS Code:

- 安装完成后，可以选择立即启动 VS Code。
- 之后，你可以从开始菜单或桌面快捷方式启动 VS Code。

• macOS 上安装 VS Code

1. 下载安装程序:

- 访问 VS Code 官方网站:  Visual Studio Code
- 点击首页的“Download for Mac”按钮，下载将自动开始。如果安装失败，请在 <https://code.visualstudio.com/Download> 上选择和电脑芯片类型匹配的版本。

2. 安装 VS Code:

- 找到下载的 .zip 文件，双击以解压，通常会自动解压到和下载的 zip 文件同一文件夹。
- 拖动解压出的 Visual Studio Code 应用程序到你的“应用程序”文件夹，这样就完成了安装。

3. 启动 VS Code:

- 打开“应用程序”文件夹，找到 Visual Studio Code 并双击打开。
- 第一次打开可能会看到一个警告说应用是从互联网下载的，点击“打开”继续。

使用提示

- 安全设置 (macOS): 如果无法打开因安全设置阻止，则需前往“系统偏好设置”->“安全性与隐私”，在“通用”选项卡中点击“仍要打开”。
- 更新: VS Code 通常会自动检查更新，确保你总是使用最新版本。

安装完 Visual Studio Code (VS Code) 之后，你需要为 Python 编程和 Jupyter Notebook 添加特定的插件，以增强编辑和运行 Python 代码的能力。以下是在 VS Code 中添加 Python 和 Jupyter 插件的步骤，适用于 Windows 和 macOS:

- 安装 Python 插件

Python 插件提供了代码补全、智能感知、代码格式化、调试等功能，是 Python 开发者的必备扩展。

1. 打开 VS Code。

2. 打开扩展视图

- 点击侧边栏中的扩展图标（四个小方块组成的图标，或使用快捷键 **Ctrl+Shift+X (Windows)** 或 **Cmd+Shift+X (macOS)**）。

3. 搜索 Python

- 在扩展搜索框中输入“Python”。
- 选择由 Microsoft 发布的 Python 扩展（通常是搜索结果中的第一个）。

4. 安装 Python 扩展

- 点击“安装”按钮。

• 安装 Jupyter 插件

Jupyter 插件允许在 VS Code 内直接编辑和运行 Jupyter 笔记本，支持.ipynb 文件格式。

1. 确保 Python 插件已安装

- Jupyter 插件依赖于 Python 插件，因此请确保先安装 Python 插件。

2. 搜索 Jupyter

- 在扩展搜索框中输入“Jupyter”。
- 选择由 Microsoft 发布的 Jupyter 扩展（通常也是搜索结果中的第一个）。

3. 安装 Jupyter 扩展

- 点击“安装”按钮。

• 配置 Python 环境

在安装完 Python 和 Jupyter 插件后，你可能需要配置 Python 解释器：

1. 打开命令面板

- 使用 **Ctrl+Shift+P (Windows)** 或 **Cmd+Shift+P (macos)** 打开命令面板。

2. 选择 Python 解释器

- 输入 **Python: Select Interpreter**，选择它。
- 从出现的列表中选择一个已安装的 **Python** 解释器。这通常是你的系统中安装的 **Python** 环境或虚拟环境。

• VS Code 中使用 Jupyter Notebook

安装了 Jupyter 插件后，你可以直接在 VS Code 中创建或打开 `.ipynb` 文件：

1. 创建 Jupyter 笔记本：

- 打开命令面板（**Ctrl+Shift+P** 或 **Cmd+Shift+P**）。
- 输入 **Jupyter: Create New Blank Notebook**，选择它。
- 开始编写你的 Jupyter 笔记本代码。

2. 打开现有的 Jupyter 笔记本：

- 通过文件菜单选择“打开文件”或直接拖放 `.ipynb` 文件到 VS Code 中。

2.2 第一个 Python 程序

对于初学者来说，编写第一个 Python 程序是一个重要的开始步骤。以下是一个简单的 Python 程序示例：

```
1 name = input('Please input your name:')
2 print(f'Hello, {name}')
```

该程序一共有两行代码：

1. 第一行代码的功能是获取用户用键盘输入的字符。`input()` 是 Python 中的一个内置函数，用于接收用户的输入信息。当程序执行到这一行时，它会暂停，并在终端或命令行界面显示括号内的字符串 `'Please input your name: '` 作为提示信息。用户输入内容后按回车键（**enter**），输入的内容将被赋值给变量 `name`。这样，变量 `name` 就存储了用户输入的名字。
2. 第二行代码用于输出一条消息。`print()` 是另一个内置函数，用来在屏幕上显示信息。这里使用了格式化字符串（也称为 **f-string**），它以 **f** 开头，并使用大括号 `{}` 包围变量名。当这行代码执行时，Python 会将 `name` 变量的值替换到

`{name}` 的位置，然后输出整个消息。例如，如果用户输入的名字是 "Alice"，那么输出将会是 `Hello, Alice`。

2.2.1 编写程序

1. 打开文本编辑器（如 Notepad (Windows), Visual Studio Code, 或任何其他代码编辑器）。
2. 将上述代码复制粘贴到一个新建的文件中。
3. 保存文件，扩展名为 `.py`，文件全名如 `hello.py`。

2.3 运行 Python 程序

2.3.1 命令行方式

命令行方式运行 Python 程序具有多个重要的优势，尤其是在如远程服务器这样的环境中。命令行运行 Python 如此重要的主要原因：

1. 无需图形用户界面 (GUI)

远程服务器通常不安装图形用户界面。在这种环境下，命令行是执行和管理程序的主要方式。Python 能够很好地通过命令行方式运行，使其成为自动化任务和服务器级应用程序的理想选择。

2. 自动化和脚本化

命令行允许用户轻松地编写脚本来自动化各种任务，如数据备份、更新和复杂的计算过程。Python 脚本可以通过简单的命令行指令自动运行，这对于维护大型服务器和处理周期性任务非常有效。

3. 资源效率

运行命令行程序通常比运行图形界面程序占用更少的计算资源。对于需要高效资源使用的服务器环境，这一点尤其重要。

4. 远程访问和控制

在通过 SSH (Secure Shell Protocol 安全外壳协议) 远程连接到服务器时, 命令行是与服务器交互的主要方法。你可以在本地机器上编写 Python 代码, 然后通过命令行远程执行这些代码, 这对于远程软件开发和维护至关重要。

5. 集成和兼容性

许多服务器应用和工具都设计为通过命令行进行控制和配置。Python 脚本可以轻松集成到这些工具链中, 提供自动化支持、数据处理、日志分析等功能。

6. 批处理能力

命令行运行 Python 程序使得批量处理数据和任务变得简单。例如, 可以一次性处理成千上万个文件, 而无需手动干预每个过程。

• 运行步骤

1. 打开命令行界面 (在 Windows 中是 CMD 或 PowerShell, 在 macOS 和 Linux 中是 Terminal) 并激活 Python 虚拟环境。
2. 切换到保存 Python 文件的目录。例如 windows 中, 输入 `cd` 命令显示当前目录, 输入 `cd TargetDirectory` 切换到指定目标目录, `dir` 显示当前目录中的内容。
3. 运行命令 `python hello.py`。

2.3.2 IPython 方式

IPython (Interactive Python) 是一个增强版的 Python 解释器, 提供了标准 Python 解释器的所有功能, 外加一些有用的额外功能, 主要目的是提高编码、测试、调试的效率。以下是 IPython 的一些显著特点:

1. 交互式 Shell:

IPython 提供了一个增强的交互式命令行界面, 比默认的 Python shell 有更多的用户友好特性, 如自动缩进、代码补全和颜色高亮。

2. 内建的调试支持:

IPython 支持使用 `%debug` 命令进入交互式调试模式, 这使得追踪代码错误和异常变得更简单。

3. 强大的历史命令功能:

用户可以通过上下箭头访问历史命令，还可以搜索历史命令，这对于重复测试和调试非常有帮助。

4. 内联编辑支持:

IPython 允许使用 `%edit` 命令在文本编辑器中打开并编辑代码，然后直接在 Shell 中运行。

5. 集成的数据可视化:

与 Matplotlib 和其他图形库紧密集成，可以在 IPython 中方便地生成并查看图形。

6. 魔法命令:

IPython 提供了一系列的“魔法”命令（以 `%` 或 `%%` 开头），这些命令是为了方便各种常见任务而设计的，例如 `%timeit` 用于测量代码执行时间。使用 `%magic` 命令查看所有可用的魔法命令，按 `q` 键可以随时退出命令帮助查看环境。

• 运行步骤

1. 打开命令行界面（在 Windows 中是 CMD 或 PowerShell，在 macOS 和 Linux 中是 Terminal）并激活 Python 虚拟环境。

2. 在命令行中输入 `ipython` 来启动 IPython 环境

```
1 | ipython
```

3. 在打开 IPython 的交互式 Shell 中，输入 Python 代码进行实时执行:

```
1 | In [1]: name = input("Please input your name: ")
2 | In [2]: print(f'Hello, {name}')
```

2.3.3 Jupyter Notebook 方式

Jupyter Notebook 是一个开源的 Web 应用程序，允许你创建和共享包含实时代码、数学方程、可视化以及文本的文档。它支持多种编程语言，包括 Python，通过 IPython（Jupyter 的 Python 内核 (kernel)）提供强大的交互式编程环境。以下是 Jupyter Notebook 的一些主要特点:

1. 交互式代码单元：

用户可以在 **Notebook** 中编写并执行代码，代码执行的结果直接在代码块下方显示。这对于实验性编程、数据分析和教学非常有用。

2. 富文本元素：

Notebook 支持 **Markdown**，允许添加富文本元素，如标题、列表、链接和图片等。此外，还可以嵌入数学方程和其他媒体。

3. 数据可视化集成：

Jupyter 与数据可视化库如 **Matplotlib**、**Seaborn** 紧密集成，可以直接在 **Notebook** 中生成并嵌入图形。

4. 支持扩展：

Jupyter 社区提供了大量的插件和扩展，如交互式小部件、主题和工具，以增强 **Notebook** 的功能。

5. 跨平台和可移植：

作为 Web 应用程序，**Jupyter Notebook** 可以在任何支持现代 Web 浏览器的操作系统上运行。

• 运行步骤

1. 打开命令行界面或终端。
2. 切换到你的工作目录。
3. 输入 `jupyter notebook` 命令，按 **Enter**。这将在你的默认浏览器中启动 **Jupyter Notebook**，通常会打开一个新的标签页显示 **Jupyter** 的文件浏览界面。
4. 在 **Jupyter** 的文件浏览界面，你可以通过点击右上角的“New”按钮来创建新的 **Notebook**。
5. 选择 **Python 3** 或你安装的任何其他环境来开始一个新的 **Notebook**。
6. 在打开的 **Notebook** 页面中，你可以通过点击单元格并输入 **Python** 代码或 **Markdown** 文本来使用它。按 **Ctrl + Enter** 可以执行单元格中的代码或渲染 **Markdown** 文本。

A Jupyter Notebook 的使用

Jupyter Notebook 提供了两种主要的工作模式：编辑模式和命令模式。

- 编辑模式

编辑模式允许你在单元格内直接编辑文本。你可以输入代码或使用 **Markdown** 语法格式化文本。

- 如何进入：点击单元格，或者在命令模式下按 **Enter** 键。
- 辨识方式：单元格边框会变成绿色，并且可以看到光标闪烁。
- 命令模式

命令模式让你可以使用键盘快捷键来操作 **Notebook**，比如添加或删除单元格、改变单元格类型、移动单元格等。

- 如何进入：按 **Esc** 键或者点击单元格外的任何区域。
- 辨识方式：单元格边框会变成蓝色。
- 切换方法
 - 从编辑模式切换到命令模式：在单元格内编辑时，按 **Esc** 键。
 - 从命令模式切换到编辑模式：选中单元格后，按 **Enter** 键。
- 命令模式下的常用快捷键

- 创建新单元格

- * A：在当前单元格上方插入新单元格。
 - * B：在当前单元格下方插入新单元格。

- 删除单元格

- * DD（连按两次 D）：删除当前单元格。

- 更改单元格类型 **Notebook** 有两种基本的单元格类型：(1) 代码单元格：默认的单元格类型，用于输入和执行代码。(2) **Markdown** 单元格：用于书写格式化文本，使用 **Markdown** 语法。

- * **Y**: 将当前单元格设置为代码单元格。
- * **M**: 将当前单元格设置为 **Markdown** 单元格。
- 复制、剪切和粘贴单元格
 - * **C**: 复制当前单元格。
 - * **X**: 剪切当前单元格。
 - * **V**: 在当前选中单元格下方粘贴剪贴板的单元格。
- 移动单元格
 - * **Shift + Down**: 向下扩展选中的单元格。
 - * **Shift + Up**: 向上扩展选中的单元格。
- 执行单元格
 - * **Shift + Enter**: 运行当前单元格并移动到下一个单元格。
 - * **Ctrl + Enter**: 运行当前单元格并保持在当前单元格。
 - * **Alt + Enter**: 运行当前单元格，并在下方插入一个新单元格。
- 保存和分享 Notebook
 - * 保存: 点击工具栏上的“保存”图标，或使用快捷键 **Ctrl + S** (Windows/Linux) 或 **Cmd + S** (Mac) 来保存你的 Notebook。
 - * 导出: 你可以将 Notebook 导出为多种格式 (如 HTML, PDF, Markdown 或 Python 脚本) 通过“File”菜单下的“Download as”。

可以在 Jupyter Notebook 的官网，查阅  更多的 Jupyter Notebook 使用指南。

B Markdown 基本语法

Markdown 是一种广泛使用的轻量级标记语言，它使文本可以方便地转换为 HTML 或其他格式。Markdown 的目标是易读易写，它的语法简洁明了，学习起来非常容易。下面，将介绍 Markdown 的一些常用用法和语法，这些用法在编写文档、撰写博客、创建报告等场景中非常有用。

1. 标题

Markdown 使用 **#** 符号来创建标题。**#** 的数量表示标题的级别。例如：

```

1  ## 标题 1
2  ### 标题 2
3  #### 标题 3
4  ##### 标题 4
5  ##### 标题 5
6  ##### 标题 6

```

2. 强调文本

- 粗体：将文本两侧用两个星号 ****** 或下划线 **__** 包围。
- 斜体：将文本两侧用一个星号 *** 或下划线 *_* 包围。

```

1  ** 粗体文本 **
2  __ 粗体文本 __
3  *** 粗斜文本 ***
4  ___ 粗斜文本 ___
5  * 斜体文本 *
6  _ 斜体文本 _

```

3. 列表

- 无序列表使用星号 *、加号 + 或减号-。
- 有序列表使用数字后跟点号 1.、2. 等。代码：

```

1  - 项目一
2  - 项目二
3  - 项目三
4
5  1. 第一项
6  2. 第二项
7  3. 第三项

```

4. 链接与图片

- 链接：[显示文本](URL)
- 图片原图显示：![替代文本](图片 URL)
- 图片指定大小显示

```

1  [OpenAI](https://www.openai.com)
2  ![Logo](https://upload.wikimedia.org/wikipedia/commons/4/4d/OpenAI_Logo.svg)
3  

```

5. 引用

使用 `>` 符号来创建引用区块。代码：

```
1 | > 这是一段引用文本。
```

6. 代码

- 内联代码：用单个反引号“包围代码。
- 代码块：用三个反引号“`````”或三个波浪号“`~~~`”包围代码段，并可指定语言。

```
```python
def hello():
 print("Hello, World!")
```
```

7. 表格 Markdown 表格使用竖线 `|` 和短横线`-`来创建。短横线用于分隔表头和其他行。代码：

```
1 | 名称 | 描述 |
2 | ---:|:---|
3 | Apple | 苹果 |
4 | Banana | 香蕉 |
```

8. 水平线

使用三个或更多的短横线`---`、星号 `***` 或下划线 `___` 来创建水平线。

9. 数学公式

Markdown 文本中可以添加 LaTeX 格式的数学公式，行内公式使用前后单个美元符号包围，块级公式使用前后双美元符号包围。这些公式在 Markdown 渲染时会被转换为相应的 HTML，然后由 MathJax 或 KaTeX 渲染成最终的视觉格式。常用的公式如下：

a. 基本数学运算

```
1 | $a + b, a - b, a \times b, a \cdot b, a / b, \frac{a}{b}$
```

$$a + b, a - b, a \times b, a \cdot b, a/b, \frac{a}{b}$$

b. 上标与下标

```
1 | $a^i, b_j, a^i_j$
```

$$a^i, b_j, a_j^i$$

c. 根号与对数

```
1 | $\sqrt{x}$, \sqrt[n]{x}, \log(x), \ln(x)$
```

$$\sqrt{x}, \sqrt[n]{x}, \log(x), \ln(x)$$

d. 希腊字母和特殊符合

```
1 | $\alpha, \beta, \gamma, \rho, \sigma, \delta, \epsilon, \Delta, \infty$
```

$$\alpha, \beta, \gamma, \rho, \sigma, \delta, \epsilon, \Delta, \infty$$

e. 方程与对齐

```
1 | \begin{align*}
2 | a &= b + c \\
3 | x &= y - z \\
4 | \end{align*}
```

f. 矩阵

```
1 | \begin{bmatrix}
2 | a & b \\
3 | c & d \\
4 | \end{bmatrix}
```

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

g. 求和、积分与连乘

```
1 | \sum_{i=1}^n a_i
2 | \int_a^b f(x) \, dx
3 | \prod_{i=1}^n a_i
```


$$\sum_{i=1}^n a_i$$
$$\int_a^b f(x) dx$$
$$\prod_{i=1}^n a_i$$

h. 极限与导数

```
1 \lim_{x \to a} f(x) \\
2 \frac{\partial y}{\partial x} \\
3 \frac{dy}{dx}
```

$$\lim_{x \rightarrow a} f(x)$$
$$\frac{\partial y}{\partial x}$$
$$\frac{dy}{dx}$$

想了解更多公式的写法，请查阅📖[Latex 公式指南](#)。

• **Markdown 使用场景**

Markdown 的应用非常广泛，常见的使用场景包括：

- **技术博客**: 许多博客平台支持 **Markdown**，使得撰写和格式化文章更加方便。
- **项目文档**: 在 **GitHub** 等代码托管平台上，**Markdown** 通常用于编写 **README** 文件和其他文档。
- **学术写作**: 支持 **Markdown** 的编辑器和工具可以用来撰写科研论文或笔记。
- **书籍编写**: 有些作者使用 **Markdown** 来撰写并发布整本书籍，本书就是采用 **Markdown** 书写，然后通过 **pandoc** 转换为 **latex** 格式来编写完成的。

2.3.4 集成开发环境 (IDE) 方式

使用 IDE 来运行 Python 程序提供了许多便利，如代码补全、调试支持和项目管理。一些流行的 Python IDE 包括 PyCharm、Visual Studio Code。

- **PyCharm**: 专为 Python 开发设计的强大 IDE，提供丰富的功能，如代码调试、项目导航、版本控制和虚拟环境支持，通常适合开发大型项目中使用。
- **Visual Studio Code**: 一个轻量级且功能强大的编辑器，支持通过扩展来增强 Python 开发功能。

IDE 方式运行 Python 代码建议查阅对应的官网使用指南或手册。

2.3.5 运行方式的选择

由于金融科技涉及大量的数据分析和可视化，方便用户编写并即时运行代码，查看代码执行结果，Jupyter Notebook 的文件格式（.ipynb）易于分享，并且可以通过网络托管服务（如 GitHub、NBViewer）轻松共享给他人查看，建议初学者优先选择 Jupyter Notebook 作为运行环境。

练习

1. 安装 Python 运行环境，运行 Python 程序
 1. 安装 miniconda
 2. 创建环境 fintech
 3. 编写课程中的 Python 示例程序
 4. 掌握四种不同方式运行该程序，IDE 选用 VS Code。
2. 创建一个新的 Jupyter Notebook，命名为 `first.ipynb`，在该笔记本中，写一个 Markdown 单元格，练习书中介绍的所有 Markdown 语法。
3. 添加一个 Markdown 单元格，练习书中介绍的所有 Markdown 语法，并书写如下公式：

- $E = mc^2$
- $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$
- $a^2 + b^2 = c^2$
- $\int_a^b f(x) dx$

$$\bullet \hat{y} = b_0 + b_1x_1 + b_2x_2 + \epsilon$$

4. 添加一个代码单元格，复制如下代码并运行：

```
1 total = 0
2 for i in range(1, 101):
3     total = total + i
4 print(f'1+2+...+100={total}.')
```

5. 保存 first.ipynb 文件，关闭 Jupyter Notebook 后，再次打开 first.ipynb 文件。

3

状态表示 — 数据类型

3.1 信息形态与数据类型

计算机的主要功能包括接收、处理、存储和传输信息。从最基本的个人电脑到复杂的服务器和超级计算机，所有计算机系统都在执行这些基本的信息处理任务。而信息有多种形态，并且根据信息的性质和用途，我们可以使用不同的数据类型来有效地存储、处理和传输这些信息。如下是几种常见的信息形态以及它们对应的数据类型：

1. 数值信息

数值信息涉及数字，可以是整数或小数，用于计算、测量和其他需要精确数值度量的场景。数据类型主要包括：

- **整型 (Integer)**：用于表示没有小数部分的数字，如股数、手机销量等。
- **浮点型 (Float/Double)**：用于表示有小数部分的数字，如收益率、温度等。

2. 逻辑信息

逻辑信息通常表示是非判断或二元状态，用于决策制作和条件判断。通常对应如下数据类型：

- **布尔型 (Boolean)**：只有两个可能的取值（真 **True** 或假 **False**）。适用于需要表示对/错、是/否等二元决策的情况。

3. 文本信息

文本信息是最常见的信息形态之一，包括书籍、文章、报告中的文字等。对应数据类型：

- **字符串 (String)**：用于存储和表示文本数据。在编程中，字符串可以包含字母、数字和其他符号，用于表达诸如句子、名称或任何其他文字信息。

4. 图像信息

图像信息以视觉形式出现，如照片、图表和其他图形。数据类型：

- **二进制数据**：图像文件通常以二进制形式存储（如 JPEG, PNG, GIF 文件格式）。
- **数组/矩阵**：在处理图像内容时，像素数据常常被存储在数组或矩阵中，每个元素代表一个像素点的颜色值。彩色图像通常 **RGB** 颜色表示模式。在 **RGB** 颜色模式中，每个像素由红色（R）、绿色（G）、和蓝色（B）三种颜色的强度组成。每种颜色的强度由 8 个比特位表示，从 0 到 255。因此每个像素总共占用 24 位（3 字节）。例如，红色可以表示为（255, 0, 0）。

5. 音频信息

音频信息包括音乐、对话、声音效果等。数据类型：

- **二进制数据**：音频文件（如 MP3, WAV）通常以二进制格式存储。
- **波形数据**：在音频处理中，声音信号常被转换为波形数据，通常是采样值的时间序列。

6. 视频信息

视频信息结合了图像和音频，用于电影、监控、会议等应用。

- **二进制数据**：视频文件（如 MP4, AVI）通常以二进制格式存储。
- **复合数据结构**：视频流可能包含多个数据类型的组合，例如图像帧序列加上时间同步的音频轨道。

3.2 变量与数据

计算的本质就是状态的记录和状态变化的控制。在现代电子计算机中，任何状态的记录都是由一系列 **0** 和 **1** 的数字组成，这些数字系列就是记录状态的数据。在程序运行时，这些数据通常存储在内存（**RAM**）。现代计算机通常配备有多达数十千兆字节（**GB**）的随机存取内存（**RAM**）。这种广阔的内存空间允许多个应用程序同时运行，同时存储大量的数据和复杂的程序指令。内存的容量越大，计算机处理大规模数据和多任务操作的能力越强。每个内存单元都有一个唯一的地址，**CPU** 通过这些地址读取或写入数据。因为内存的空间庞大，通常需要较长数字表示地址。程序员对数据进行操作，如果是通过数据地址对数据进行访问，即使是几个数据，都是非常困难的。而**变量**的使用可以解决这一问题。没有变量，程序员需要手动管理内存地址来存储和检索数据，这不仅复杂而且容易出错。通过使用变量，程序员可以更专注于解决问题的逻辑，而不是数据存储的细节。

3.2.1 变量

变量可以被视为数据存储的“名称”或“标签”，它们使得程序员能够以易于理解和引用的方式操作存储在计算机内存中的数据。当程序中定义一个变量时，计算机会在内存中分配一个空间来存储该变量的值。这个内存地址通常是随机的，程序员通常不需要知道其具体的地址。变量名则用作这个内存位置的标签，使得程序可以通过变量名来访问或修改数据。

想象你的手机电话号码本是一个存储朋友、家人和同事电话号码的地方。在这个电话号码本中，每个联系人的名字代表一个“变量”，而与之关联的电话号码则是这个变量所存储的“数据”。

A 变量命名

在这个比喻中，当你想要联系某人时，你不需要记住他们的电话号码，你只需要知道他们的名字。在编程中，变量的作用也类似。你定义一个变量来引用某个具体的信息（如一个数值、一个字符串等）。因此，变量名（例如 **ZhangSan**）就像联系人的名字，它是你用来访问存储在内存中的数据（即电话号码）的标签。

• Python 变量的命名规则

1. 字符限制

- **字母和数字**: 变量名可以包含字母 (a-z, A-Z) 和数字 (0-9), 但不能以数字开头。
- **下划线**: 可以使用下划线 (_) 作为变量名的一部分, 例如 `_name` 或 `name_1`。

2. 开头字符

- 变量名必须以字母或下划线开头。

3. 大小写敏感

- **Python** 对变量名是大小写敏感的, 因此, `variable`、`Variable` 和 `VARIABLE` 是三个不同的变量。

4. 避免使用关键字

- **Python** 的关键字不能用作变量名, 因为关键字用于定义语言的结构和功能。
Python 3.11 的关键字如下表:

| | | | | | |
|--------|----------|-------|--------|----------|--------|
| 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 | |

5. 描述性命名

- 虽然不是语法要求, 但推荐使用表达性强的名称来帮助其他人或自己理解变量的用途。例如, 使用 `count` 比单个字母 `c` 更能清晰表达变量的意图。

B 数据的存储

每当你在电话号码本里保存一个新的联系人和他们的电话号码时, 你实际上是建立了电话号码和联系人的关联。在编程中, 当你声明一个变量并赋予它一个值时, 计算机的内存就会存储这个值。例如, 执行 `ZhangSan = "130123456789"` 就是在告诉计算机 “请在内存中找到一个空间, 并标记为 `ZhangSan`, 并在那里存储数值 `"130123456789"`”。

数据类型告诉计算机变量需要多少内存空间以及如何解释储存在内存中的数据。例如，整数类型（如 `int`）可能会占用 4 个字节的内存，而浮点类型（如 `double`）可能会占用 8 个字节。正确的数据类型不仅帮助保证数据的正确解释，还能有效地管理内存资源。

C 访问和修改数据

就如同你可以随时查看或者修改电话号码本中某个联系人的电话号码一样，你也可以在程序中使用变量名来访问或修改存储在内存中的值。如果某个联系人更换了电话号码，你只需要在电话号码本中找到他们的名字，然后更新存储在那里的号码。在编程中，你可以通过简单地重新赋值来更新变量的值，比如 `ZhangSan = "130987654321"`。

D 变量的作用域

如果你的电话号码本是个人的，那么存储在其中的联系信息只能由你访问，这类类似于在程序中的局部变量，它只能在定义它的特定区域（如一个函数内部）被访问。如果电话号码本是家庭共享的，那么家里的任何成员都可以访问里面的任何联系信息，这类类似于全局变量，可在整个程序中访问。

当数据不再被使用时，就可以释放这部分数据所占用的空间。有些程序设计语言需要程序员管理变量对应内存数据的释放，而 `Python` 可以自动进行内存回收管理。

3.2.2 数学和计算机中的变量

A 数学中的变量

1. **概念性和抽象性**：在数学中，变量通常用来表示一个未知数或任意数，它是一个抽象的符号。
2. **普适性**：数学变量不依赖于任何具体的存储介质或环境，它们是完全抽象的，并用于理论推导和表达数学关系。
3. **不变性**：一旦在数学运算或推理过程中确定了变量的值，这个值在当前的计算或论证中通常保持不变。

B 计算机中的变量

1. **存储和引用**：计算机中的变量用于标识在内存中存储数据。每个变量都关联一个内存地址，变量名用作该地址的标签或引用。例如，在编程语言中，声明 `age = 30`；会在内存中分配空间来存储整数值 30，并将 `age` 作为这段内存的引用。
2. **可变性**：计算机变量的值通常是可变的，可以在程序执行过程中被改变或更新。变量的这种特性使得程序可以动态处理数据和状态。
3. **类型系统**：计算机变量具有数据类型，这定义了变量可以存储的数据的种类（如整数、浮点数、字符串等）和所占的存储空间。数据类型决定了该数据的处理方式和内存的使用。

C 主要区别

- **抽象与具体**：数学变量更抽象，不涉及具体的存储和数据类型，而计算机变量具体到内存存储和数据类型。
- **功能目的**：数学变量用于表达数学关系和逻辑推理；计算机变量则是为了数据存储、程序状态管理和过程控制。

3.2.3 赋值语句

赋值语句的主要功能是将一个数据与一个变量名关联起来。当一个赋值语句执行时，发生以下几个步骤：

1. **计算右侧表达式**：赋值语句右侧的表达式（运算式）被首先计算。这可以是一个简单的值、一个复杂的表达式或者是一个函数调用的结果。
2. **创建或更新变量**：如果变量之前未定义，则创建一个新变量。如果变量已经存在，则更新其值。
3. **存储值**：将右侧表达式的结果值存储在变量名指向的内存位置中。

A 赋值语句形式

- **基本赋值**：

– `x = 30 + 42` # 将表达式的结果赋值给一个变量。

- **链式赋值**：

– `x = y = 31.2` # 同时将同一个值赋给多个变量。

- **增量赋值**（复合赋值）：

- `variable += 5` # 这些操作符将变量与值进行特定的运算，并将结果重新赋给该变量，还有其它操作符 `--`，`*`，`/` 等。

- **多重赋值**：

– `a, b = 10, 20` # 同时为多个变量赋值。

3.3 Python 的数据类型

在本节中，我们将深入探讨 Python 中的常用数据类型，包括简单数据类型和复合数据类型。理解这些常用数据类型对于编写高效的 Python 程序至关重要。我们将首先介绍各种数据类型的特征，最后通过示例展示如何在实际编程中有效地使用这些类型。

3.3.1 简单数据类型

A 整数 (Integers)

整数类型代表没有小数部分的数值。它们可以是正数、负数或零。Python 中，整数类型不像 C 或 Java 中的整型那样有固定的大小限制（如 32 位或 64 位）。Python 的整数可以扩展到使用几乎所有可用内存，因此，理论上它可以表示非常大的数，数的上界受可用内存空间的限制。

整数可以表示为不同的进制形式，包括十进制、二进制、八进制和十六进制。

- **十进制 (Decimal)**

十进制是我们日常生活中最常用的数制系统。Python 中输入的数字，默认就是十进制的表示。例如，数字 $29 = 2 \times 10^1 + 9 \times 10^0$ 。在读数时，我们通常把 10^0 对应位置的数称为个位， 10^1 对应位置的数称为十位，以此类推。

- **二进制 (Binary)**

二进制是基于 2 的数制，只使用数字 0 和 1。在 Python 中，二进制数以 0b 或 0B 开头。例如，十进制数 29 在二进制中写作 0b11101:

$$11101 = 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 16 + 8 + 4 + 0 + 1 = 29$$

• 八进制 (Octal)

八进制是基于 8 的数制，使用数字 0 到 7。在 Python 中，八进制数以 0o 或 0O 开头。例如，十进制数 29 在八进制中写作 0o35：

$$35 = 3 \times 8^1 + 5 \times 8^0$$

• 十六进制 (Hexadecimal)

十六进制是基于 16 的数制，使用数字 0 到 9 和字母 A 到 F（或 a 到 f，分别表示 10, 11, ..., 15）。Python 中十六进制数以 0x 或 0X 开头。例如，十进制数 29 在十六进制中写作 0x1d：

$$1d = 1 \times 16^1 + 13 \times 16^0$$

• 进制的相互转换

这些不同进制在数字表达能力上是一致的，它们之间可以进行相互转换。Python 提供了如下表所示的内置函数来转换这些不同的进制表示。

| 函数 | 功能 | 输入参数 | 输出结果 |
|-------|---------------|-----------|--------------------|
| bin() | 将整数转换为二进制字符串 | 十进制整数 | 二进制字符串（以 '0b' 开头） |
| oct() | 将整数转换为八进制字符串 | 十进制整数 | 八进制字符串（以 '0o' 开头） |
| hex() | 将整数转换为十六进制字符串 | 十进制整数 | 十六进制字符串（以 '0x' 开头） |
| int() | 将字符串或数字转换为整数 | 字符串或数字，基数 | 十进制整数 |

• 示例代码

```
1 # 定义十进制数
2 decimal_number = 29
3
4 # 定义二进制
5 binary_number = 0b11101
6
7 # 十进制转换为二进制
8 binary_string = bin(decimal_number)
9 print(binary_string) # 输出 '0b11101'
10
11 # 十进制转换为八进制
```

```
12 octal_string = oct(decimal_number)
13 print(octal_string) # 输出 '0o35'
14
15 # 十进制转换为十六进制
16 hexadecimal_string = hex(decimal_number)
17 print(hexadecimal_string) # 输出 '0x1d'
18
19 # 二进制字符串转换为十进制整数
20 decimal_from_binary = int('0b11101', 2)
21 print(decimal_from_binary) # 输出 29
22
23 # 八进制字符串转换为十进制整数
24 decimal_from_octal = int('0o35', 8)
25 print(decimal_from_octal) # 输出 29
26
27 # 十六进制字符串转换为十进制整数
28 decimal_from_hexadecimal = int('0x1d', 16)
29 print(decimal_from_hexadecimal) # 输出 29
```

上述将十进制数据转换为其它进制的函数，输出的结果均为字符串。我们稍后讨论字符串数据类型。

B 浮点数 (Floats)

浮点数是带有小数部分的数值。它们用于表示更精确的值或科学计算。我们知道在整个数轴上有无穷多个实数，而计算机中标准的浮点数通常用 **32** 位（单精度）或 **64** 位（双精度）来存储，有限的数据位数无法一一对应的表示无穷的数量状态，因此，浮点数自然具有精度和表示范围的限制。表示浮点数的这些位分为三部分：

- 符号位（表示正负）
- 指数位（决定数值的范围）
- 尾数位（决定数值的精度）

标准的浮点数主要有如下两种表示方法：

- 准小数形式：

- 这是最直接的浮点数表示方式，例如 3.14、-0.001 和 123.456。这些都是直接用小数点和数字表示的浮点数。

- 科学记数法：

科学记数法用于表示非常大或非常小的数。这种格式使用 `e` 或 `E` 来表示 `10` 的幂。例如: `-1.5e2` 表示 1.5×10^2 , 即 `150`。 `-6.02E23` 表示 6.02×10^{23} 。 `-1.23e-4` 表示 1.23×10^{-4} , 即 `0.000123`。

Python 中, 最大精度约为 15 到 17 位有效数字, 但浮点数的精确度不是固定的, 因为随着数值的增大, 可用于表示小数部分的位数减少, 因此表示的精度也随之下降。浮点数可以表示的数值范围大约是 2.22×10^{-308} 到 1.79×10^{308} 。

• 示例代码

```
1 # 测试浮点数精度
2 x = 0.1
3 y = 0.2
4 z = x + y
5 print(z)
6 # 输出不会精确为 0.3, 而是一个非常接近的数, 如 0.30000000000000004
7
8 # 测试浮点数值范围
9 w = 2e308
10 print(w) # 输出 inf, 表示结果超出浮点数的表示范围, inf 表示一个无穷大的数
```

• 精确表示

- 由于标准浮点数可能会有精度问题, Python 提供了 `decimal` 模块支持高精度的十进制浮点数。例如:

```
1 from decimal import Decimal # 导入对象 如同在工作台上放好要用的工具
2 Decimal('0.1') # 可以精确表示 0.1, 而不是一个近似值。
```

- 对于分数的精确表示, Python 提供了 `fractions` 模块支持精确的分数计算, 可以防止浮点数运算中可能出现的精度问题。例如:

```
1 from fractions import Fraction
2 f = Fraction(3, 4) # 表示 3/4
3 f = Fraction(0.5) # 表示 1/2
4 f = Fraction('0.5') # 表示 1/2
```

需要了解更多的 `decimal` 和 `fractions` 模块的功能, 可以点击阅读 Python 的官方文档。在金融数据分析中, 这两种精确数据表示形式使用较少, 故不做更详细的介绍。

数值运算

• 运算符

整数和浮点数主要用于数值运算。**Python** 程序中的数值运算主要由如下数值运算符定义：

| 运算符 | 描述 | 示例 | 结果 |
|-----|--------|--------|-----|
| + | 加法 | 5 + 3 | 8 |
| - | 减法 | 5 - 3 | 2 |
| * | 乘法 | 5 * 3 | 15 |
| / | 除法 | 5 / 2 | 2.5 |
| // | 整除 | 5 // 2 | 2 |
| % | 取模（余数） | 5 % 2 | 1 |
| ** | 幂运算 | 5 ** 2 | 25 |

• 表达式

表达式是一种结合操作符、运算符、函数调用的组合运算结构，通常计算会得到一个值。表达式的主要作用是执行计算，产生结果或值。表达式通常包括以下几部分：

- 1. **操作数 (Operands)**：这些是表达式计算的数据，可以是常量、变量名或者更复杂的表达式。
- 2. **运算符 (Operators)**：定义了对操作数进行的操作类型，如加法、乘法、逻辑比较等。
- 3. **函数调用**：函数调用本身也可看着是一种表达式，其返回值可以是任何类型的数据，这个值可以被进一步用作其他表达式的一部分。最常用的函数为 **Python** 的内置函数，下表给出最基本和常用的内置函数，请阅读 **Python** 官方文档了解更多的内置函数说明。

| 函数 | 功能描述 | 输入 | 输出 |
|---------|-----------------|------------|-----------------------------------|
| abs() | 返回数的绝对值 | 数字（整数或浮点数） | 绝对值 |
| bool() | 将给定参数转换为布尔值 | 任何数据类型 | 布尔值（ True 或 False ） |
| float() | 将一个字符串或数字转换为浮点数 | 字符串或数字 | 浮点数 |

| 函数 | 功能描述 | 输入 | 输出 |
|----------------------|---------------------|------------------------|----------------|
| <code>int()</code> | 将一个字符串或数字转换为整数 | 字符串或数字 | 整数 |
| <code>len()</code> | 返回对象（字符串、列表、字典等）的长度 | 任何序列或集合类型 | 整数（长度） |
| <code>max()</code> | 返回最大值 | 至少两个参数或一个可迭代对象 | 最大值 |
| <code>min()</code> | 返回最小值 | 至少两个参数或一个可迭代对象 | 最小值 |
| <code>print()</code> | 打印给定对象 | 对象、变量 | 无（输出到控制台） |
| <code>round()</code> | 四舍五入到给定的精度 | 数字，精度（可选） | 四舍五入后的值 |
| <code>str()</code> | 将对象转换为字符串 | 任何对象 | 字符串 |
| <code>sum()</code> | 计算输入的总和 | 可迭代的数字对象 | 总和 |
| <code>type()</code> | 返回对象的类型 | 任何对象 | 对象的类型 |
| <code>dir()</code> | 列出对象的所有属性和方法 | 对象或无（无参数时列出当前范围的属性和方法） | 属性和方法列表 |
| <code>help()</code> | 提供对象的帮助文档 | 对象或无（无参数时进入交互式帮助系统） | 帮助文档（通常在控制台输出） |

Python 众多的内置函数中，`type()`，`dir()`，和 `help()` 是三个极其重要和常用的内置函数。它们在学习阶段和编写代码时提供了极大的帮助，尤其是在处理不熟悉的代码或库时。

`type()` 函数用来查询一个对象的类型。在 Python 中，了解变量或数据结构的数据类型是非常重要的，因为不同类型的数据支持不同的方法和操作。使用 `type()` 可以帮助开发者理解变量的具体类型，从而正确地处理数据。此外，当代码的行为不符合预期时，检查数据类型常常能帮助快速定位问题的根源。

`dir()` 函数用来列出对象的所有属性和方法。当你使用一个模块或者一个对象时，常常需要知道它可以做什么，或者说它提供了哪些方法（功能）。`dir()` 可以快速给你列出一个对象的所有可用属性和方法，这对于学习新模块或者理解某个对象的功能极为有用，是辅助编写和测试代码的重要工具。

`help()` 函数提供了一个交互式帮助界面，用于查看对象、类型、模块等的详细说明，包括它们的功能、参数和用法等。Python 的一个强大之处在于其丰富的文档。`help()` 函数可以直接在代码编辑器或终端中查看任何对象的文档，这对于理解和学习如何使用各种功能是非常有价值的。不论是标准库中的模块还是第三方库，`help()` 都能提供即时的帮助信息，这对于开发者快速掌握和应用新知识至关重要。例如：

复杂的数学运算函数通常由数学模块 `math` 提供。例如：

```
1 import math # 导入模块 如同取工具箱到操作台
2 result = math.sqrt(25) # 结果为 5
3 result = math.sin(math.pi / 2) # 结果为 1.0
4 result = math.exp(1) # e**1, 结果约为 2.71828
5 result = math.log(2.71828) # 结果接近 1
6 result = math.pi # 3.141592653589793
7 result = math.e # 2.718281828459045
```

4. 括号：用于改变运算顺序或明确表达式的界限。

• 运算符优先级

运算符优先级决定了表达式中运算符的计算顺序。在复杂的表达式中，括号可以用来改变计算顺序。以下是一些基本的运算符优先级规则：

- ****** 优先级最高。
- 然后是 *****、**/**、**//** 和 **%**。
- 加法和减法 **+**、**-** 优先级较低。
- 括号 **()** 可用于强制提升某部分表达式的优先级。

• 示例代码

```
1 x = 5
2 y = 3
3 result = x + y - x * y # 5 + 3 - 15
4 print(result) # 输出 -7
5 result = x + (y - x) * y
6 print(result) # 输出结果是 -1
```

C 布尔型 (Booleans)

布尔类型只有两个值：**True**（真）和**False**（假）。在 Python 中，布尔类型是整数类型的子类。本质上，布尔值就是整数。**True** 实际上等价于 **1**，而 **False** 则等价于 **0**。布尔值通常适合用于那些只有两种可能结果的场景，如条件判断（满足或不满足条件）和逻辑判断（真与假）。

在计算机科学中，主要的运算类型可以分为算术运算和逻辑运算。算术运算主要处理数值计算，使用算术表达式来表达，如加、减、乘和除法等。而逻辑运算则主要处理布尔值，使用逻辑表达式来表达，涉及逻辑运算符如逻辑与、逻辑或和逻辑非等。算术运算和逻辑运算的一个关键区别在于它们处理的数据类型不同。算术运

算的变量和常量通常是数值类型，如整数或浮点数。相比之下，逻辑运算的变量和常量则通常是布尔类型，即 **True** 或 **False**。这种区分使得逻辑运算特别适合于处理条件判断，而算术运算则广泛应用于数量的计算和处理。

• 逻辑表达式

逻辑表达式是由一组逻辑运算符和操作数构成的表达式，用于进行布尔运算，即操作和评估返回布尔值 (**True** 或 **False**) 的表达式。逻辑表达式有如下主要组成部分：

- 1. **操作数**：操作数通常是产生布尔值的表达式，可以是布尔值常数（如 **True** 或 **False**）、布尔值变量、关系表达式（如 **a > b**），或者是其他返回布尔值的函数或表达式。以下是 **Python** 中常用的关系运算符的介绍：

| 符号 | 描述 | 示例 | 结果 |
|--------------|-------|------------------|--------------|
| == | 等于 | 5 == 3 | False |
| != | 不等于 | 5 != 3 | True |
| > | 大于 | 5 > 3 | True |
| < | 小于 | 5 < 3 | False |
| >= | 大于或等于 | 5 >= 5 | True |
| <= | 小于或等于 | 5 <= 3 | False |

- 2. **逻辑运算符**：逻辑表达式中连接计算多个布尔值的运算符。基本的逻辑运算符包括：

- 逻辑非 (**not**)：用于反转其操作数的布尔值。如果操作数为 **True**，结果为 **False**，反之亦然。
- 逻辑与 (**and**)：如果操作数都为 **True**，结果才为 **True**；否则为 **False**。
- 逻辑或 (**or**)：只要有一个操作数为 **True**，结果就为 **True**；只有所有操作数都为 **False**，结果为 **False**。

逻辑运算符的优先级

逻辑运算符有不同的优先级和结合性规则，这决定了在没有括号明确指示的情况下，表达式中的运算顺序。通常，**not** 有最高的优先级，其次是 **and**，然后是 **or**。例如，在表达式 **not a and b** 中，**not a** 会首先被评估，然后结果与 **b** 进行 **and** 运算。

3. 为了改变默认的运算顺序或提高表达式的可读性，可以使用括号。括号内的表达式总是优先计算。

- 示例代码

```
1 a = 3 > 2
2 b = 5 < 3
3 condition1 = not a and b
4 conditon2 = not (a and b)
5 print(condition1) # 输出 False
6 print(condition2) # 输出 True
7 print(condition1 or condition2)
```

D 字符串 (Strings)

字符串是由字符组成的序列，用于存储和表示文字信息。在编程中，字符串通常被包围在引号内，如 "hello"、"1234" 或 'John Doe'。字符串的用途非常广泛，包括：

- 存储和显示文本信息，如用户姓名、地址等。
- 从文件中读取或向文件写入文本。
- 在网络上发送和接收文本数据。

尽管字符串可以包含数字（比如 "1234"），但这些数字只是字符的形式，不能直接用于数学计算。比如，字符串 "123" 和 "456" 无法直接相加得到 579，而是得到"123456"。

Python 字符串使用 Unicode 标准的字符。Unicode 是一个全球性的字符编码标准，旨在包括世界上所有的书写系统的字符。截至 Unicode 15.1 版本，Unicode 包含超过 **149,813** 个字符。Unicode 的设计允许每个字符分配一个从 0 到 0x10FFFF 的代码点，这提供了超过 1,114,112 个可能的代码点。这些代码点中的很多仍未使用，预留给将来扩展使用。代码点可以简单视为字符的编号。在 Python 中，可以通过 Unicode 编码（如 \uXXXX 和 \UXXXXXXXX 转义序列）直接在字符串中表示任何 Unicode 字符。Unicode 的字符主要有以下类别：

1. 基本拉丁字符 (ASCII) :

- 包括英文字母 (A-Z, a-z)、数字 (0-9) 和基本标点符号。

- 这是最初的 ASCII (American Standard Code for Information Inter-change) 字符集，共有 128 个字符 (0-127)。

2. 扩展拉丁字符:

- 包括有重音符号的字母和其他西欧语言特有的字符。

3. 全球字符:

- 包括其他语言的字符集，如汉字、希腊语、俄语、阿拉伯语、希伯来语、日文假名、韩文等。

4. 特殊符号:

- 包括数学符号、货币符号、法律和专利符号、音符等。

5. 表情符号:

- 包括各种表情符号，这些在社交媒体和文本消息中非常流行。

6. 控制字符:

- 包括像换行符 \n、制表符 \t 等，在控制文本格式中起着关键作用。

在 Python 中，转义字符通常用一个反斜线 (\) 开始，后跟一个字符，用于表示特殊的字符或控制序列。下表列出了一些最常用的转义字符：

| 转义字符 | 描述 | 示例 | 结果 |
|------------|--------------|----------------------|------------------------|
| \\ | 反斜线 | '\\' | \ |
| \' | 单引号 | 'It\'s easy.' | It's easy. |
| \" | 双引号 | "He said, \"Hello\"" | He said, "Hello" |
| \n | 换行 | 'Line 1 \nLine 2' | Line 1
Line 2 |
| \t | 水平制表符 (Tab) | 'Column 1\tColumn 2' | Column 1 Column 2 |
| \r | 回车 | 'Hello\rWorld' | World |
| \b | 退格 | 'abc\bdef' | abdef |
| \ooo | 八进制表示的字符 | '\141' | a |
| \xhh | 十六进制表示的字符 | '\x61' | a |
| \uXXXX | 代码点不超过 2 个字节 | \u56fd | 国 |
| \UXXXXXXXX | 代码点超过 2 个字节 | \U0001F349 | 🍷 |

D.1 字符串的表示 在 Python 中，字符串可以通过不同的表示形式来定义，下表介绍了这些不同的字符串表示形式：

| 表示形式 | 描述 | 示例 | 结果 |
|-------------|-------------------------------------|-------------------------------------|----------------|
| 单引号 (') | 用单引号包裹的文本，适用于大多数情况。 | 'Hello, world!' | Hello, world! |
| 双引号 (") | 用双引号包裹的文本，适用于字符串内需要包含单引号的情况。 | "That's a cat." | That's a cat. |
| 三单引号 ('''') | 用三个连续单引号包裹的文本，可跨多行，保留格式。 | '''Line 1\nLine 2''' | Line 1Line 2 |
| 三双引号 ("""") | 用三个连续双引号包裹的文本，功能同三单引号，可跨多行。 | """Line 1\nLine 2""" | Line 1Line 2 |
| Unicode 编码 | 使用 \u 或 \U 跟随字符的 Unicode 码点。 | '\u4F60\u597D' | 你好 |
| 原始字符串 (r) | 通过前缀 r 定义，不对反斜杠进行转义，适用于文件路径和正则表达式等。 | r"C:\Users\namex" | C:\Users\namex |
| 格式化字符串 (f) | 通过前缀 f 或 F 定义，允许在字符串中嵌入表达式。 | name = "world"
f"Hello, {name}!" | Hello, world! |

● 格式化字符串

f-string（格式化字符串）是 Python 3.6+ 中引入的一种字符串格式化方法，其提供了一种非常快速且直观的方式来嵌入 Python 表达式到字符串常量中。其常用功能如下：

在花括号 {} 中直接引用变量名：

```
1 name = "Alice"
2 age = 30
3 print(f"Name: {name}, Age: {age}")
```

在 {} 中可以包含任何有效的 Python 表达式：

```
1 x = 10
2 print(f"Half of {x} is {x / 2}")
```

可以在 {} 中调用函数：

```
1 age = 20
2 print(f"Age in hexadecimal: {hex{age}}")
```

可以使用冒号：后跟格式指定符来控制数字或字符串的显示格式：

```
1 import datetime
2 today = datetime.datetime.now()
3
```

```

4 # 格式化日期
5 print(f"Today's date: {today:%Y-%m-%d}")
6
7 # 浮点数精度
8 pi = 3.14159265
9 print(f"Pi rounded to two decimal places: {pi:.2f}") # 保留两位小数
10
11 # 填充和对齐
12 number = 123
13 print(f"Number aligned right (width 10): {number:>10}") # :< 左对齐 := 居中对齐
14 print(f'{number:5}') # 默认空格填充, 宽度为 5
15 print(f'{number:05}') # 用 0 填充, 宽度为 5, 输出 00123
16
17 # 大数格式化
18 large_number = 123456789
19 print(f"{large_number:,}") # 输出 123,456,789

```

如果需要在 f-string 中显示花括号本身, 需要对其进行双重编码:

```

1 print(f"Curly braces: {{ and }}")

```

D.2 字符串中字符的访问 字符串是字符序列。每个字符在字符串中有一个特定的位置, 即索引, 从而可以通过索引来访问字符串中的单个字符。下面详细介绍这种访问方式:

• 索引访问

1. 正索引: 从字符串的开头开始, 第一个字符的索引是 0, 第二个字符的索引是 1, 依此类推。
2. 负索引: 从字符串的末尾开始逆向计数, 最后一个字符的索引是 -1, 倒数第二个字符的索引是 -2, 依此类推。

假设我们有一个字符串 `s = "Hello, world!"`:

| | | | | | | | | | | | | |
|-----|-----|-----|-----|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| H | e | l | l | o | , | | w | o | r | l | d | ! |
| -13 | -12 | -11 | -10 | -9 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 |

图 3.1: 字符串的字符索引

```
1 s = "Hello, world!"
2
3 # 使用正索引访问
4 print(s[0])    # 输出 'H'
5 print(s[7])    # 输出 'w'
6
7 # 使用负索引访问
8 print(s[-1])   # 输出 '!'
9 print(s[-2])   # 输出 'd'
```

• 切片访问

可以使用切片来访问字符串中的一段字符子串。切片语法格式为 `s[start:stop:step]`，其中：

- **start** 是切片开始的索引（包含该索引位置的字符）；
- **stop** 是切片结束的索引（不包含该索引位置的字符）；
- **step** 是步长，用来指定取值间隔，默认为 **1**（每个字符都取）。

```
1 s = "Hello, world!"
2
3 # 从索引 1 开始到索引 5 停止
4 print(s[1:5])  # 输出 'ello'
5
6 # 从开头到索引 5（不包含索引 5）
7 print(s[:5])   # 输出 'Hello'
8
9 # 从索引 7 开始到末尾
10 print(s[7:])   # 输出 'world!'
11
12 # 使用步长跳跃取值
13 print(s[0:12:2]) # 输出 'Hlo ol'
14
15 # 从末尾开始逆序取值
16 print(s[::-1])  # 输出 '!dlrow ,olleH'
```

• 单字符访问和切片的使用途

- **单字符访问**：适用于当你需要从字符串中获取某个特定位置的字符时。
- **切片访问**：适用于需要从字符串中提取子串或者需要按一定规律跳跃取字符的情况。

D.3 字符串函数 字符串具有一系列的函数，每个函数都对应特别的作用或功能。如下按照函数的功能类别对常用的字符串函数进行简要介绍：

• 字符判别

| 方法 | 描述 |
|------------------------|-------------------|
| <code>isalpha()</code> | 检查字符串是否只包含字母。 |
| <code>isdigit()</code> | 检查字符串是否只包含数字。 |
| <code>isalnum()</code> | 检查字符串是否只由字母和数字组成。 |
| <code>isspace()</code> | 检查字符串是否只包含空白字符。 |
| <code>islower()</code> | 检查字符串中的字母是否都是小写。 |
| <code>isupper()</code> | 检查字符串中的字母是否都是大写。 |

```
1 # 示例字符串
2 s = "Hello123"
3
4 # 使用 isalpha() 检查字符串是否只包含字母
5 print(s.isalpha()) # 输出: False
6
7 # 使用 isalnum() 检查字符串是否只包含字母和数字
8 print(s.isalnum()) # 输出: True
```

• 字符检索

| 方法 | 描述 |
|---------------------------|--|
| <code>find()</code> | 返回子字符串在字符串中首次出现的索引，如果没有找到则返回 -1。 |
| <code>index()</code> | 与 <code>find()</code> 类似，但如果未找到子字符串则会引发异常。 |
| <code>startswith()</code> | 检查字符串是否以指定的前缀开始。 |
| <code>endswith()</code> | 检查字符串是否以指定的后缀结束。 |
| <code>count()</code> | 返回子字符串在字符串中出现的次数。 |
| <code>in</code> | 返回字符串是否存在某个子串， <code>'ab' in 'deabc'</code> 返回 <code>True</code> |

```
1 # 示例字符串
2 s = "Hello, world!"
3
4 # 使用 find() 查找子字符串的位置
5 print(s.find("world")) # 输出: 7
6
7 # 使用 startswith() 检查字符串是否以指定的前缀开始
8 print(s.startswith("Hello")) # 输出: True
```

• 字符修改

| 方法 | 描述 |
|---------------------------|----------------------------------|
| <code>replace()</code> | 返回一个字符串，其中指定的子字符串被替换为另一个指定的子字符串。 |
| <code>strip()</code> | 从字符串的开始和结尾处删除所有空白字符（或其他指定字符）。 |
| <code>rstrip()</code> | 删除字符串尾部的空白字符（或其他指定字符）。 |
| <code>lstrip()</code> | 删除字符串开头的空白字符（或其他指定字符）。 |
| <code>upper()</code> | 将字符串中的所有字母转换为大写。 |
| <code>lower()</code> | 将字符串中的所有字母转换为小写。 |
| <code>capitalize()</code> | 将字符串的第一个字母变为大写，其余为小写。 |
| <code>title()</code> | 将字符串每个单词的首字母大写。 |

```
1  # 示例字符串
2  s = "hello world"
3
4  # 使用 replace() 替换字符串中的子字符串
5  print(s.replace("world", "there"))  # 输出: "hello there"
6
7  # 使用 upper() 将字符串中的所有字母转换为大写
8  print(s.upper())  # 输出: "HELLO WORLD"
```

• 字符串拆分

| 方法 | 描述 |
|---------------------------|--|
| <code>split()</code> | 根据指定的分隔符将字符串分割成一个列表。如果没有指定分隔符，则默认按空白字符（如空格、换行等）进行分割。 |
| <code>rsplit()</code> | 类似于 <code>split()</code> ，但是从字符串的末尾开始分割。 |
| <code>splitlines()</code> | 根据换行符 (<code>\n</code>) 来分割字符串，返回一个包含各行作为元素的列表。 |

```
1  # 示例字符串
2  s = "apple, banana, cherry"
3
4  # 使用 split() 根据逗号和空格拆分字符串
5  parts = s.split(", ")
6  print(parts)  # 输出: ['apple', 'banana', 'cherry']
```

• 字符串合并

| 方法 | 描述 |
|-----------------------|---|
| <code>join()</code> | 通过指定的字符串（通常称为”分隔符”），将一个字符串列表或元组中的元素合并成一个新的字符串。 |
| <code>+</code> 操作符 | 可以用来连接两个或多个字符串，简单直接。 |
| <code>format()</code> | 提供了一种格式化字符串的方式，可以在字符串中插入一个或多个占位符，然后将它们替换为相应的值，虽然主要用于格式化，但也可用于合并字符串。 |

```
1 # 字符串列表
2 parts = ['apple', 'banana', 'cherry']
3
4 # 使用 join() 将字符串列表合并为一个新的字符串，以逗号和空格作为分隔符
5 combined = ", ".join(parts)
6 print(combined) # 输出: "apple, banana, cherry"
```

更多的有关字符串的函数可以参考📖官网文档。

3.3.2 简单数据类型的相互转换

简单数据类型之间的相互转换，主要是通过个数据类型对应的构造函数来实现。想转换为哪种数据类型，就调用其对应的构造函数即可。如：

- 整数: `int(x)`
- 浮点数: `float(x)`
- 字符串: `str(x)`
- 布尔型: `bool(x)`

```
1 # 整数和浮点数之间的转换
2 int_value = 67
3 float_value = float(int_value) # 将整数转换为浮点数
4 print(" 整数转浮点数:", float_value)
5
6 float_value = 3.14159
7 int_value = int(float_value) # 将浮点数转换为整数（丢弃小数部分）
8 print(" 浮点数转整数:", int_value)
9
10 # 整数、浮点数和字符串之间的转换
11 str_value = str(int_value) # 将整数转换为字符串
12 print(" 整数转字符串:", str_value)
13
```

```
14 str_value = "123.456"
15 float_value = float(str_value) # 将字符串转换为浮点数
16 print(" 字符串转浮点数:", float_value)
17
18 int_from_str = int(float(str_value)) # 字符串转浮点数再转整数
19 print(" 字符串通过浮点数转整数:", int_from_str)
20
21 # 布尔型与其他类型的转换
22 bool_value = bool(int_from_str) # 使用非零整数转换为布尔值 True
23 print(" 非零整数转布尔值:", bool_value)
24
25 bool_value = bool(0) # 0 转换为布尔值 False
26 print("0 转布尔值:", bool_value)
27
28 int_from_bool = int(True) # 将 True 转换为整数
29 print(" 布尔 True 转整数:", int_from_bool)
30
31 str_from_bool = str(False) # 将 False 转换为字符串
32 print(" 布尔 False 转字符串:", str_from_bool)
```

练习

1. 编写程序，根据用户输入 x 值，计算多项式值 $5x^3 - 3.2x^2 + 2x - 7.5$ 的值。
2. 编写程序，用于判断学生是否符合某特定项目的资格。该项目要求参与学生必须满足以下条件之一：

- 学生的平均成绩至少为 85 分以上，且没有挂科（即所有科目成绩都必须大于等于 60 分）。
- 学生是学生会的成员，且至少参加过 2 次学生会组织的活动。

用户输入学生的平均成绩，是否有挂科的科目，是否是学生会成员，以及参加的活动次数。根据输入的数据，程序判断学生是否符合参加项目的条件。

3. 编写程序，对用户将输入一个长句子进行以下操作：

1. 将句子转换为全大写和全小写。
2. 统计句子中单词的数量。
3. 替换句子中的某个单词。
4. 检查句子是否以特定单词开始或结束。
5. 提取句子中的某个片段。

3.4 复合数据类型

复合数据类型通常被描述为数据容器，因为它们可以包含、存储和组织多个数据项。复合数据类型主要包括列表、元组、字典和集合，每种类型都有其独特的特性和用途。

3.4.1 列表 (List)

字符串是字符组成的有序序列，列表是由各种数据组成的有序、可变数据容器。

A 特征:

- **有序**: 列表中的元素按照添加顺序进行存储。
- **可变**: 列表中的元素可以被修改。
- **可重复**: 列表可以包含重复的元素。
- **支持多种数据类型**: 列表可以包含不同类型的元素，包括列表和其他的复合数据类型。

B 作用:

- 用于存储一系列相关数据项，常用于数据集合的迭代处理。
- 动态地添加、删除或更改元素。
- 适用于需要经常修改内容的场景。

C 列表的创建和表示

| 表示方式 | 描述 | 示例 |
|----------|---------------------------------|---|
| 空列表 | 不包含任何元素的列表 | <code>[]</code> |
| 混合类型元素列表 | 列表中包含不同类型的元素 | <code>[1, 'hello', 3.14, True]</code> |
| 嵌套列表 | 列表中的部分元素也是列表 | <code>['abc', 5.78, ['Macao', 14]]</code> |
| 使用构造函数 | 使用 <code>list()</code> 构造函数创建列表 | <code>list('hello')</code>
输出 <code>['h', 'e', 'l', 'l', 'o']</code>
<code>list(range(6))</code> # 输出 <code>[0, 1, 2, 3, 4, 5]</code> |

D 列表元素的访问和修改

列表元素的访问方式与字符串一致，都是通过索引和切片来访问元素。只是字符串中的字符不可直接修改，而列表中的元素可以任意修改。

```
1 my_string = 'hello'
2 my_string[0] = 'H'
```

上述代码会产生如下错误：

TypeError: 'str' object does not support item assignment

```
1 nested_list = [
2     [1, 2, 3],
3     [4, 5, 6],
4     [7, 8, 9]
5 ] # 一个 3x3 的矩阵
6 print(nested_list[0][0]) # 输出 1
7 nested_list[1][1] = 500
8 print(nested_list[1][:2]) # 获取第二行的前两个元素，输出 [4, 500]
```

E 列表的操作

| 操作/方法 | 描述 |
|-------------------------------|--|
| append(x) | 在列表末尾添加一个元素 x。 |
| extend(iterable) | 扩展列表，将一个可迭代对象的所有元素逐个添加到列表末尾。 |
| insert(i, x) | 在指定位置 i 插入一个元素 x。 |
| remove(x) | 移除列表中第一个值为 x 的元素。如果没有找到元素 x，则抛出 ValueError。 |
| pop(i) | 移除指定位置的元素（如果没有任何参数，默认是最后一个），并返回该元素的值。 |
| clear() | 移除列表中的所有元素。 |
| index(x, start, end) | 返回列表中第一个值为 x 的元素的索引。如果没有找到元素 x，则抛出 ValueError。start 和 end 都有默认值。 |
| count(x) | 返回 x 在列表中出现的次数。 |
| sort(key=None, reverse=False) | 对列表进行排序。如果指定了 key，则根据 key 的返回值进行排序。 |
| reverse() | 反转列表中元素的排列顺序。 |
| copy() | 返回列表的浅拷贝。 |
| + | 连接两个列表。 |
| * | 重复列表中的元素指定次数。 |
| a, b,*rest = [1, 2,3, 4] | 使用星号（*）变量来收集多余的值。 |

```
1 my_list = [3, 1, 4, 1, 5, 9, 2]
2
3 # 添加元素
4 my_list.append(6)
5 print(my_list) # [3, 1, 4, 1, 5, 9, 2, 6]
6
7 # 扩展列表
8 my_list.extend([7, 8])
9 print(my_list) # [3, 1, 4, 1, 5, 9, 2, 6, 7, 8]
10
11 # 插入元素
12 my_list.insert(0, 0)
13 print(my_list) # [0, 3, 1, 4, 1, 5, 9, 2, 6, 7, 8]
14
15 # 移除元素
16 my_list.remove(1) # 移除第一个出现的 1
17 print(my_list) # [0, 3, 4, 1, 5, 9, 2, 6, 7, 8]
18
19 # 弹出元素
20 last_item = my_list.pop()
21 print(last_item) # 8
22 print(my_list) # [0, 3, 4, 1, 5, 9, 2, 6, 7]
23
24 # 反转列表
25 my_list.reverse()
26 print(my_list) # [7, 6, 2, 9, 5, 1, 4, 3, 0]
27
28 # 排序列表
29 my_list.sort()
30 print(my_list) # [0, 1, 2, 3, 4, 5, 6, 7, 9]
31
32 # + 连接列表
33 list1 = [1, 2, 3]
34 list2 = [4, 5, 6]
35 combined_list = list1 + list2
36 print(combined_list) # 输出: [1, 2, 3, 4, 5, 6]
37
38 # * 重复列表
39 list1 = [1, 2, 3]
40 repeated_list = list1 * 3
41 print(repeated_list) # 输出: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

3.4.2 元组 (Tuple)

元组是一个不可变的有序数据容器。

A 特征：

- **有序**：元组中的元素按照定义时的顺序存储。
- **不可变**：一旦创建，元组中的元素不能被修改。
- **可重复**：元组可以包含重复的元素。
- **支持多种数据类型**：同列表。

B 作用：

- 用于存储不应改变的数据集。
- 通常用于函数返回多个值或将数据传递给函数。
- 将多个数据“打包”为单个实体。

C 元组的创建和表示

| 表示方式 | 描述 | 例子 |
|-----------------------------|---|---|
| 空元组 | 创建一个没有任何元素的空元组。 | <code>my_tuple = ()</code> |
| 单元素元组 | 创建一个只有一个元素的元组，注意元素后面需要加逗号。 | <code>my_tuple = (42,)</code> |
| 多类型元素元组 | 创建一个包含不同类型元素的元组。 | <code>my_tuple = (1, 'hello', 3.14)</code> |
| 不使用括号的元组 | 元组的括号是可选的，不使用括号也可以定义元组。 | <code>my_tuple = 'a', 2, True</code> |
| 使用构造函数 <code>tuple()</code> | 使用 <code>tuple()</code> 构造函数将其他可迭代对象（如列表）转换为元组。 | <code>my_tuple = tuple([1, 'two', True])</code> |

D 元组元素的访问

元组元素的访问与列表完全一致。

E 元组的操作

| 操作类型 | 描述 | 代码示例 |
|------|--------------------|---|
| 连接 | 将两个元组连接起来形成一个新的元组。 | <pre>tuple1 = (1, 2) tuple2 = (3, 4) print(tuple1 + tuple2)</pre> |
| 重复 | 通过重复元组元素来创建新的元组。 | <pre>my_tuple = ('repeat',) print(my_tuple * 3)</pre> |
| 成员检查 | 检查特定元素是否存在于元组中。 | <pre>my_tuple = (1, 2, 3) print(2 in my_tuple)</pre> |
| 计数 | 计算某个元素在元组中出现的次数。 | <pre>my_tuple = (1, 1, 2, 3) print(my_tuple.count(1))</pre> |
| 索引查找 | 找出特定元素在元组中的索引位置。 | <pre>my_tuple = (1, 2, 3) print(my_tuple.index(2))</pre> |
| 长度 | 获取元组中元素的数量。 | <pre>my_tuple = (1, 2, 3) print(len(my_tuple))</pre> |

3.4.3 字典 (Dictionary)

字典是一个由键值对组成的数据容器。字典和列表是 Python 中用来存储数据的两种非常常用的数据结构，但它们各自的特点和适用场景有所不同。我们可以通过一个简单的例子——学生成绩的存储，来理解它们之间的区别。

列表是一个有序的数据集合，其中的元素按照添加的顺序排列。如果我们用列表来存储学生成绩，我们可能会这样做：

```
1 | students_scores = [90, 82, 88, 95]
```

在这个例子中，每个分数都存储在列表中一个特定的位置（或称为索引）。例如，第一个学生的分数是 90，第二个学生的分数是 82，依此类推。但这里有一个问题：如果我们只看这个列表，实际上我们并不知道每个分数对应哪个学生，除非我们有额外的方式来记录每个索引对应的学生信息。

字典是一种存储键值对 (key-value pairs) 的数据结构，每个键对应一个唯一的值。如果用字典来存储学生成绩，我们可以这样做：

```
1 | students_scores = {
2 |     "Alice": 90,
3 |     "Bob": 82,
4 |     "Charlie": 88,
5 |     "Diana": 95
6 | }
```

在这个字典中，每个学生的名字是一个键（**key**），对应的分数是值（**value**）。这种方式让我们可以直接通过学生的名字来查找或修改他们的分数，非常直观和方便。例如，如果我们想知道 **Alice** 的分数，只需要查找 `students_scores["Alice"]`。

A 特征：

- **无序**：虽然 **Python 3.7+** 中字典维持插入顺序，但一般认为它是无序的。
- **键值对**：数据以键值对形式存储，键必须是不可变类型，值可以是任何类型。
- **键唯一**：每个键必须是唯一的，同一个键的多个赋值会覆盖之前的值。

B 作用：

- 用于存储和管理相关联的数据项（如对象的属性和值）。
- 快速检索（基于键的查找）。
- 适用于键值对映射。

C 字典的创建和表示

| 表示方式 | 描述 | 示例代码 |
|-----------------------------|--|--|
| 空字典 | 创建一个没有任何键值对的空字典。 | <code>my_dict = {}</code> 或 <code>my_dict = dict()</code> |
| 使用大括号 | 直接使用大括号和键值对创建字典。 | <code>my_dict = {'a': 1, 'b': 2}</code> |
| 使用 <code>dict()</code> 构造函数 | 通过键值对序列使用 <code>dict()</code> 创建字典。 | <code>my_dict = dict([('a', 1), ('b', 2)])</code> |
| 关键字参数 | 使用关键字参数传递给 <code>dict()</code> 。 | <code>my_dict = dict(a=1, b=2)</code> |
| 从键列表创建 | 使用 <code>fromkeys()</code> 为多个键设置相同的初始值。 | <code>keys = ['a', 'b']</code>
<code>my_dict = dict.fromkeys(keys, 1)</code> |
| 使用 <code>zip()</code> | 使用 <code>zip()</code> 结合两个列表或元组，生成键值对创建字典。 | <code>keys = ['a', 'b']</code>
<code>values = [1, 2]</code>
<code>my_dict = dict(zip(keys, values))</code> |

D 字典元素的访问

访问字典元素主要通过使用键（**key**）进行。字典是一个键值对（**key-value pairs**）的集合，每个键映射到一个特定的值。

访问字典元素的几种常用方法：

• 直接通过键访问

使用方括号 `[]` 并提供键名来直接访问字典中的值。如果键不存在于字典中，这种方法会引发一个 `KeyError`。

```
1 my_dict = {'name': 'Alice', 'age': 25}
2 print(my_dict['name']) # 输出: Alice
3 ## print(my_dict['gender']) # 如果键 'gender' 不存在, 这将引发 KeyError
```

• 使用 `get()` 方法

`get()` 方法提供了一种访问字典元素的方式，其中可以设置默认值，如果指定的键不存在于字典中，不会引发错误，而是返回默认值。

```
1 my_dict = {'name': 'Alice', 'age': 25}
2 print(my_dict.get('age')) # 输出: 25
3 print(my_dict.get('gender', 'Not specified')) # 输出: Not specified
```

E 字典的操作

| 操作/函数 | 描述 |
|--|---|
| <code>dict.keys()</code> | 返回一个视图，包含字典中的所有键。 |
| <code>dict.values()</code> | 返回一个视图，包含字典中的所有值。 |
| <code>dict.items()</code> | 返回一个视图，包含字典中的所有键值对（元组形式）。 |
| <code>dict.update(other)</code> | 用另一个字典 <code>other</code> 或键值对的可迭代对象更新当前字典。 |
| <code>dict.pop(key, default)</code> | 删除指定键 <code>key</code> 并返回其值，如果键不存在则返回 <code>default</code> 。 |
| <code>dict.popitem()</code> | 随机删除并返回一个字典键值对。如果字典为空，引发 <code>KeyError</code> 。 |
| <code>del dict[key]</code> | 删除指定键（ <code>key</code> ）及其对应的值。 |
| <code>dict.clear()</code> | 清空字典，移除所有项。 |
| <code>dict.copy()</code> | 返回字典的一个浅拷贝。 |
| <code>len(dict)</code> | 返回字典中键值对的数量。 |
| <code>key in dict</code> | 检查键（ <code>key</code> ）是否存在于字典中。 |
| <code>dict.setdefault(key, default)</code> | 如果键 <code>key</code> 存在于字典中，返回对应的值；如果不存在，插入键值对 <code>key: default</code> 并返回 <code>default</code> 。 |

```
1 # 创建并初始化一个字典
2 my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}
3
4 # 删除指定键及其对应的值
5 del my_dict['city'] # 删除键 'city'
6 print("After deletion:", my_dict)
```

```
7
8 # 返回字典中所有键
9 keys = my_dict.keys()
10 print("Keys:", list(keys))
11
12 # 返回字典中所有值
13 values = my_dict.values()
14 print("Values:", list(values))
15
16 # 返回字典中所有键值对（元组形式）
17 items = my_dict.items()
18 print("Items:", list(items))
19
20 # 解包字典中的所有键和值
21 keys, values = my_dict.items()
22 print(keys, values)
23
24 # 用另一个字典更新当前字典
25 my_dict.update({'age': 30, 'gender': 'female'})
26 print("After update:", my_dict)
27
28 # 删除指定键并返回其值，如果键不存在则返回默认值
29 age = my_dict.pop('age', None)
30 print("Popped age:", age)
31 print("After pop:", my_dict)
32
33 # 清空字典
34 my_dict.clear()
35 print("After clear:", my_dict)
36
37 # 重新初始化字典，以便继续操作
38 my_dict = {'name': 'Bob', 'age': 22}
39
40 # 返回字典的一个浅拷贝
41 new_dict = my_dict.copy()
42 print("Copied dictionary:", new_dict)
43
44 # 返回字典中键值对的数量
45 length = len(my_dict)
46 print("Length of dictionary:", length)
47
48 # 检查键是否存在于字典中
49 exists = 'name' in my_dict
50 print("Is 'name' in dictionary?", exists)
51
52 # 如果键存在于字典中，返回对应的值；如果不存在，插入键值对并返回默认值
```

```
53 status = my_dict.setdefault('status', 'single')
54 print("Set default:", status)
55 print("After setdefault:", my_dict)
```

3.4.4 集合 (Set)

集合是一个无序的数据容器，用于存储互不相同的元素。

A 特征:

- 无序：集合中的元素没有固定顺序。
- 无重复元素：集合自动去除重复元素。
- 可变：可以添加或删除元素。

B 作用:

- 用于存储无重复项的数据集。
- 高效地进行数学运算，如并集、交集、差集等。
- 过滤数据以消除重复项。#### 集合的创建和表示

| 方法 | 描述 |
|---|-------------------------------------|
| <code>set()</code> | 创建一个空集合。 |
| <code>{element1, element2, ...}</code> | 使用花括号直接创建一个包含若干元素的集合。 |
| <code>set([element1, element2, ...])</code> | 通过一个列表或任何可迭代对象来创建一个集合。 |
| <code>set(range(start, stop, step))</code> | 使用 <code>range()</code> 函数生成一个数值集合。 |

C 集合元素的访问

在 Python 中，集合 (set) 是一个无序的数据结构，这意味着它不支持索引、切片或其他序列操作。因此，不能像列表或元组那样直接通过索引来访问集合中的元素，也不能通过像字典那样用键值访问。只可以使用 `in` 和 `not in` 来检查一个元素是否存在于集合中。

```
1 my_set = {1, 2, 3, 4}
2 3 in my_set # 输出 True
```

D 集合的操作

| 操作 | 描述 |
|---|---|
| <code>set.add(elem)</code> | 向集合添加一个元素。 |
| <code>set.update(other_set)</code> | 将 <code>other_set</code> 中的元素添加到调用它的集合中，直接修改原集合。 |
| <code>set.remove(elem), set.discard(elem)</code> | 从集合中移除一个元素。 <code>remove()</code> 在元素不存在时会抛出错误，而 <code>discard()</code> 不会。 |
| <code>set.pop()</code> | 随机移除并返回集合中的一个元素。如果集合为空，抛出 <code>KeyError</code> 。 |
| <code>set.clear()</code> | 清空集合中的所有元素。 |
| <code>len(set)</code> | 返回集合中元素的数量。 |
| <code>elem in set, elem not in set</code> | 检查元素是否存在于集合中。 |
| <code>set.copy()</code> | 返回集合的一个浅拷贝。 |
| <code>set.union(other_set), set other_set</code> | 返回两个集合的并集。 |
| <code>set.intersection(other_set), set & other_set</code> | 返回两个集合的交集。 |
| <code>set.difference(other_set), set - other_set</code> | 返回两个集合的差集（在第一个集合中但不在第二个集合中的元素）。 |
| <code>set.symmetric_difference(other_set), set ^ other_set</code> | 返回两个集合的对称差集（在一个集合中但不在两个集合的交集的元素）。 |
| <code>set.issubset(other_set), set <= other_set</code> | 检查当前集合是否是另一个集合的子集。 |
| <code>set.issuperset(other_set), set < other_set</code> | 检查当前集合是否是另一个集合的超集。 |
| <code>set.isdisjoint(other_set)</code> | 检查两个集合是否没有交集。 |

```
1  # 创建集合
2  set1 = {1, 2, 3}
3  set2 = {3, 4, 5}
4
5  # 添加元素
6  set1.add(6)
7  print("After add:", set1)
8
9  # 更新集合
10 set1.update(set2)
11 print("After update:", set1)
12
13 # 删除元素
14 set1.remove(6) # 如果元素不存在会抛出 KeyError
15 print("After remove:", set1)
16 set1.discard(10) # 如果元素不存在，不会抛出错误
17 print("After discard:", set1)
```

```
18
19 # 随机移除元素
20 removed_element = set1.pop()
21 print("After pop:", set1, "Removed:", removed_count)
22
23 # 清空集合
24 set1.clear()
25 print("After clear:", set1)
26
27 # 获取集合的长度
28 print("Length:", len(set2))
29
30 # 检查元素是否存在
31 print("3 in set2:", 3 in set2)
32
33 ## 集合的拷贝
34 set3 = set2.copy()
35 print("Copied set3:", set3)
36
37 # 集合的并集
38 union_set = set1.union(set2)
39 print("Union of set1 and set2:", union_set)
40
41 # 集合的交集
42 intersection_set = set1.intersection(set2)
43 print("Intersection of set1 and set2:", intersection, set2)
44
45 # 集合的差集
46 difference_set = set1.difference(set2)
47 print("Difference of set1 from set2:", difference_set)
48
49 # 集合的对称差集
50 symmetric_difference_set = set1.symmetric_difference(set2)
51 print("Symmetric difference between set1 and set2:", symmetric_difference_set)
52
53 # 子集和超集检查
54 print("set2 is subset of set3:", set2.issubset(set3))
55 print("set3 is superset of set2:", set3.issuperset(set2))
56
57 # 检查两个集合是否无交集
58 print("set1 and set2 are disjoint:", set1.isdisjoint(set2))
```

3.4.5 复合数据类型的相互转换

在 Python 中，复合数据类型的互相转换是常见的需求，尤其是在列表（list）、元组（tuple）、集合（set）和字典（dict）之间进行转换。转换的主要方式是调用各个复合数据类型的构造函数。如列表 `list(x)`，元组 `tuple(x)`，集合 `set(x)`，字典 `dict(zip(keys, values))`，其中 `x`，`keys`，`values` 为可迭代的复合数据类型。但要注意，`keys` 中元素为不可更改的对象。

```
1 list_example = [1, 2, 2, 3, 4]
2
3 # 列表转元组
4 tuple_from_list = tuple(list_example)
5 print(" 元组:", tuple_from_list)
6
7 # 列表转集合（自动删除重复元素）
8 set_from_list = set(list_example)
9 print(" 集合:", set_from_list)
10
11 # 集合转列表
12 list_from_set = list(set_from_list)
13 print(" 从集合到列表:", list_from_set)
14
15 # 集合转元组
16 tuple_from_set = tuple(set_from_list)
17 print(" 从集合到元组:", tuple_from_set)
18
19 # 定义键和值
20 keys = ["apple", "banana", "cherry"]
21 values = [100, 200, 300]
22
23 # 创建字典
24 dictionary = dict(zip(keys, values))
25 print(" 字典:", dictionary)
26
27 # 字典转列表（提取键）
28 list_keys = list(dictionary.keys())
29 print(" 字典键的列表:", list_keys)
30
31 # 字典转元组（提取值）
32 tuple_values = tuple(dictionary.values())
33 print(" 字典值的元组:", tuple_values)
34
35 # 字典转集合（提取键）
36 set_keys = set(dictionary.keys())
```

37 | `print(" 字典键的集合:", set_keys)`

练习

1. 创建一个列表 `stocks`，包含至少 5 种不同的股票名称。
 1. 打印这个列表的第一个和最后一个元素。
 2. 添加两支新股票。
 3. 删除列表 `stocks` 第二位置的股票。
 4. 打印列表中股票的数量。
2. 创建两个变量，`name` 和 `age`，并将它们打包成一个元组 `person`。
 1. 解包 `person` 元组到变量 `unpacked_name` 和 `unpacked_age`。
 2. 打印解包后的变量以确认它们的值。
3. 创建一个字典 `course_students`，键是课程名称，值是选修该课程学生姓名的列表。添加至少三个课程和相应的学生名单。
 1. 为其中一个课程添加一个新学生
 2. 任选一门课程，打印其选课学生的列表。
4. 创建两个集合 `setA` 包含数字 1, 2, 3, 4 和 `setB` 包含数字 3, 4, 5, 6。
 1. 打印 `setA` 和 `setB` 的并集。
 2. 打印 `setA` 和 `setB` 的交集，试用 `update` 和并集运算两种方法，了解两者的差别。
 3. 打印 `setA` 对 `setB` 的差集。
 4. 打印 `setA` 和 `setB` 的对称差集。

4

状态变化控制

根据图灵机的理论，任何可以被算法描述的计算都可以通过图灵机来完成，而图灵机的操作可以通过使用循环和条件判断来模拟。这意味着所有的计算任务都可以通过这两种控制结构来实现。本章主要介绍条件判断语句和循环语句的语法及其使用方法，此外还介绍这两种控制语句与复杂数据类型结合以构造复杂数据类型的生成器。

4.1 条件判断语句

在计算机程序设计中，条件判断语句是非常基础也非常重要。它们允许程序根据不同的情况执行不同的代码，使程序具有决策能力。

4.1.1 作用

1. **决策制定**：条件判断语句让计算机能够根据指定条件（如比较两个数的大小、检查某个状态是否为真等）来选择不同的执行路径。这类似于日常生活中的决策，例如：“如果外面在下雨，我就带伞出门，否则不带。”
2. **流程控制**：它们帮助控制程序的执行流程，确保在适当的条件下执行适当的代码块。这对于创建反应灵敏和适应性强的应用程序至关重要。
3. **错误处理**：在发生特定错误或异常情况时，条件判断可以引导程序执行错误处理代码，避免程序崩溃或产生不正确的结果。

4. 功能分支：在复杂的应用中，根据用户的选择或其他运行时条件，条件判断可以引导程序执行不同的功能。

4.1.2 特点

1. 简单性：条件判断语句通常结构简单，易于理解和实现。基本的结构包括“如果（某条件成立），那么（执行某操作）；否则（执行另一操作）”。
2. 灵活性：可以根据需求嵌套多个条件判断，形成更复杂的决策树结构，以处理更复杂的情况。
3. 普遍性：几乎所有的编程语言都支持某种形式的条件判断语句，如 `if`、`else` 等。
4. 效率：条件判断语句通常对程序的运行效率影响很小，但是不恰当的过多嵌套或复杂的条件表达式可能会降低程序的执行速度和代码的可读性。

4.1.3 语法规则

A `if` 语句

`if` 语句是最基本的条件判断语句。它会执行一个条件检查，如果条件为真 (True)，则执行随后的代码块。

在 **Python** 中，缩进是一种用于定义代码块的语法元素，这与许多其他编程语言中定义代码块的方式不同。正确的缩进对于 **Python** 代码的正确执行至关重要，因为它直接影响到代码的结构和逻辑判断。

• 缩进的基本规则

1. 一致性：在一个代码块中，每一行代码必须具有相同数量的前置空白（通常是空格或制表符）。通常推荐使用 4 个空格来进行缩进，这是 **Python** 常用标准。
2. 层级结构：当你编写嵌套的代码块时（例如，在 `if` 语句内部嵌套另一个 `if` 语句），内部的代码块需要比外部的代码块缩进多一层级的空格。这样可以清晰地显示代码的层级关系。
3. 缩进的统一性：在同一个 **Python** 文件中，应当统一使用空格或制表符进行缩进，而不应混用。多数现代文本编辑器和集成开发环境 (IDE) 如 **PyCharm**、**VSCode** 等，都支持自动缩进和缩进级别的可视化显示，帮助

开发者保持缩进的一致性。这些工具通常也允许开发者设置缩进的大小（例如设置每级缩进为 4 个空格）。

- 基本语法

```
1 | if 条件表达式:  
2 |     # 条件为真时，执行的代码块，在此条件下执行的代码块，均需缩进，如下示例所示。
```

- 示例

```
1 | age = 20  
2 | sex = 'male'  
3 | if age >= 18:  
4 |     if sex == 'male':  
5 |         print(" 成年的男性")
```

B else 语句

`else` 语句与 `if` 语句搭配使用，用于定义当 `if` 的条件不为真时执行的代码块。

- 基本语法:

```
if 条件表达式 :  
    # 条件为真时，执行的代码块  
else:  
    # 条件为假时，执行的代码块
```

- 示例:

```
1 | age = 16  
2 | if age >= 18:  
3 |     print(" 你已经成年了。")  
4 | else:  
5 |     print(" 你还未成年。")
```

C elif 语句

`elif` 是 `else if` 的缩写，用于在多个条件之间进行选择。它必须跟在 `if` 或另一个 `elif` 语句之后，可以有多个 `elif` 语句。

- 基本语法:

```
1 | if 条件表达式1:
2 |     # 条件 1 为真时, 执行的代码块
3 | elif 条件表达式2:
4 |     # 条件 1 为假且条件 2 为真时, 执行的代码块
5 | else:
6 |     # 前面所有条件都为假时, 执行的代码块
```

• 示例:

```
1 | age = 65
2 | if age < 18:
3 |     print(" 你还未成年。")
4 | elif age < 60:
5 |     print(" 你是成年人。")
6 | else:
7 |     print(" 你还是正当年。")
```

D 注意事项

1. 条件表达式的结果必须是布尔值 (True 或 False)。可以使用关系运算符 (如 `==`, `!=`, `<`, `<=`, `>`, `>=`, `in`, `not in`) 和逻辑运算符 (如 `and`, `or`, `not`) 来构建这些表达式。条件表达式中也可以出现某种数据类型的常量或变量, 只要这种数据类型内在支持对布尔型的转换。在 **Python** 中, 几乎所有的常用的数据类型都可以在条件表达式中使用, 因为 **Python** 内部会自动将它们转换成布尔值。这种转换遵循一些基本的规则:

- 数字类型: 0 被认为是 **False**, 所有非零值被认为是 **True**。
- 字符串类型: 空字符串 `""` 被认为是 **False**, 非空字符串被认为是 **True**。
- 列表、元组、字典和集合: 空的被认为是 **False**, 非空的被认为是 **True**。
- **None** 类型: 总是被认为是 **False**。**None** 在 **Python** 中是一个非常有用的特殊数据类型, 用于表示缺失值或无值的情况。使用 **None** 来初始化变量是常见的做法, 尤其是在之后需要检查该变量是否已被赋予了其他值的情况下。

2. 缩进非常重要 **: **Python** 使用缩进来定义代码块的范围。`if`, `elif`, 和 `else` 后面的代码块必须缩进。

4.1.4 三元条件表达式

三元条件表达式的主要作用是提供一种更加简洁和可读的方式来执行条件操作，尤其是在需要根据条件快速选择值的情况。这种表达式通常用于：

- 简化代码，减少需要编写的行数。
- 在赋值操作中直接使用条件逻辑，而无需编写完整的 `if-else` 语句。

A 语法

```
1 | a if condition else b
```

在这里，`condition` 是一个布尔表达式，`a` 和 `b` 是当条件为真时和假时要返回的值。如果 `condition` 为真，表达式的结果是 `a`，否则结果是 `b`。

B 示例

```
1 | ## 基于条件选择数字
2 | num1 = 10
3 | num2 = 20
4 | max_num = num1 if num1 > num2 else num2
5 | print(max_num) # 输出: 20
6 |
7 | ## 根据年龄决定输出
8 | age = 18
9 | status = " 成年人" if age >= 18 else " 未成年"
10 | print(status) # 输出: 成年人
```

练习

1. 编写一个程序，计算任意输入数 `x` 的绝对值。如果输入的为-5，打印结果如下：
`|-5| = 5`。
2. 编写一个程序，根据输入的学生成绩，打印其等级。成绩 `>=90` 为'A'级，`80<= 成绩 <90` 为'B'级，`70<= 成绩 <80` 为'C'级，`60<= 成绩 <70` 为'D'级，成绩 `<60` 为'F'级。
3. 编写一个程序，模拟交通信号灯。用户输入红色、黄色或绿色，根据输入提示相应的行动指示：红灯停，绿灯行，黄灯请注意。
4. 编写一个程序来计算个人所得税。假设税率按以下分级进行计算：

- 收入不超过 50,000 美元的部分，税率为 0%。
- 收入超过 50,000 美元至 100,000 美元的部分，税率为 10%。
- 收入超过 100,000 美元至 150,000 美元的部分，税率为 15%。
- 收入超过 150,000 美元的部分，税率为 20%。

要求用户输入其年收入，然后打印输出收入和应缴纳的税款的信息。

4.2 循环语句

在计算机编程中，循环语句是一种常用的重要控制结构，它允许我们重复执行一段代码多次。这不仅使得程序更简洁，也让程序设计更有效率。想象一下，如果你需要执行相同的任务多次，比如分析股票指数的所有成分股，或者检查电脑上每张照片文件的时间，手动重复这些任务会非常繁琐和容易出错。循环语句就是为了解决这个问题而设计的。

循环语句非常重要，主要原因包括：

- **效率**：循环允许我们高效地执行重复任务，而不需要编写大量重复的代码。这使得程序更简洁、更易于维护和理解。
- **自动化**：循环可以自动处理重复的任务，例如数据处理（如排序、搜索、统计数据等）、文件操作（读取多个文件中的数据）、网络请求（比如，检查多个网页的响应）等。
- **控制流**：循环提供了控制程序执行流的方式。通过循环，程序可以根据需要多次执行某些操作，直到满足特定条件。
- **资源利用**：在处理大量数据时，循环使得计算机能够充分利用其处理能力，通过批量处理操作来优化性能。

4.2.1 种类

通常在大多数编程语言中，包括 Python，有如下两种基本的循环语句：

- **for 循环**：当你知道需要重复执行代码的确切次数时，for 循环非常有用。比如，“为每个学生分发一本书”，如果你有 30 个学生，你可以使用 for 循环来重复这个分发动作 30 次。

- **while 循环**：当你不确定需要重复执行代码多少次，但是知道重复的条件时，**while** 循环很有用。比如，“一直分发书籍，直到所有学生都有书为止”。

A 作用

- **重复执行任务**：循环最基本的作用就是重复执行某些任务。例如，如果你要编写一个程序来读取 **100** 个学生的成绩并计算平均分，使用循环可以让你不必为每个学生编写单独的读取和计算代码，而是编写一次，让循环自动重复执行 **100** 次。
- **处理集合数据**：在处理数组、列表或任何集合数据时，循环可以遍历这些集中的每个元素。例如，你可能需要检查一个列表中的所有项目，看看是否有任何项目满足特定条件，或者对它们进行排序或修改。
- **简化代码，避免冗余**：循环可以大大减少必须写的代码量。没有循环，你可能需要复制粘贴相同的代码多次，这不仅使得代码更长、更难以读写，而且如果需要修改逻辑，你可能需要在多个地方进行更改，容易出错。

B 特点

1. **初始化**：在使用循环之前，通常需要设置一个或多个初始条件。例如，在 **for** 循环中，你通常会初始化一个计数器，而在 **while** 循环中，你可能会设置一个条件判断的初始状态。
2. **条件检查**：循环的核心是条件检查，这决定了循环是否继续执行。如果条件为真，循环体中的代码将被执行。一旦条件不再满足（变为假），循环停止。
3. **更新状态**：在循环的每次执行中，通常需要更新某些状态，以便最终达到结束循环的条件。在 **for** 循环中，这通常是增加或减少计数器；在 **while** 循环中，这可能是更新一个或多个变量的值。
4. **灵活性与控制**：循环提供了极大的灵活性。你可以使用 **break** 语句提前退出循环，使用 **continue** 语句跳过当前循环的剩余部分直接进入下一次循环，这使得循环的控制更加精细和灵活。
5. **潜在的无限循环**：如果循环的条件永远不会变为假，那么循环将永远继续执行，这被称为无限循环（死循环）。在设计循环时，必须小心确保每个循环都有一个清晰的结束条件，以避免程序陷入无限循环。

总之，循环语句是编程中解决问题的强大工具，它们帮助程序员以简洁的方式处理重复任务和大量数据。

4.2.2 while 循环

while 循环是一种基本的循环语句，它用于重复执行一组语句，直到指定的条件不再为真。**while** 循环非常适合于处理那些我们事先不知道需要循环多少次的情况。

A 基本语法

while 循环的最基本形式如下：

```
1 while 条件表达式：  
2     循环体
```

这里，“条件表达式”是每次循环开始前都会进行评估的条件。如果条件为真 (**True**)，则执行“循环体”中的语句。每执行完一次循环体后，条件表达式会再次被评估。如果条件仍然为真，循环继续执行。这个过程重复进行，直到条件表达式为假 (**False**)，此时循环停止。

B 示例

让我们通过一些例子来深入了解 **while** 循环的使用。

- 示例 1：简单的计数器

假设我们要打印从 1 到 5 的数字。这可以通过如下的 **while** 循环实现：

```
1 count = 1  
2 while count <= 5:  
3     print(count)  
4     count += 1
```

在这个例子中，**count** 是一个计数器，从 1 开始。每次循环，我们打印当前的 **count** 值，然后通过 **count += 1** (等同于 **count = count + 1**) 语句将 **count** 的值增加 1。当 **count** 的值超过 5 时，条件 **count <= 5** 变为假，循环停止。

- 示例 2：使用 **break** 退出循环

有时候，我们需要在满足特定条件时立即退出循环，即使循环条件本身还为真。这可以通过 **break** 语句实现：

```
1 n = 0
2 while True: # 无限循环
3     if n == 3:
4         break # 当 n 等于 3 时退出循环
5     print(n)
6     n += 1
```

这里使用了 `True` 作为条件，创建了一个无限循环。我们通过在 `n` 等于 3 时执行 `break` 语句来退出循环。这个例子打印数字 0 到 2，然后在打印 3 之前停止。

- 示例 3：使用 `continue` 跳过当前迭代

我们有时可能想跳过循环的某次特定迭代。这可以使用 `continue` 语句来完成：

```
n = 0
while n < 5:
    n += 1
    if n == 3:
        continue # 当 n 等于 3 时，跳过当前循环的余下部分
    print(n)
```

这个例子中，当 `n` 等于 3 时，`continue` 语句会被执行，导致循环跳过当前迭代的剩余部分（即不执行 `print(n)`），直接进入下一次循环。请分析输出的结果。

- 示例 4：计算从 1 到 100 的所有整数之和

```
1 # 初始化计数器和累加器
2 count = 1
3 total_sum = 0
4
5 # 使用 while 循环进行累加
6 while count <= 100:
7     total_sum += count # 将当前计数器的值加到总和中
8     count += 1 # 将计数器递增 1
9
10 # 输出结果
11 print("1 到 100 之和是:", total_sum)
```

代码解释：

1. 初始化计数器和累加器：

- `count` 初始化为 1，这是循环的起始值。

- `total_sum` 初始化为 0，用于存储从 1 到 100 的累加和。

2. 循环条件：

- `while count <= 100`：这表示只要 `count` 的值小于或等于 100，循环就会继续执行。这是因为我们需要计算从 1 到 100 的所有数字的和。

3. 循环体：

- `total_sum += count`：这一行代码将当前 `count` 的值加到 `total_sum` 上。这是实现累加的关键步骤。
- `count += 1`：每次循环结束后，`count` 递增 1，为下一次循环迭代准备。

4. 输出结果：

- 循环完成后，`total_sum` 变量将包含从 1 到 100 的所有数字的总和。使用 `print()` 函数输出这个结果。

4.2.3 for 循环

在 Python 中，`for` 循环是用来遍历序列（如列表、元组、字典、集合、字符串等）中的每个元素，并执行代码块的一种控制结构。它比 `while` 循环更直接地适用于对序列的迭代处理。

A 基本语法

`for` 循环的基本语法如下：

```
1 | for 变量 in 序列：  
2 |     循环体  # 可包含 break 和 continue
```

在这里，“变量”代表序列中的一个元素，每次循环迭代时会被赋予序列中的下一个元素的值。“序列”是你要遍历的数据集合。每次循环，“变量”自动更新为序列的下一个值，直到遍历完整个序列。

B 示例

让我们通过一些例子来深入了解 `for` 循环的使用。

• 示例 1：遍历列表

假设我们有一个数字列表，我们想打印出列表中的每个数字：

```
1 | numbers = [1, 2, 3, 4, 5]
2 | for number in numbers:
3 |     print(number)
```

这个例子中，`for` 循环遍历列表 `numbers` 中的每个元素，并将每个元素的值赋给变量 `number`，然后打印这个值。

- 示例 2：遍历字符串

`for` 循环同样可以用来遍历字符串中的每个字符：

```
1 | for char in "hello":
2 |     print(char)
```

在这个例子中，每个字符依次被赋值给变量 `char`，然后打印出来。

- 示例 3：使用 `range()` 函数

当需要迭代一系列数字时，可以使用内置的 `range()` 函数。例如，打印从 0 到 4 的数字：

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

`range(5)` 生成一个从 0 到 4 的整数序列，`for` 循环遍历这些数字。请使用 `help(range)` 查看 `range` 函数的详细使用说明。

- 示例 4：遍历字典

对于字典，你可以遍历键、值或键值对：

```
1 | info = {'name': 'Alice', 'age': 25}
2 | # 遍历键
3 | for key in info:
4 |     print(key)
5 |
6 | # 遍历值
7 | for value in info.values():
8 |     print(value)
9 |
10 | # 遍历键值对
11 | for key, value in info.items():
12 |     print(key, value)
```

在这个例子中，我们展示了三种遍历字典的方式：只遍历键、只遍历值以及同时遍历键和值。

• 示例 5：计算从 1 到 100 的所有整数之和

```
1 # 初始化累加器
2 total_sum = 0
3
4 # 使用 for 循环遍历从 1 到 100 的整数
5 for number in range(1, 101):
6     total_sum += number # 将当前数字加到累加器上
7
8 # 输出结果
9 print("1 到 100 之和是:", total_sum)
```

代码解释：

1. 初始化累加器：

- `total_sum` 初始化为 0，它用来存储从 1 到 100 的所有数字的累加和。

2. 循环遍历：

- `for number in range(1, 101):` 这里使用了 `range()` 函数来生成一个从 1 到 100 的整数序列。注意 `range(1, 101)` 包括 1 但不包括 101，正好符合从 1 到 100 的需求。
- 在每次循环中，`number` 变量将被赋予序列中的下一个数值。

3. 累加操作：

- `total_sum += number`：这行代码将当前循环到的数字 `number` 添加到 `total_sum` 变量上。这样，每次循环时 `total_sum` 都会更新，最终累加到 100。

4. 输出结果：

- 使用 `print()` 函数来输出从 1 到 100 的所有数字的总和。

4.2.4 判断与循环语句的结合

判断语句（如 `if`、`elif` 和 `else`）常常与循环结构（如 `for` 和 `while` 循环）结合使用，以实现更复杂的逻辑控制。这种结合可以帮助你根据特定条件执行或跳过循环中的代码块，或者在满足某些条件时终止循环，或者在一定条件下进行某种循环运算。

A 基本语法

判断语句可以放在循环体内部，根据条件来控制循环的行为。基本语法如下：

```
1 for 变量 in 序列:
2     if 条件:
3         执行代码块1
4     else:
5         执行代码块2
```

或者在 `while` 循环中：

```
1 while 条件1:
2     if 条件2:
3         执行代码块1
4     else:
5         执行代码块2
```

B 示例

• 示例 1：过滤数据

使用 `for` 循环结合 `if` 语句过滤出符合条件的数据：

```
1 numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2 even_numbers = []
3
4 for number in numbers:
5     if number % 2 == 0:
6         even_numbers.append(number)
7
8 print(" 偶数列表:", even_numbers)
```

这个例子中，循环遍历列表 `numbers`，利用 `if` 语句检查每个数字是否为偶数。如果是偶数，则将其添加到新列表 `even_numbers` 中。

• 示例 2：加密字符串

接受用户一个字符串作为输入，并将每个英文字符加密成其在字母表中的下一个字符，非英文字符保持不变。字母 ‘z’ 或 ‘Z’，加密为 ‘a’ 或 ‘A’。

```
1 # 从用户那里获取输入
2 input_string = input(" 请输入要加密的字符串: ")
3
4 encrypted_string = ""
```

```
5 for char in input_string:
6     if char.isalpha(): # 确保字符是字母
7         if char == 'z':
8             encrypted_string += 'a'
9         elif char == 'Z':
10            encrypted_string += 'A'
11        else:
12            # 将字符转换为下一个字符
13            encrypted_string += chr(ord(char) + 1)
14    else:
15        # 如果字符不是字母，直接添加到结果中
16        encrypted_string += char
17
18 # 输出加密后的字符串
19 print(" 加密后的文本:", encrypted_string)
```

代码解释

1. 获取用户输入:

```
1 input_string = input(" 请输入要加密的字符串: ")
```

这行代码使用 `input()` 函数提示用户输入一个字符串。用户输入的内容将被存储在变量 `input_string` 中。

2. 初始化加密字符串:

```
1 encrypted_string = ""
```

这行代码初始化一个空字符串 `encrypted_string`，用来存放加密后的结果。

3. 遍历输入的字符串:

```
1 for char in input_string:
```

使用 `for` 循环遍历 `input_string` 中的每个字符。`char` 变量在每次循环中代表字符串中的当前字符。

4. 判断字符是否为字母:

```
1 if char.isalpha():
```

使用 `isalpha()` 方法检查 `char` 是否为字母。如果是字母，执行下面的加密逻辑；如果不是字母，直接添加到结果字符串中。

5. 处理字母‘z’和‘Z’:

```
1 | if char == 'z':  
2 |     encrypted_string += 'a'  
3 | elif char == 'Z':  
4 |     encrypted_string += 'A'
```

如果当前字符是‘z’，则在结果中添加‘a’；如果是‘Z’，则添加‘A’。这是因为在英文字母表中，‘z’和‘Z’后面没有其他字母，所以需要循环回到‘a’或‘A’。

6. 处理其他字母:

```
1 | else:  
2 |     encrypted_string += chr(ord(char) + 1)
```

对于除‘z’和‘Z’外的其他字母，使用 `ord()` 函数获取其 ASCII 编码值，增加 1 后用 `chr()` 函数转换回字符，实现将字符向前移动一个位置的效果。

7. 处理非字母字符:

```
1 | else:  
2 |     encrypted_string += char
```

如果当前字符不是字母，它将直接被添加到 `encrypted_string` 中，不进行任何变更。

8. 输出加密后的文本:

```
1 | print(" 加密后的文本:", encrypted_string)
```

最后，打印出加密后的字符串。

练习

1. 解密字符串：编写一个解密程序，将输入的加密消息（示例 2 中的加密方法）还原成原始文本。
2. 编写程序打印如图4.1所示的九九乘法表。
3. 编写程序，接受一短英文作为输入，并计算其中的单词和数字的数量。

假设向程序提供以下输入：

| | | | | | | | | | |
|-------|--------|--------|--------|--------|--------|--------|--------|--------|--|
| 1×1=1 | | | | | | | | | |
| 1×2=2 | 2×2=4 | | | | | | | | |
| 1×3=3 | 2×3=6 | 3×3=9 | | | | | | | |
| 1×4=4 | 2×4=8 | 3×4=12 | 4×4=16 | | | | | | |
| 1×5=5 | 2×5=10 | 3×5=15 | 4×5=20 | 5×5=25 | | | | | |
| 1×6=6 | 2×6=12 | 3×6=18 | 4×6=24 | 5×6=30 | 6×6=36 | | | | |
| 1×7=7 | 2×7=14 | 3×7=21 | 4×7=28 | 5×7=35 | 6×7=42 | 7×7=49 | | | |
| 1×8=8 | 2×8=16 | 3×8=24 | 4×8=32 | 5×8=40 | 6×8=48 | 7×8=56 | 8×8=64 | | |
| 1×9=9 | 2×9=18 | 3×9=27 | 4×9=36 | 5×9=45 | 6×9=54 | 7×9=63 | 8×9=72 | 9×9=81 | |

图 4.1: 九九乘法表

```
1 | hello world! 123
```

那么输出应该是：

单词数： 2
数字数： 1

4. 编写一个程序来构建以下图案：



5. 编写一个程序来验证用户输入的密码是否有效。密码验证规则如下：

- 至少包含 1 个小写字母（a-z）和 1 个大写字母（A-Z）。

- 至少包含 1 个数字 (0-9)。
- 至少包含 1 个特殊字符 (\$#(中的任意一个?))。
- 密码长度至少为 6 个字符。
- 密码长度不得超过 16 个字符。

4.3 列表、字典推导与生成器表达式

列表推导和字典推导可以看做是复合数据类型（列表和字典）与判断循环语句的结合。这种结合提供了一种非常高效和简洁的方式来处理数据，并生成新的数据对象。

4.3.1 列表推导 (List Comprehension)

列表推导是一种在 Python 中快速生成列表的方法，通过一个简洁的表达式可以从其他可迭代对象（如列表、元组、字符串等）中创建出新的列表。使用列表推导，你可以对原始数据进行过滤和转换处理。

A 作用

- **简洁性**：使用一行代码替代多行循环和判断语句，使代码更简洁易读。
- **效率**：列表推导在很多情况下比传统的循环方法更加高效。
- **方便性**：可以直接在列表推导中进行复杂的操作（如条件筛选、函数应用等）。

B 示例

假设我们需要从一个数字列表中找出所有偶数，并对它们乘以 2，传统的方法可能需要多行代码，而使用列表推导，只需一行：

```
1 numbers = [2, 3, 5, 6, 8, 9, 11, 14, 15, 16]
2 result = [x * 2 for x in numbers if x % 2 == 0] # 找出 numbers 中的偶数，并乘以
3 print(result)
```

4.3.2 字典推导 (Dict Comprehension)

字典推导类似于列表推导，但它是用来快速生成字典的。在字典推导中，你可以直接从一个列表或者任何其他序列生成一个字典，同时进行键和值的自定义计算。

A 作用

- **快速转换**：可以将一组数据快速转化为字典形式，非常适合处理键值对。
- **数据关联**：方便地对数据进行键值关联，比如从两个相关联的列表创建字典。
- **代码简化**：减少代码量，增加代码的可读性和清晰度。

B 示例

如果我们有两个列表，一个是学生的名字，另一个是他们的分数，我们可以快速将这两个列表转换成一个字典，其中学生的名字为键，分数为值：

```
1 names = ["Alice", "Bob", "Charlie"]
2 scores = [85, 90, 88]
3 score_dict = {name: score for name, score in zip(names, scores)}
```

4.3.3 生成器表达式

生成器（Generator）是一种在 Python 中用来创建迭代器的工具。它使用了一种特殊的函数或表达式，这种函数或表达式可以在保持最后执行状态的情况下，一次产生一个结果项，依次迭代执行，直到满足结束条件。生成器因此非常适合处理大量数据，或者在不确定所有数据大小时进行迭代，因为它们在任何时间点都只需要保留当前的状态和数据，而不是像列表那样一次性加载所有数据到内存。**生成器表达式**语法上类似于列表推导，但使用圆括号而不是方括号。使用函数进行构造生成器将在后续章节讨论。

想象一下，你在麦当劳点了一杯饮料，需要使用吸管来喝。这里的吸管盒子就像是一个生成器。每次你需要吸管时，你就按一下吸管盒的按键，吸管盒弹出一根吸管，这个动作类似于调用生成器的 `next()` 函数来获取下一个值。

A 生成器的特性

1. **按需使用**：你不会一次拿出所有的吸管，而是每次需要时才拿一根。这就像生成器在每次迭代时只生成一个值，而不是一次生成所有的数据并存储在内存中。
2. **节省资源**：如果每个人都一次性拿出他们某个时间区间所需的所有吸管，那将需要更多的空间来放置这些吸管，而且可能会有很多浪费。生成器通过只在需要时生成值，帮助节省内存资源。

- 3. **无需等待初始化**：使用生成器时，你不需要等待所有数据都处理完毕才开始处理。你可以从生成第一个值开始立即进行后续处理。
- 4. **状态保持**：吸管盒子保持其状态记住还剩多少吸管，这样你每次拿一个都知道还剩下多少。同样，生成器也保持其状态，知道下一次调用 `next()` 时从哪里开始继续生成下一个值。
- 5. **外部交互**：如果麦当劳决定改变吸管的类型或颜色，这将影响你下次拿吸管时得到的吸管类型。类似地，生成器可以通过 `send()` 方法接收外部数据，这可能会影响生成器接下来生成的值。

B 生成器的常用操作

| 操作/函数 | 描述 |
|--------------------------------|--|
| <code>next(generator)</code> | 从生成器获取下一个元素。如果没有更多元素，抛出 <code>StopIteration</code> 异常。 |
| <code>generator.close()</code> | 关闭生成器，阻止生成器产生更多的值。之后对生成器使用 <code>next()</code> 或 <code>send()</code> 将会抛出 <code>StopIteration</code> 异常。 |
| <code>iter(object)</code> | 从符合迭代器协议的对象中获取一个迭代器。对于生成器，它返回生成器本身。 |

C 示例

```
1 squares_gen = (x**2 for x in range(10))
2 print(type(squares_gen)) # 输出 generator
3 print(next(squares_gen)) # 输出 0
4 print(next(squares_gen)) # 输出 1
```

生成器提供了一种高效处理大量或无限数据集的方法，而列表推导则更适用于较小的数据集或在内存限制不严格的场合。两者在 Python 编程中都非常有用，选择哪一个取决于具体的应用场景和性能考虑。

练习

- 1. 编写一个列表推导，生成一个列表，其中包含 1 到 20 中偶数的开方。
- 2. 给定一个单词列表，使用列表推导创建一个新列表以包含所有长度大于 5 的单词。

示例输入:

```
1 | words = ["apple", "banana", "cherry", "kiwi", "mango", "blueberry"]
```

预期输出:

```
1 | ["banana", "cherry", "blueberry"]
```

3. 给定相同的单词列表, 使用字典推导创建一个字典, 其中键是单词, 值是相应单词的长度。

预期输出:

```
1 | {"apple": 5, "banana": 6, "cherry": 6, "kiwi" : 4,  
2 |   "mango": 5, "blueberry": 9}
```

4. 使用字典推导, 根据给定的一组单词, 创建一个字典, 其中键是单词, 值是该单词在列表中的索引。

示例输入:

```
1 | words = ["apple", "banana", "cherry", "kiwi"]
```

预期输出:

```
1 | {"apple": 0, "banana": 1, "cherry": 2, "kiwi": 3}
```

5. 假设你有一份学生的成绩列表, 该列表以字符串列表的形式存储, 每个字符串包含学生的姓名和分数, 格式如下:

```
scores = ["Alice:92", "Bob:85", "Cara:78", "David:88", "Eva:100"]
```

编写 Python 代码, 使用生成器表达式来实现以下功能:

(1) **筛选及转换数据:** 从列表中筛选出分数超过 85 分的学生, 并将这些数据转换成元组, 每个元组包含学生的姓名和分数。

(2) **计算平均分:** 使用生成器表达式计算筛选后学生的平均分。

5

程序结构

从理论上讲，使用基本的赋值、判断和循环语句就足以构建任何类型的计算机程序，但单用这些基本语句在处理复杂应用时效率较低，且难以维护和拓展。为了提升编程效率并优化代码的组织结构，本章将深入探讨几种关键的程序结构组织技术，包括函数、类、模块和包。通过这些技术，程序员能够编写出更清晰、更灵活、更易于维护的代码。

Python 的代码结构可以分为多个层次，如下图所示。这些组件共同构成了 **Python** 程序的基础架构，每个层次都承担着特定的功能和角色。通过有效地利用这些组件，可以大幅提高代码的可重用性和模块化程度，进而简化复杂软件的开发和维护工作。

Python 程序的结构可以类比为一部精心编排的文学作品，包含了卷、章、节等多个层级，每一层都承载着故事的发展和人物的变化，正如 **Python** 程序中各个结构元素的作用和关联。包相当于作品中的卷，每个卷都是独立的，但又与全书的其它部分相互关联。在 **Python** 中，包（**package**）包含各种特定功能的模块。模块（**Modules**）类似于每卷中的不同章，一个模块就是一个包含函数、类、变量等的文件，各自独立但可以被其它模块引用。类（**classes**）相当于节，是定义对象的蓝图，封装了数据（属性）和行为（方法），用于创建相互关联但独立运作的对象实例。函数（**functions**）可以比作段落，每个函数封装了一段特定的程序代码，用来执行一个具体的任务。语句（**statements**）可以看着句子，是代码的基本构建块，用于执行操作，如赋值、条件判断、循环等。而变量（**variables**）常量（**constants**）等类似于词汇，是语句的重要构成元素。

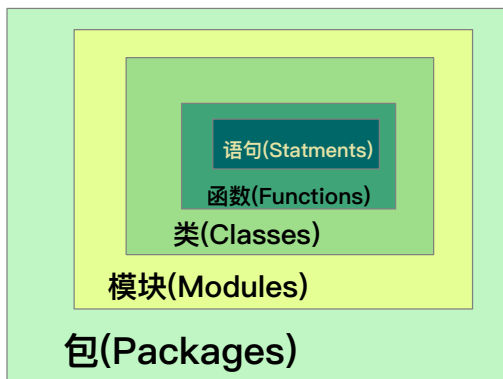


图 5.1: Python 程序结构

- **语句**: 最基本的代码单元。
- **函数**: 函数是组织好的、可重用的代码块，用于执行特定任务。
- **类**: 封装数据和功能的方法。
- **模块和包**: 组织大型代码库的方式。

5.1 函数

在计算机程序设计中，函数是一种代码组织或封装的方法，旨在方便代码重复使用、模块化和提高代码的可读性。函数通过其名称进行标识，并可以接受输入参数，执行预定义的代码块后返回一个结果。因此，函数可以被视为将输入转换为输出的转换器。这与数学中的函数在概念上一致。数学中的函数将输入值（自变量）映射（转化）到唯一的输出值（因变量）。尽管 **Python** 函数和数学函数都涉及“输入”和“输出”的概念，但它们在应用和复杂性上有明显的不同：

1. 目的与用途：

- **数学函数** 主要用于表达数值之间的关系，是抽象和理论的。
- **Python 函数** 更具实用性，可以执行复杂的操作，如文件操作、数据处理等，不仅限于数值计算。

2. 灵活性：

- **数学函数** 通常有固定的定义，每个输入对应一个输出。

- **Python 函数** 可以具有更灵活的结构，可以根据条件语句返回不同的结果，甚至修改外部变量的状态或进行各种输入输出操作。

3. 参数与返回值:

- **数学函数** 的参数和返回值几乎总是数值或数值的集合。
- **Python 函数** 的参数和返回值可以是任何类型，包括数字、字符串、列表、字典、自定义的数据对象等。

4. 确定性:

- **数学函数** 是确定性的，对于同一个输入总是产生相同的输出。
- **Python 函数** 可以包含随机性或依赖于外部状态，因此可能不总是对同一输入产生相同的输出。

5.1.1 函数定义

在 Python 中，函数是通过使用 `def` 关键字来定义的，后跟函数名和圆括号，圆括号中可以包含一些参数。函数体开始于下一行，并且必须缩进。函数可以包含返回语句 `return`，用来返回值给调用者。如果没有 `return` 语句，函数默认返回 `None`。

A 基本语法

函数的定义

函数定义的基本语法如下:

```
1 def function_name(parameters):  
2     """docstring"""  
3     # Function body  
4     return value
```

- **function_name:** 函数名，用于标识函数。命名规则与变量相同，通常使用小写字母，多个单词可以用下划线分隔。
- **parameters:** 参数列表，用于从函数外部接收值。参数是可选的；如果函数不需要输入，则这里为空。
- **docstring:** 文档字符串（可选），用于描述函数的功能，函数参数的意义，输出结果。为函数提供文档说明是一个好习惯，可以增加代码的可读性。

- **Function body:** 函数体，执行转换操作的语句块。
- **return:** 返回语句，用于将结果传回给调用者。这也是可选的；如果省略，函数将返回 `None`。
- 示例：

自定义绝对值函数

```

1  def my_abs(x):
2      """
3      Return the absolute value of a number.
4      x: a float or integer number.
5      """
6      if x < 0:
7          return -x
8      else:
9          return x

```

函数的调用

函数调用发生在你想要使用一个函数来执行特定任务时。在调用函数时，你必须提供一些必要的信息——这通常包括函数名称和一组参数（如果函数需要的话）。函数在接收到这些信息后，会执行预定义的操作，然后可能会返回一个结果。比如想求-5的绝对值：

```

1  abs_val = my_abs(-5)  # 调用函数 my_abs 计算-5 的绝对值
2  print(abs_val)       # 打印函数的返回结果

```

B 函数定义与调用的理解

想象一下你有一个食谱，这个食谱上写着如何制作一种特定的菜肴——比如说番茄炒鸡蛋。食谱上会详细列出需要哪些原料（相当于函数的参数），可能包括”鸡蛋“和”番茄“，以及按照什么步骤去烹饪这些原料（相当于函数体）。完成后，你会得到一盘美味的番茄炒鸡蛋（相当于函数的返回值）。番茄炒鸡蛋食谱如同函数，其定义代码可能如下：

```

1  def make_tomato_egg(eggs, tomatoes):
2      """Prepare tomato and egg stir-fry"""
3      chopped_tomatoes = chop(tomatoes)  # 切番茄
4      beaten_eggs = beat(eggs)  # 打鸡蛋
5      cooked_dish = cook(beaten_eggs, chopped_tomatoes)
6      # 将打好的鸡蛋和切好的番茄一起炒制
7      return cooked_dish

```

当你决定要做这道菜时，你就根据定义好的函数来“调用”它，提供必要的原料（参数的具体值），比如提供了两个鸡蛋和两个番茄。

函数调用的代码可能看起来是这样的：

```
1 dish = make_tomato_egg(2, 2)
2 print(dish)  # 输出制作完成的番茄炒鸡蛋
```

以上番茄炒鸡蛋的函数代码只是用于描述和说明函数的定义与调用，无法直接运行。

C 参数的定义

在 Python 中定义函数时，可以使用多种类型的参数。这些参数类型包括位置参数、关键字参数、默认参数、可变参数（*args 和 **kwargs）。下面将详细说明每种参数的定义和使用方法。

• 位置参数

位置参数是函数定义中最基本的参数类型。调用函数时，必须按顺序提供这些参数。位置参数可以被视为采用元组的方式存储的。

```
1 def minus(x, y):
2     return x - y
3
4 result = minus(5, 3)  # 位置参数，按顺序传递
5 print(result)  # 输出 2
6 result = minus(3, 5)
7 print(result)  # 输出 -2
```

• 关键字参数

关键字参数允许在调用函数时，通过参数名指定参数值，从而无需关注参数的顺序。关键字参数可以被视为采用字典的方式存储的。

```
1 def greet(name, message):
2     print(f"{message}, {name}!")
3
4 greet(name="Alice", message="Hello")  # 使用关键字参数
5 greet(message="Hi", name="Bob")      # 调用的顺序可以不同
```

• 默认参数

默认参数在函数定义时为参数指定默认值。如果调用函数时未提供该参数，则使用默认值。


```

1 def greet(name, message="Hello"):
2     print(f"{message}, {name}!")
3
4 greet("Alice")           # 使用默认参数值, 输出 "Hello, Alice!"
5 greet("Bob", "Hi")       # 覆盖默认参数值, 输出 "Hi, Bob!"

```

• 可变参数

__*args__

*args 用于接收任意数量的位置参数，返回一个元组。

```

def add(*args):
    return sum(args)

result = add(1, 2, 3, 4) # 传递多个位置参数
print(result) # 输出 10

```

__**kwargs__

**kwargs 用于接收任意数量的关键字参数，返回一个字典。

```

1 def print_info(**kwargs):
2     for key, value in kwargs.items():
3         print(f"{key}: {value}")
4
5 print_info(name="Alice", age=30, city="New York")
6 # 输出:
7 # name: Alice
8 # age: 30
9 # city: New York

```

• 参数组合

在定义函数时，可以组合使用上述不同类型的参数。参数的顺序应该是：位置参数、默认参数、*args、关键字参数和 **kwargs。

```

1 def demo(pos1, pos2, default1=10, default2=20, *args, kw1,
2         kw2=100, **kwargs):
3     print(f"pos1: {pos1}, pos2: {pos2}")
4     print(f"default1: {default1}, default2: {default2}")
5     print(f"args: {args}")
6     print(f"kw1: {kw1}, kw2: {kw2}")
7     print(f"kwargs: {kwargs}")
8

```

```

9 | demo(1, 2, 3, 4, 5, 6, 7, kw1="Keyword1", extra="Extra")
10 | # 输出:
11 | # pos1: 1, pos2: 2
12 | # default1: 3, default2: 4
13 | # args: (5, 6, 7)
14 | # kw1: Keyword1, kw2: 100
15 | # kwargs: {'extra': 'Extra'}

```

通过上述方式，你可以灵活地定义和调用函数，满足不同的需求。

• 参数类别限定符

在 Python3.8+ 版本的函数定义中，/ 和 * 分别用于指定位置参数和关键字参数。它们帮助我们更明确地控制参数的传递方式。下面详细说明它们的使用。

/ 用于位置参数

/ 用于将前面的参数指定为仅限位置参数。这意味着这些参数必须通过位置传递，不能使用关键字来传递。

```

1 | def func(a, b, /, c, d):
2 |     print(a, b, c, d)
3 |
4 | # 正确
5 | func(1, 2, 3, 4)
6 |
7 | # 错误
8 | func(a=1, b=2, c=3, d=4)
9 | # TypeError: func() got some positional-only arguments passed as keyword arguments

```

在这个例子中，a 和 b 是仅限位置参数，而 c 和 d 可以通过位置或关键字传递。

* 用于关键字参数

* 用于将后面的参数指定为仅限关键字参数。这意味着这些参数必须通过关键字来传递，不能通过位置传递。

```

1 | def func(a, b, *, c, d):
2 |     print(a, b, c, d)
3 |
4 | # 正确
5 | func(1, 2, c=3, d=4)
6 |
7 | # 错误
8 | func(1, 2, 3, 4)
9 | # # TypeError: func() takes 2 positional arguments but 4 were given

```

在这个例子中，`c` 和 `d` 是仅限关键字参数，而 `a` 和 `b` 可以通过位置传递。

综合使用 / 和 *

我们可以在一个函数定义中同时使用 / 和 *，从而精确地控制哪些参数是仅限位置的，哪些是可以通过任意方式传递的，以及哪些是仅限关键字的。

```

1 def func(a, b, /, c, d, *, e, f):
2     print(a, b, c, d, e, f)
3
4 # 正确
5 func(1, 2, 3, 4, e=5, f=6)
6
7 # 错误
8 func(1, 2, 3, 4, 5, 6)
9 # TypeError: func() takes 4 positional arguments but 6 were given
10 func(a=1, b=2, c=3, d=4, e=5, f=6)
11 # TypeError: func() got some positional-only arguments
12 # passed as keyword arguments: 'a, b'

```

在这个例子中：

- `a` 和 `b` 是仅限位置参数，必须通过位置传递。
- `c` 和 `d` 可以通过位置或关键字传递。
- `e` 和 `f` 是仅限关键字参数，必须通过关键字传递。

以下是一个更加复杂的示例，展示如何在实际函数定义中使用这些符号：

```

1 def complex_func(x, y, /, z, *, a, b=10):
2     print(f"x: {x}, y: {y}, z: {z}, a: {a}, b: {b}")
3
4 # 调用方式
5 complex_func(1, 2, 3, a=4) # 正确
6 complex_func(1, 2, z=3, a=4, b=5) # 正确
7
8 # 错误调用示例
9 complex_func(1, 2, 3, 4)
10 # TypeError: complex_func() takes 3 positional arguments
11 # but 4 were given
12 complex_func(x=1, y=2, z=3, a=4)
13 # TypeError: complex_func() got some positional-only arguments
14 # passed as keyword arguments: 'x, y'

```

通过这些例子，/ 和 * 的用法会变得更加清晰，帮助你在定义函数时更好地控制参数传递方式。

D 函数返回值

函数可以通过 `return` 语句或 `yield` 返回值。如果函数没有返回语句，或者 `return` 语句没有指定值，函数会返回 `None`。

- 使用 `return` 语句

单一返回值

最基本的 `return` 语句返回一个值。

```
1 def multiply(a, b):  
2     return a * b  
3  
4 result = multiply(3, 5)  
5 print(result) # 输出 15
```

多个返回值

Python 函数可以一次返回多个值，多个返回值将作为一个元组返回。也可以把所有需要返回的元素放入列表或字典后返回。

```
1 def get_coordinates():  
2     x = 10  
3     y = 20  
4     return x, y  
5  
6 coords = get_coordinates()  
7 print(coords) # 输出 (10, 20)  
8  
9 x, y = get_coordinates()  
10 print(x, y) # 输出 10 20
```

使用 `yield` 返回生成器

`return` 语句会立即终止函数的执行，并将一个值（或多个值作为元组）返回给调用者。但 `yield` 可以多次返回值，而不终止函数的执行。每次 `yield` 返回时，函数的状态会被“冻结”，以便下次继续执行。采用 `yield` 返回结果的函数称为生成器函数。

```
1 def generate_numbers():  
2     for i in range(5):  
3         yield i  
4  
5 gen = generate_numbers()
```

```
6 | for number in gen:
7 |     print(number)
8 | # 输出:
9 | # 0
10 | # 1
11 | # 2
12 | # 3
13 | # 4
```

5.1.2 lambda 函数

Lambda 函数，也称为匿名函数，允许用户定义短小的、无名的函数对象。这些函数通常在需要一个函数，但又不想正式定义一个函数时使用，以简化代码或避免代码冗余。

Lambda 函数的基本语法如下：

```
1 | lambda arguments: expression
```

- `lambda` 是一个关键字，指示后跟的是一个匿名函数。
- `arguments` 是传递给匿名函数的参数，可以是多个，用逗号分隔。
- `expression` 是关于这些参数的表达式，该表达式的计算结果会被该函数自动返回。

A 特点

- Lambda 函数体比普通 Python 函数体更简单。
- 它们通常用于编程中需要传递简单函数作为参数的场景。
- Lambda 函数可以访问其包含作用域内的变量。
- Lambda 只能有一个表达式，不允许多个表达式或语句。

B 示例

```
1 | a = 1
2 | square_adda = lambda x: x**2 + a
3 | print(square_adda(2))

1 | adder = lambda x, y: x + y
2 | print(adder(5, 3)) # 输出: 8
```

5.1.3 变量作用范围与命名空间

变量的作用范围（Scope）决定了变量的可见性和生命周期。理解变量的作用范围对于编写清晰的代码和避免错误至关重要。Python 中主要有以下几种作用范围：

1. 局部作用域（Local）
2. 嵌套作用域（Enclosing）
3. 全局作用域（Global）
4. 内置作用域（Built-in）

A 局部作用域

局部作用域指的是在函数内部定义的变量。这些变量只能在函数内部访问，函数执行完毕后，这些变量会被销毁。

```
1 def my_function():
2     local_var = 10
3     print(local_var)
4
5 my_function()
6 print(local_var)  # 这行会报错，因为 local_var 是局部变量，函数外部不可见
```

B 嵌套作用域

嵌套作用域指的是嵌套函数中的作用域，通常与 `nonlocal` 关键字一起使用。`nonlocal` 用于声明一个变量来自上一级（但不是全局）作用域。

```
1 def outer_function():
2     outer_var = "I'm outer"
3     print(outer_var)
4     def inner_function():
5         nonlocal outer_var
6         outer_var = "I'm modified by inner"
7         print(outer_var)
8
9     inner_function()
10    print(outer_var)
11
12 outer_function()
13 # 输出:
14 # I'm outer
15 # I'm modified by inner
16 # I'm modified by inner
```

C 全局作用域

全局作用域指的是在模块级别定义的变量。这些变量可以在模块的任何地方访问。使用 `global` 关键字可以在函数内部声明一个全局变量。在 Python 中, `__name__` 是一个特殊的变量, 用来表示当前模块的名字。当一个 Python 文件被直接运行时, `__name__` 的值是 `'__main__'`。这意味着该文件是主模块, 通常用于脚本执行的入口点。在 Jupyter Notebook 中, 每个单元格的执行环境相当于一个独立的脚本。当你运行一个单元格时, Jupyter Notebook 会将 `__name__` 变量设置为 `'__main__'`, 这意味着该单元格就是当前的主模块。

```
1 | global_var = "I'm global"
2 |
3 | def my_function():
4 |     global global_var
5 |     print(global_var)
6 |     global_var = "Modified by function"
7 |     print(global_var)
8 |
9 | my_function()
10 | print(global_var)
11 | # 输出:
12 | # I'm global
13 | # Modified by function
14 | # Modified by function
```

D 内置作用域

内置作用域包括 Python 内置的函数和异常等, 它们在任何地方都可以被访问。

```
1 | print(len([1, 2, 3])) # len 是一个内置函数
```

E 命名空间 (Namespace)

命名空间是变量名到对象的映射。你可以把命名空间理解为一个名字和对象的字典。命名空间确保了变量名的唯一性, 并避免命名冲突。

想象一下, 你有一个装满各种物品的房间。房间里的每个物品都有一个标签 (名字), 告诉你它是什么。这个房间就是一个命名空间。你可以有多个房间, 每个房间都有自己的物品和标签。这些房间可以是彼此独立的, 也可以是嵌套的:

- **局部命名空间**：就像一个盒子，只有在你打开盒子时才能看到里面的物品。函数内部的变量就是这样，它们只在函数执行时可见。
- **嵌套命名空间**：就像一个盒子里再放一个盒子。内部盒子可以访问外部盒子的物品，但反之不行。这就是嵌套函数的变量作用范围。
- **全局命名空间**：就像这个房间中的共享的物品，如 **wifi** 信号，房间中所有位置都可以使用它们。
- **内置命名空间**：就像房间的固定设施，比如水电，大楼的所有房间都可以使用。

练习

1. 定义一个函数 `power_sum`，该函数计算并返回以下算式的值：

$$1^m + 2^m + \dots + n^m$$

其中，`n` 的默认值为 `100`，`m` 的默认值为 `1`。

2. 定义两个函数 `encrypt` 和 `decrypt`，分别用于对字符串进行加密和解密。
 - 加密过程：将字符串中的每个字母替换为其后一个字符。例如，字母 `a` 变为 `b`，`b` 变为 `c`，以此类推。对于字母 `z`，则替换为 `a`。非字母字符保持不变。
 - 解密过程：将加密后的字符串还原成原始字符串。
3. 利率可以以不同的方式复利。一种常见的方式是年复利 `m` 次，另一种方式是连续复利。给定年复利 `m` 次的利率 `r`，编写一个函数 `convert_to_continuous(r, m)`，将其转换为等价的连续复利利率。
4. 编写一个函数 `sort_by_last_char(lst)`，接受一个字符串列表 `lst`，并返回一个按每个字符串最后一个字符排序的新列表。
5. 编写一个函数 `fibonacci(n)`，接受一个整数 `n`，返回斐波那契数列的第 `n` 个数。斐波那契数列的定义为： $F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2)$ 对于 $n \geq 2$ 。
6. 编写一个函数来判断给定的整数是否是质数。

5.2 类

在面向对象编程（**OOP**）中，**类**是一种用户定义的数据类型，它是用于创建实例的模板或蓝图。类定义了一组属性（变量）和方法（函数），使得我们可以创建多个具有相同属性和行为的实例。可以把类比作一张详细的车辆设计图纸。这张图纸包含了车辆设计的所有重要信息和细节，比如：

- **属性**：车的颜色、型号、发动机类型、车轮数量等。
- **方法**：车如何启动、加速、转弯、刹车等。

这张图纸并不是一辆实际的车，而是一个抽象的概念，描述了车应该是什么样子的，以及它能做什么。

实例是根据类创建出来的具体对象。在 **C++** 和 **Java** 等支持面向对象的程序设计语言中，通常把类生成的具体实例称为对象，但在 **Python** 中，**object** 本身是一种特殊的数据类型，为避免混淆，采用实例这个名称。每个实例都是类的一个具体实现，拥有类定义的方法和属性。如果类是车辆的设计图纸，那么实例就是根据这张图纸制造出来的实际车辆。每辆具体的车都是一个实例，它们有自己的具体属性值，比如：

- 一辆蓝色的比亚迪秦 L。

虽然每辆比亚迪秦 L（实例）可能在颜色、发动机、内饰配置等方面有所不同，但它们都遵循同一张图纸（类）的设计规范。具体来说：

- **属性**：每辆车（实例）都有自己的颜色、发动机编号、内饰配置等。
- **方法**：每辆车都可以启动、加速、转弯、刹车，尽管它们在具体表现上可能有所差异。

类在程序设计中的作用，可以概况如下几点：

1. **组织和管理代码**：类帮助我们相关的属性和方法组织在一起，使代码更清晰，更易管理。
2. **提高代码重用性**：通过定义类，可以创建多个实例，而不需要重复写相同的代码。比如，定义一个“车”类后，可以很容易地创建不同的车。

3. **简化代码维护**: 如果需要修改某类事物的共同特性或行为, 只需修改类的定义, 而不需要逐个修改实例。
4. **实现抽象和封装**: 类可以隐藏复杂的实现细节, 只对外暴露简单的接口, 使得代码更易于理解和使用。
5. **支持继承和多态**: 类可以继承其他类的属性和方法, 从而实现代码复用和扩展。同时, 不同类的实例可以以相同的方式使用 (多态), 这使得编写灵活而强大的代码成为可能。

5.2 .1 类的定义

类的定义通常包括以下几个部分:

- **类名**: 遵循大驼峰命名规则, 即所有单词的首字母大写, 不使用下划线。
- **属性**: 类中定义的变量, 用于存储数据。
- **方法**: 类中定义的函数, 用于描述对象的行为或操作对象的数据。

基本的类定义语法如下:

```
1 class ClassName:
2     def __init__(self, param1, param2):
3         self.attribute1 = param1
4         self.attribute2 = param2
5
6     def method1(self):
7         # 方法 1 的代码
8
9     def method2(self, param):
10        # 方法 2 的代码
```

- **class 关键字**: 用于开始一个类的定义。
- **ClassName**: 类名, 按照惯例首字母大写, 如果有多个单词则采用大驼峰命名法 (CamelCase)。
- **方法定义**: 使用 **def** 关键字定义函数。类中的函数通常称为方法。所有在类中定义的方法和类的属性均需要缩进, 表示其从属于所定义的类。
- **__init__ 方法**: 一个特殊的方法, 称为类的构造器。当一个类的实例被创建时, 这个方法会被自动调用。用于初始化实例的属性或执行一些启动任务。

- **self 参数**: 在类的方法定义中第一个参数通常是 **self**, 它代表类实例自身。在调用方法时你不需要传递这个参数, Python 会自动处理。
- **属性**: 使用 **self.attribute** 来定义, 意味着这些属性与特定的类实例相关联。

A 示例:

A.1 定义 Student 类

```
1 class Student:
2     def __init__(self, student_id, name, gender):
3         self.student_id = student_id # 学生的学号
4         self.name = name              # 学生的姓名
5         self.gender = gender          # 学生的性别
6         self.scores = []              # 学生的成绩单, 初始为空列表
7
8     def add_score(self, course, score):
9         """
10        添加成绩记录到成绩单。
11        :param course: 课程名称
12        :param score: 该课程的成绩
13        """
14        self.scores.append((course, score))
15
16    def print_scores(self):
17        """
18        打印学生的成绩单。
19        """
20        print(f" 成绩单 - {self.name}:")
21        print('-'*16)
22        for course, score in self.scores:
23            print(f"{course}: {score}")
24
25        print('-'*16)
26        print(f" 平均成绩: {self.calculate_average():.2f}")
27
28    def calculate_average(self):
29        """
30        计算并返回学生的平均成绩。
31        :return: 平均成绩
32        """
33        if not self.scores:
34            return 0
35        total = sum(score for _, score in self.scores)
36        return total / len(self.scores)
```

A.2 使用 Student 类

```
1  ## 创建学生实例
2  student1 = Student(student_id=1, name=" 张三", gender=" 男")
3
4  ## 添加成绩
5  student1.add_score('数学', 92)
6  student1.add_score('英语', 88)
7  student1.add_score('物理', 95)
8
9  ## 计算并打印平均成绩
10 average_grade = student1.calculate_average()
11 print(f"{student1.name}的平均成绩是: {average_grade:.2f}")
12
13 ## 打印成绩单
14 student1.print_scores()
```

上述代码运行的输出结果如下:

```
1 张三的平均成绩是: 91.67
2 成绩单 - 张三:
3  -----
4  数学: 92
5  英语: 88
6  物理: 95
7  -----
8  平均成绩: 91.67
```

B 运算符重载

Python 的内置函数 `print` 能打印 python 的各种数据对象, 但是如果我们调用 `print` 打印自定义数据类型 `Student` 创建的对象 `student1`, 结果是什么?

```
1 print(student1)
```

当运行上述代码时, 输出将会如下:

```
1 <__main__.Student object at 0x10c1936a0>
```

而这一打印结果只是简单显示 `student1` 是一个在主模块中定义的 `Student` 类的对象。如果我们希望调用 `print` 打印 `student1` 就会输出该同学的基本信息, 我们就需要重新定义 `Student` 类的特殊函数 `__str__`。在 Python 中, 这种重新类的特殊函数, 通常也被称为运算符重载。

在 Python 中，运算符重载是通过定义特殊的方法来实现的，这些方法有固定的名称并且以双下划线 (__) 开头和结尾。__str__ 是这些特殊方法之一，用于定义对象的可打印的字符串表示，通常是面向用户的。

当你使用 print() 函数或者 str() 函数对一个对象进行操作时，Python 会自动寻找并调用该对象的 __str__ 方法。如果你在你的类中定义了 __str__ 方法，当实例传递给 print() 或 str() 时，Python 将使用你定义的方法来生成字符串表示。

让我们在前面定义的 Student 类中添加一个如下 __str__ 方法，以便打印学生对象时可以得到更有信息的输出。

```
1 def __str__(self):
2     return (f" 学号: {self.student_id}\n"
3           f" 姓名: {self.name}\n"
4           f" 性别: {self.gender}\n"
5           f" 平均成绩: {self.calculate_average():.2f}")
```

重新运行上述创建学生实例的代码，然后运行

```
1 print(student1)
```

输出结果如下：

```
1 学号: 1
2 姓名: 张三
3 性别: 男
4 平均成绩: 91.67
```

在 Python 中，自定义类型数据对象的许多运算符可以通过定义特定的方法来重载。下面是 Python 中一些常见的被重载的运算符及其对应的特殊方法：

| 运算符 | 特殊方法 | 描述 |
|--------|-------------|----------|
| + | __add__ | 加法 |
| - | __sub__ | 减法 |
| * | __mul__ | 乘法 |
| / | __truediv__ | 真除法 |
| == | __eq__ | 等于 |
| != | __ne__ | 不等于 |
| < | __lt__ | 小于 |
| <= | __le__ | 小于等于 |
| > | __gt__ | 大于 |
| >= | __ge__ | 大于等于 |
| str() | __str__ | 对象的字符串表示 |
| len() | __len__ | 对象长度 |
| bool() | __bool__ | 布尔值表示 |

C 类的继承

继承允许定义一个类（子类或派生类）继承另一个类（父类或基类）的属性和方法。子类自动获得父类的所有特性，并可以添加新的特性或修改继承的行为。

C.1 示例：

```
1 class Animal:
2     def __init__(self, name):
3         self.name = name
4
5     def speak(self):
6         raise NotImplementedError(" 子类必须重新实现这一方法")
7
8 class Dog(Animal):
9     def speak(self):
10        return f"{self.name}正在汪汪...!"
11
12 class Cat(Animal):
13     def speak(self):
14        return f"{self.name} 正在喵喵...!"
```

在上面的例子中，`Animal` 是一个基类，定义了一个构造函数和一个 `speak` 方法。`speak` 方法被设计为抽象的，意味着基类的 `speak` 方法不提供具体实现，而是要求任何继承自 `Animal` 的类提供自己的 `speak` 方法实现。`Dog` 和 `Cat` 类继承自 `Animal` 类，并实现了自己版本的 `speak` 方法。

D 多态

多态是指在不同的对象上调用相同的方法，可以产生不同的行为。在 `Python` 中，多态是隐式的，因为 `Python` 是动态类型语言，它不要求对象具有相同的类来实现相同的方法。

```
1 def animal_speak(animal):
2     print(animal.speak())
3
4 ## 创建 Animal 类的实例
5 dog = Dog(" 小黄")
6 cat = Cat(" 汤姆")
7
8 ## 调用相同的函数
9 animal_speak(dog)
10 animal_speak(cat)
```

上述代码的运行结果如下：

```
1 小黄正在汪汪...!  
2 汤姆正在喵喵...!
```

5.2.2 类的使用场景

A 需要表示和操作复杂数据结构时

当你需要处理具有复杂属性和行为的数据时，使用类可以帮助你更清晰地组织这些信息。例如，如果你在开发一个图书管理系统，定义一个 `Book` 类可以让你轻松管理每本书的标题、作者、ISBN、库存等信息。

```
1 class Book:  
2     def __init__(self, title, author, isbn, stock):  
3         self.title = title  
4         self.author = author  
5         self.isbn = isbn  
6         self.stock = stock  
7  
8     def borrow(self):  
9         if self.stock > 0:  
10             self.stock -= 1  
11             return True  
12         else:  
13             return False  
14  
15     def return_book(self):  
16         self.stock += 1
```

B 需要创建多个具有相同属性和方法的对象时

如果你的程序需要操作大量相似的对象，使用类可以避免重复代码。比如，在一个游戏中，你可能需要创建多个敌人，每个敌人都有相似的属性和行为。

```
1 class Enemy:  
2     def __init__(self, name, health, damage):  
3         self.name = name  
4         self.health = health  
5         self.damage = damage  
6  
7     def attack(self, target):  
8         target.health -= self.damage
```

C 需要实现抽象和封装时

类可以帮助你隐藏实现细节，仅暴露必要的接口，从而提高代码的可读性和可维护性。例如，银行系统中的账户类可以封装余额的细节，只提供存款和取款的方法。

```
1 class BankAccount:
2     def __init__(self, balance=0):
3         self.__balance = balance
4
5     def deposit(self, amount):
6         self.__balance += amount
7
8     def withdraw(self, amount):
9         if amount <= self.__balance:
10             self.__balance -= amount
11             return True
12         else:
13             return False
14
15     def get_balance(self):
16         return self.__balance
```

D 需要使用继承和多态时

类的继承和多态特性允许你创建层次结构和共享行为，从而提高代码的复用性和扩展性。例如，前面示例中的动物类可以作为一个基类，不同类型的动物（如狗和猫）可以继承这个基类并实现各自特有的行为。

E 需要与现实世界的实体进行映射时

在面向对象的设计中，类提供了一种自然的方式来模拟现实世界的实体。例如，电子商务网站中的用户、产品、订单等都可以通过类来表示，使代码更直观。

```
1 class User:
2     def __init__(self, username, email):
3         self.username = username
4         self.email = email
5
6 class Product:
7     def __init__(self, name, price):
8         self.name = name
```



```
9         self.price = price
10
11 class Order:
12     def __init__(self, user, product):
13         self.user = user
14         self.product = product
```

使用类的主要场景包括表示和操作复杂数据结构、创建多个相似对象、实现抽象和封装、使用继承和多态、以及与现实世界的实体映射。这些场景下，类可以帮助组织代码、提高复用性、简化维护和增强扩展性。

F 实例、子类关系判断

`isinstance()` 和 `issubclass()` 是两个内置函数，通常用于检查对象的类型和类之间的继承关系。这两个函数在面向对象编程中非常有用，尤其是在需要验证对象类型或实现基于类型的逻辑时。

- `isinstance()` 函数用来判断一个对象是否是一个特定类或一个类的元组中的任何一个类的实例。这意味着该函数不仅检查直接类型，还会考虑继承链。

语法：

```
1 | isinstance(object, classinfo)
```

- `object`: 要检查的对象。
- `classinfo`: 可以是一个类或类的元组，用于比较对象类型。如果 `classinfo` 是一个包含多个元素的元组，只要 `object` 是元组中任何一个类的实例，函数就会返回 `True`。

示例：

```
1 | class Fruit:
2 |     pass
3 |
4 | class Apple(Fruit):
5 |     pass
6 |
7 | class Banana(Fruit):
8 |     pass
9 |
10 | apple = Apple()
11 |
```

```

12 | print(isinstance(apple, Apple))    # True
13 | print(isinstance(apple, Banana))  # False
14 | print(isinstance(apple, Fruit))   # True, 因为 Apple 继承自 Fruit
15 | print(isinstance(apple, object))
16 | # True, 所有的 Python 类都是 object 的子类

```

- `issubclass()` 函数用来检查一个类是否是另一个类的子类（派生类）。这个函数也接受一个类或包含类的元组作为第二个参数，用于检查第一个类是否是元组中任何一个类的子类。

语法：

```

1 | issubclass(cls, classinfo)

```

- `cls`: 要检查的类。
- `classinfo`: 可以是一个类或一个包含多个类的元组。如果 `cls` 是元组中任一类的子类，函数就返回 `True`。

示例：

```

1 | print(issubclass(Apple, Fruit))    # True
2 | print(issubclass(Apple, Banana))  # False
3 | print(issubclass(Apple, (Banana, Fruit)))
4 | # True, Apple 是 Fruit 的子类

```

练习

1. 设计一个名为 `Point` 的类，来表示二维平面上的点。该类包括的属性与方法如下：

Point 类：

- 属性
 - `x (float)`: 点在 x 轴上的坐标。
 - `y (float)`: 点在 y 轴上的坐标。
- 方法
 - `__init__(self, x, y)`: 构造方法，设置点的 x 和 y 坐标。
 - `distance_to(self, other)`: 计算当前点与另一个 `Point` 对象之间的距离。

- `__str__(self)`: 返回点的字符串表示, 格式为“(x,y)”。

2. 小组项目: 股票资产组合管理系统

• 项目描述

在本项目中, 你将设计和实现一个股票资产组合管理系统。你需要创建两个主要的类: `Stock` 和 `Portfolio`。`Stock` 类将代表单个股票, 而 `Portfolio` 类将管理一个股票组合, 允许添加、删除股票, 并计算总价值。此外, 你将重载一些运算符来提供额外的功能, 如比较两个投资组合的价值。

• 目标

- 理解面向对象编程的基本概念, 包括类的定义、属性、方法和封装。
- 学习如何在 `Python` 中重载运算符。
- 应用面向对象的方法来解决实际问题。

• 项目要求

1. 类定义:

– `Stock` 类

- * 属性: `symbol` (股票代码), `quantity` (持有数量), `price` (当前价格)。
- * 方法: `value()`, 计算并返回股票的当前市值 (数量 * 价格)。

– `Portfolio` 类

- * 属性: 一个字典, 键为股票代码, 值为 `Stock` 对象。
- * 方法:
 - `add_stock(stock)`: 添加一个 `Stock` 对象到组合。如果股票已存在, 增加数量。
 - `remove_stock(symbol)`: 根据股票代码移除股票。
 - `total_value()`: 返回组合的总市值。
- * 运算符重载:
 - `__str__`: 返回组合的详细描述。
 - `__lt__`, `__gt__`, `__eq__`: 比较两个投资组合的市值。

2. 功能实现:

- 实现 **Stock** 和 **Portfolio** 类的所有方法。
- 确保可以通过运算符重载比较两个 **Portfolio** 对象。

3. 测试:

- 创建至少两个 **Portfolio** 对象，进行测试，包括：
 - * 添加和删除股票。
 - * 打印投资组合详情。
 - * 比较两个投资组合的总价值。

4. 文档和注释:

- 为你的代码编写清晰的文档。
- 添加必要的注释来解释每个类和方法的工作原理。

5. 额外挑战 (可选):

- 实现更复杂的股票分析功能，例如计算投资组合的预期回报率或风险评估。

• 提交材料

- 完整的 **Python** 代码文件。
- 一个简短的报告，描述你的设计决策和你所遇到的任何挑战，以及如何克服这些挑战。
- 代码运行结果的截图，显示测试用例的输出。

5.3 模块与包

5.3.1 模块

在 **Python** 中，模块 (**module**) 是一个包含 **Python** 变量、函数、类等对象的定义和声明的文件。文件名就是模块名加上 **.py** 后缀。模块可以包含函数、类、变量定义以及可执行代码。模块的主要目的是帮助组织代码，并使其可以重用。模块可以被其他模块或程序导入，以使用模块中定义的函数、类等对象。

5.3.2 包

包 (**package**) 是一个包含多个模块的目录。**Python** 的包允许你将代码逻辑上组织成层次化的方式，这样可以更方便地管理和命名空间的分配。一个包通常由一

个名为 `__init__.py` 的文件定义，这个文件可以为空，但它告诉 **Python** 这个目录应该被视为一个 **Python** 包。包可以包含子包和模块。

5.3.3 两者的联系

1. **组织结构**: 包和模块都是 **Python** 代码的组织形式。模块提供了一种将代码和数据封装到单个单元的方式，而包使用这些模块来进一步组织成更大的结构。
2. **代码重用**: 无论是模块还是包，它们的主要目的都是促进代码重用。通过模块和包，代码可以被多个程序或者模块重用。
3. **导入机制**: 在 **Python** 中，无论是导入包还是模块，使用的语法结构是相似的。可以通过 `import` 语句来导入它们，并使用 `from ... import ...` 语句来从特定的包或模块中导入特定的功能。

5.3.4 两者的区别

1. **层次**: 模块是单个文件，而包是一个包含多个模块的目录。包可以被视为模块的容器。
2. **文件结构**: 模块是一个 `.py` 文件，而包至少包含一个名为 `__init__.py` 的文件的目录，可能还会包含多个模块文件和子目录。
3. **用途**: 单个模块可以用来组织高度相关的函数和类。而包用来组织相关的模块，提供更复杂的功能和更大范围的代码组织。
4. **命名空间**: 包创建了一个更大的命名空间。例如，在包中，可以有多个模块，每个模块可以定义相同的函数名而不会冲突。模块内的命名空间更局限，它仅限于单个文件内。

示例

假设有一个金融资产管理的项目，项目的结构可能如下所示：

```
1 | assets/ (包)
2 |     __init__.py
3 |     bonds.py (模块, 实现债券相关功能)
4 |     stocks.py (模块, 实现股票交易相关功能)
5 |     derivatives.py (模块, 实现金融衍生品相关功能)
```

A 文件和模块详细说明

1. `bonds.py`:

- 包含与债券相关的类和函数，如债券的价值计算，到期收益率，及其风险评估等。
- 可能还包括不同类型的债券，如政府债券、公司债券等的特定功能。

2. `** stocks.py **`:

- 处理股票市场交易的函数和类，包括股票购买、销售、股票价格分析等。
- 可以扩展为包括股票市场趋势预测，历史数据分析等高级功能。

3. `** derivatives.py **`:

- 实现与金融衍生品（如期权、期货等）相关的操作和计算。
- 包括但不限于期权定价模型、风险管理工具以及衍生品交易策略的实现。

4. `__init__.py`

`__init__.py` 文件可以用来初始化包的内容，可能包括但不限于导入各个模块中定义的类和函数，以便于包的用户可以直接从包级别导入它们，例如：

```
1  ## assets/__init__.py
2  from .bonds import BondCalculator
3  from .stocks import StockMarket
4  from .derivatives import DerivativeTools
```

以下是一个简单的示例，展示如何在 `bonds.py` 模块中定义一个简单的债券计算器类：

```
1  ## bonds.py
2  class BondCalculator:
3      def __init__(self, face_value, coupon_rate, years_to_maturity):
4          self.face_value = face_value
5          self.coupon_rate = coupon_rate
6          self.years_to_maturity = years_to_maturity
7
8      def calculate_annual_coupon_payment(self):
9          return self.face_value * self.coupon_rate
```

B 调用示例

```

1 | from assets.bonds import BondCalculator # 从模块中导入 BondCalculator 类
2 |
3 | ## 创建债券计算实例
4 | bond = BondCalculator(1000, 0.05, 10)
5 |
6 | ## 计算年度利息支付
7 | print("Annual Coupon Payment:", bond.calculate_annual_coupon_payment())

```

C 调用方式

调用 Python 包或模块中定义的函数、类等对象，通常使用 `import` 语句。下图给出了三中常用的导入和调用方式：



`import toolbox`



`from toolbox import hammer`



`from toolbox import *`

图 5.2: 模块中对象导入方式

每一个模块可以看着一个工具箱，其中定义的函数、类等对象可以被视为各种工具。上图左图表示导入整个模块的方式，调用工具箱中具体工具的语句如下：

```

1 | import toolbox
2 | toolbox.hammer() # hammer 为一个函数

```

如果模块的名字较长，可以在导入时，给模块定义一个别名，例如：

```

1 | import toolbox as tb # tb 是 toolbox 模块的别名
2 | tb.hammer()

```

上图中间图表示导入模块中特定的函数或类，调用该函数的语句如下：

```

1 | from toolbox import hammer
2 | hammer()

```

上图中右图表示导入工具箱中所有函数和类，此时可以直接调用该工具箱中所有函数和类：

```
1 | from toolbox import *  
2 | hammer()  
3 | wrench()  
4 | plier()
```

但在实际使用中，不推荐使用一次性导入工具箱中所有函数和类，主要原因如下：

1. **命名冲突** 当你从多个模块导入所有内容时，很容易出现命名冲突。不同模块可能定义了相同名称的函数或类，导入时后者会覆盖前者，这会导致难以跟踪和调试的错误。
2. **可读性降低** 使用 `from module import *` 使得其他阅读代码的人难以判断某个特定的函数或类是来自哪个模块，尤其是在大型项目中。这直接影响代码的可读性。
3. **命名空间污染** 导入所有内容将大量未使用的变量、函数和类引入当前命名空间，这种“污染”可能导致意外的行为或性能问题。理想的情况是，只导入你需要的部分，保持命名空间的清洁。

5.3.5 Python 包的管理

`pip` 是 Python 的包社区用于安装和管理软件包的标准工具。`pip` 允许用户从 Python Package Index (PyPI)、其他索引以及本地包安装和管理项目依赖。以下是一些 `pip` 的常用功能：

A 安装包

使用 `pip` 安装包是非常直接的操作。例如，要安装 `numpy` 包，可以使用以下命令：

```
1 | pip install numpy
```

B 卸载包

要从 Python 环境中卸载一个包，可以使用 `uninstall` 命令。例如，卸载 `numpy` 包：

```
1 | pip uninstall numpy
```


C 包的升级

如果需要将已安装的包升级到最新版本，可以使用 `install` 命令配合 `--upgrade`（或 `-U`）参数。例如，升级 `numpy` 包：

```
1 | pip install --upgrade numpy
```

D 查看已安装的包

要列出所有已安装的包及其版本，可以使用 `list` 命令：

```
1 | pip list
```

E 查看特定包的信息

如果需要获取已安装包的详细信息，如版本、依赖、安装位置等，可以使用 `show` 命令。例如：

```
1 | pip show numpy
```

F 安装依赖文件

在项目中，常常需要使用 `requirements.txt` 文件列出所有依赖，然后可以用 `pip` 一次性安装这些依赖：

```
1 | pip install -r requirements.txt
```

G 冻结依赖

为了创建一个包含所有当前已安装包及其精确版本的 `requirements.txt` 文件，可以使用 `freeze` 命令：

```
1 | pip freeze > requirements.txt
```

H 使用不同的源

在某些情况下，由于网络问题或速度问题，可能需要使用镜像源来安装包。可以在安装时使用 `-i` 选项指定不同的源。例如，使用中国科技大学的镜像源：

```
1 | pip install -i https://pypi.mirrors.ustc.edu.cn/simple/ some-package-name
```

这些功能构成了 `pip` 的核心，使其成为管理 `Python` 包的强大工具。`Python` 还有其他包安装和管理程序，如 `conda` 等。

5.3.6 模块或包的加载过程

Python 在查找和使用包或模块时遵循一定的搜索路径与机制。这个过程涉及到 Python 解释器如何定位和加载指定的模块或包。后续谈到 `import` 导入模块或包时，不再做特别区分，都视为模块。以下是详细的步骤：

1. 导入系统

当你在 Python 中使用 `import` 语句时，Python 解释器会按照一定的顺序搜索模块：

```
1 | import yfinance
```

此语句会导入名为 `yfinance` 的模块。

如果此时出现如下错误提示：

```
1 | ModuleNotFoundError: No module named 'yfinance'
```

说明 Python 本地系统还没有安装该模块。可以在 Jupyter notebook 使用如下命令安装：

```
1 | !pip install yfinance
```

2. 搜索顺序

Python 解释器搜索模块的顺序如下：

1. **内建模块**：首先，Python 会在其内建模块中查找。这些模块是用 C 语言编写并且编译到 Python 解释器中，不需要任何外部文件。
2. **sys.path**：如果不在内建模块中找到，Python 会根据 `sys.path` 列表中的目录来查找模块。`sys.path` 是一个字符串列表，主要包含了以下元素：
 - 当前目录（即执行 Python 命令时的目录）。
 - `PYTHONPATH` 环境变量中指定的目录（如果设置的话）。
 - 标准库目录。
 - 安装的第三方包目录（通常位于 `site-packages` 目录下）。

3. 导入机制

当 Python 在 `sys.path` 中找到对应的模块后，它会按照以下步骤进行导入：

- **加载**: Python 加载模块文件（.py 文件、.pyc 文件等）。
- **编译**: 如果需要, Python 将源代码编译成字节码。
- **执行**: Python 执行编译后的字节码以初始化模块。这可能包括定义函数、类和其他变量。

4. 包的导入

如果导入的是一个包（即包含 `__init__.py` 文件的目录），Python 会：

- 查找并执行 `__init__.py` 文件。
- 根据导入的模块名称，进一步查找子模块或子包。

例如，导入 `os.path` 时，Python 首先导入 `os` 包，然后定位到 `os` 包中的 `path` 模块。

5. 缓存

为了提升性能，Python 通常会在 `__pycache__` 目录下缓存编译后的模块（以 .pyc 文件存储），使得下次导入时无需再次编译。

练习

1. 创建一个名为 `toolbox.py` 的 Python 模块，该模块将包含三个函数，分别用于模拟三种不同的工具：锤子、扳手和钳子的基本操作。然后，你需要在另一个 Python 脚本中使用这个模块，测试示例中的代码。
2. 练习使用 Python 的 `math` 包来解决一个数学计算的问题。
 1. 查看 `math` 包有哪些函数或数学常量。
 2. 如何了解包中某个函数的调用方法。
 3. 计算圆心角为 30 度，半径为 10 的圆弧的弧长。

6

文件读写

在本章中，我们将探讨 **Python** 中的文件读写操作，涵盖从基础的文本数据处理到复杂的文件格式如 **Excel** 和 **Pdf** 的处理。通过本章的学习，读者将能够掌握各种文件操作技巧，并应用于数据处理和自动化任务中。

在编程中，文件读写是一项基础且重要的功能，它类似于我们在现实生活中记笔记和查阅笔记的过程。这里将通过一些日常生活中的比喻，帮助你理解为什么编程需要文件读写。

1. 保存信息以便将来使用

想象一下，你正在做菜时查到了一个很棒的食谱，如果你不把它记下来，下次想再做这道菜时可能就记不清楚了。在编程中，我们也经常需要保存数据（例如用户信息、游戏分数、交易记录等），以便在未来可以再次使用。这就像是我们把食谱写在笔记本上，需要的时候可以翻出来查看。

2. 数据的持久化

假设你有一个小商店，每天的销售记录如果只是口头记忆，很容易出错或遗忘，所以你会把它们记在账本上。在计算机程序中，如果不将数据写入文件，那么程序关闭后，所有的数据都会消失（因为它们是暂时存储在内存中的，而内存的数据在断电后也会全部消失）。通过写入文件，数据就可以“持久化”，即使电脑关机后再开机，这些数据仍然存在。

3. 数据共享

设想你是一个团队的一员，需要和团队里其他成员共享你的工作报告。你可能会把报告写入文档并通过邮件发送给每个人。在编程中，文件也常常用来作为不同程序、不同用户之间共享和交换数据的手段。例如，一个程序可以生成一个文件，然后其他程序读取这个文件，实现数据的共享和传递。

6.1 文本文件的读写

6.1.1 文本文件的特征

文本文件是一种常用的电脑文件类型，它以标准的文本形式存储数据。这里的“文本”指的是可以阅读的字符序列，例如字母、数字和标点符号。文本文件具有如下特点：

1. **人类可读性**：文本文件的内容可以直接用文本编辑器（如记事本、VSCode 等）打开，内容对人类是可读的。
2. **字符编码**：文本文件的数据是通过某种字符编码格式存储的，常见的字符编码有 ASCII、UTF-8、GBK 等。这些编码决定了如何将字符数据转换为计算机存储的字节。
3. **扩展名**：文本文件通常具有如 `.txt`（最常见）、`.csv`、`.log` 等扩展名。这些扩展名帮助操作系统和用户识别文件类型。
4. **无格式设置**：纯文本文件不包含非文本内容（如字体大小、颜色、图像等），仅包含纯粹的文本内容和可能的一些基本格式化，如换行符。

6.1.2 用途

文本文件由于其简单性和广泛的兼容性，被广泛用于多种场景：

- **配置文件**：许多程序和应用使用文本文件来存储配置信息。
- **日志文件**：系统和应用程序常常将运行日志保存在文本文件中，便于事后审查。
- **数据交换**：文本文件如 CSV 格式，常用于数据导出和导入，因为它们易于由人类和机器解读。
- **编程**：源代码文件通常就是文本文件，包含了可以被编译器或解释器读取的代码。

6.1.3 打开和关闭文件

当我们处理文件时，如读取一个文件的内容或向文件中写入数据，我们实际上是在与一个外部资源（文件）进行交互。这个过程可以用日常生活中的一个简单比喻来说明：

想象你有一个文件柜，里面存放着很多文件夹。每当你需要查看某个文件夹中的信息或者向文件夹中添加信息时，你首先需要做的是打开这个文件夹。完成工作后，你需要将文件夹关闭，以确保文件夹的内容不会丢失或受损，并且文件柜的安全也得以保障。在计算机的世界里，打开和关闭文件的过程与此类似。

A 为什么要“打开”和“关闭”文件？

打开文件实际上是告诉计算机：“我现在需要开始使用这个文件。”这时，计算机准备好所有必要的资源，比如内存和文件的访问权限，以确保你可以顺利地读取或写入文件。打开文件的过程还包括以下几点：

- **确认文件存在：**在文件打开之前，计算机需要确认该文件是否存在于指定的位置。
- **设置访问模式：**你需要告诉计算机你打算如何使用这个文件（只读取、写入数据或两者都做），这样计算机才能正确地管理对文件的操作。
- **资源分配：**打开文件意味着计算机需要分配资源，如内存，来存储文件内容，以便程序可以快速访问。

文件使用完毕后，关闭文件同样重要。关闭文件的主要目的包括：

- **保存数据：**特别是在写入文件时，关闭文件能确保所有的数据都被正确写入文件中，而不是停留在计算机的内存里。
- **释放资源：**打开文件时占用的资源（如内存）在文件关闭时会被释放，这样这些资源就可以用于其他任务了。
- **避免数据损坏：**如果多个程序或多次操作试图同时修改一个文件，可能会导致数据损坏。关闭文件有助于管理对文件的访问，防止此类问题发生。
- **确保文件状态更新：**关闭文件有助于更新系统中关于文件的状态信息，如修改时间等。

B 打开文件

要打开一个文件，可以使用 Python 的 `open()` 函数。这个函数的基本语法如下：

```
1 | f = open(file_name, mode='r', encoding=None)
```

- **file_name**: 这是一个字符串，表示要打开的文件的路径和文件名。
- **mode**: 文件打开模式。默认为 'r'，即只读模式。常用模式包括：
 - 'r': 只读模式。如果文件不存在，将抛出一个错误。
 - 'w': 写入模式。如果文件已存在，会被覆盖。如果文件不存在，会创建一个新文件。
 - 'a': 追加模式。如果文件已存在，写入的数据会被追加到文件末尾。如果文件不存在，会创建一个新文件。
 - 'r+': 读写模式。文件必须存在。
 - 'w+': 读写模式。如果文件存在，会被覆盖；如果文件不存在，会创建新文件。
 - 'a+': 读写模式。如果文件存在，写入的数据会被追加到文件末尾；如果文件不存在，会创建一个新文件。
- **encoding**: 指定文件的编码格式，如 'utf-8'。这对于读写非 ASCII 字符的文本文件特别重要。

C 关闭文件

打开文件后，完成读写操作，需要关闭文件。可以通过调用文件对象的 `close()` 方法来实现。关闭文件是一个好习惯，可以释放系统资源，也可以确保写入到文件的数据得到保存并且文件状态被正确更新。

```
1 | f.close()
```

D 使用 with 语句自动关闭文件

Python 提供了 `with` 语句，可以自动管理文件的打开和关闭，就像有一个自动门一样。当你进入某个区域时门自动打开，离开时门自动关闭，你不需要亲自去锁门。这样做不仅方便，而且可以减少因忘记关闭文件而造成的问题。当 `with` 代码块执行完成后，即使发生错误，Python 也会确保文件被正确关闭。因此，使用 `with` 语句是处理文件的推荐方式。

```
1 with open('example.txt', 'r', encoding='utf-8') as f:
2     content = f.read()
3
4 # 文件此时已自动关闭，无需手动调用 f.close()
```

使用 `with` 语句不仅可以使代码更安全，还可以使代码更简洁，减少忘记调用 `close()` 方法导致的资源泄露问题。

6.1.4 文件读写

A 读取文件内容

下表列出了用于读取文件内容的主要方法：

| 函数 | 描述 |
|--------------------------|---|
| <code>read()</code> | 读取整个文件的内容，如果指定参数，如 <code>read(size)</code> ，则读取指定数量的字符。 |
| <code>readline()</code> | 读取文件的下一行。 |
| <code>readlines()</code> | 读取文件中的所有行，并将它们作为列表的每个元素返回。 |

B 写入文件内容

下表展示了用于写入文件内容的方法：

| 函数 | 描述 |
|--------------------------------|-----------------------------------|
| <code>write(s)</code> | 将字符串 <code>s</code> 写入文件。 |
| <code>writelines(lines)</code> | 将一个字符串列表 <code>lines</code> 写入文件。 |

C 处理文件指针

文件中的数据是连续存储的，从文件的第一个字节到最后一个字节没有任何中断。这种存储方式使得我们可以从任何位置开始读取或写入文件，但也带来了一定的挑战，特别是在需要精确控制文件操作的位置时。文件指针是一个用来追踪当前读写位置的工具，就像是你读书时用手指标记你所在的页码一样。在操作电脑上的文件时，文件指针帮助计算机知道从哪里开始读取或写入数据。文件指针可以这样理解：

- 1. **打开文件时的默认位置：**当你打开一个文件用来读写时，文件指针默认会放在文件的开始位置，就好像你打开一本新书从第一页开始阅读一样。
- 2. **读取或写入数据时的移动：**当你开始读取或写入文件内容时，文件指针会随着你的读写进度向前移动。比如，你读了 **50** 个字符，文件指针就会从文件的开始位置移动到第 **50** 个字符的位置。
- 3. **自由定位：**如果你想跳到文件的某个特定位置开始读取或写入，就可以通过调整文件指针的位置做到。这就像是你想跳到书本的某个章节开始阅读，你会先翻到那个章节的起始页。
- 4. **查看当前位置：**任何时候，如果你想知道现在文件指针处于文件中的什么位置，都可以查看它。这有点像查看你的书签现在在书的哪一页。

下表列出了文件指针相关的函数：

| 函数 | 描述 |
|-------------------------------------|--|
| <code>seek(offset, whence=0)</code> | 移动文件指针到指定的位置。 <code>offset</code> 表示偏移量， <code>whence</code> 是偏移的起点（ <code>0</code> 表示文件开头，默认值； <code>1</code> 表示当前位置； <code>2</code> 表示文件末尾）。 |
| <code>tell()</code> | 返回当前文件指针的位置。 |

D 示例

```
1  ## 步骤 1: 创建并写入文件
2  ## 打开文件，如果不存在则创建，模式为 'w' 表示写入，文件内容会被覆盖如果已存在
3  with open('example.txt', 'w') as file:
4      file.write("Hello, world!\n")
5      file.write("This is a new line.\n")
6      file.writelines(["Line one\n", "Line two\n", "Line three"])
7
8  ## 步骤 2: 读取文件内容
9  ## 打开文件，模式为 'r' 表示读取
10 with open('example.txt', 'r') as file:
11     # 读取整个文件内容
12     content = file.read()
13     print(" 读取的全部内容:")
14     print(content)
15
16 ## 步骤 3: 演示逐行读取
17 with open('example.txt', 'r') as file:
18     print(" 逐行读取内容:")
```

```
19     for line in file:
20         print(line.strip()) # strip() 用于移除行尾的换行符
21
22 ## 步骤 4: 文件指针操作
23 with open('example.txt', 'r') as file:
24     print(" 文件指针当前位置 (开始) :", file.tell())
25     # 读取一行
26     file.readline()
27     print(" 读取第一行后文件指针位置:", file.tell())
28     # 移动文件指针回到文件开头
29     file.seek(0)
30     print(" 移动指针回开始位置后:", file.tell())
31
32 ## 所有文件操作完成后, 文件自动关闭由于使用了 with 语法
```

练习

1. 编写一个 Python 程序, 要求用户输入自己的名字和年龄, 然后将这些信息保存到一个文本文件中。文件中的内容应该具有明确的格式, 例如: “Name: [用户名], Age: [年龄]”。文件要保留每次运行的结果。
2. 编写一个 Python 程序, 要求用户输入已学课程的名称, 直到用户输入 “DONE” 时停止。将所有课程名称保存到一个文本文件中, 每门课程名称占一行。

6.2 Python 对象文件

文本文件存储的是纯文本对象, 而 Python 对象文件存储的是 Python 中各种复杂的数据对象, 如列表、字典和各种自定义对象等。通常将对象的描述信息转换为可以被存储或传输的形式过程称为**序列化**, 而**反序列化**是指将这种形式的数据恢复为原始对象的过程。让我们用一种更通俗易懂的方式来解释这两个过程。

对象序列化: 想象一下, 你有一个玩具箱, 里面装满了各种玩具 (这些可以看作是程序中的对象, 比如列表、字典、自定义的数据类型等)。现在, 你需要搬家, 并希望把这个玩具箱里的内容安全地带到新家去。序列化就像是把这些玩具拆分开, 按照一定的方式打包到几个箱子里, 这样它们就更容易被搬运。在计算机程序中, “打包” 这些对象的过程就是序列化。序列化后的对象转变成一串可以存储到文件或者通过网络传输的数据。

对象反序列化：当你到了新家，你需要把这些被打包的箱子里的玩具重新拿出来，组装回原来的状态，放回玩具箱中。这个从打包箱子中取出玩具并恢复它们原始状态的过程，在计算机程序中称为反序列化。通过反序列化，你的程序可以重新获取到之前保存的对象和数据，就像它们从未被拆分和打包过一样。

Python 对象的序列化和反序列化通常用 `pickle` 模块来实现。`pickle` 是 Python 的标准库之一，用于实现 Python 对象的序列化和反序列化。如下是对 Python 对象文件的特征和应用场景的说明：

- 特征

- **数据结构的完整性：**使用 `pickle` 序列化可以保存几乎所有的 Python 数据类型，包括自定义对象和复杂的数据结构。当这些对象被反序列化时，能够保持其原有的状态和数据结构，不需要如在文本文件中那样进行复杂的解析和转换。
- **简单易用：**`pickle` 的使用非常简单，只需几行代码就可以序列化和反序列化复杂对象。这相比于文本文件，你可能需要编写或使用复杂的解析器来转换数据格式，显得更为方便。
- **空间效率：**对于复杂的对象，`pickle` 文件可能比相应的文本表示更紧凑，因为它是二进制格式，可以有效地压缩数据，但它们无法被人直接阅读。
- **安全性问题：**`pickle` 反序列化时存在安全风险。如果 `pickle` 数据来自不可信的源，它可能包含恶意代码，反序列化时可能会执行。
- **可移植性和兼容性：**`pickle` 文件是 Python 特有的，其他编程语言很难读取和解析。此外，不同版本的 Python 或 `pickle` 自身的不同版本间可能存在兼容性问题。

- 应用场景

在某些情况下，保存和加载 Python 对象的完整状态非常重要，尤其是在以下场景中：

- **持久化复杂对象：**对于需要长时间存储并且可以快速加载的复杂对象（例如机器学习模型、大型数据结构等），使用 `pickle` 序列化是一个快速且有效的方法。
- **临时数据交换：**在同一 Python 环境中的不同程序或程序的不同部分之间交换复杂数据结构时，`pickle` 可以提供一個方便的解决方案。

- **科学计算和数据分析:** 在数据科学和机器学习项目中，经常需要保存中间结果（如模型、训练数据集等），使用 `pickle` 可以帮助研究人员保存工作状态，随后从中断处恢复工作

- **对象的序列化和反序列化**

```
1 import pickle
2
3 data = {'key': 'value', 'numbers': [1, 2, 3, 4]}
4 with open('data.pkl', 'wb') as file:
5     # 将数据对象序列化并写入到文件
6     pickle.dump(data, file)
```

在上述代码中，`data.pkl` 文件就是一个“Python 对象文件”。这个文件是二进制格式的，其中包含了 `pickle` 序列化后的数据。

```
1 import pickle
2
3 with open('data.pkl', 'rb') as file:
4     # 从文件中读取并反序列化数据对象
5     data_loaded = pickle.load(file)
6 print(data_loaded)
```

练习

1. 构造一个复杂的字典，用于描述学生的信息。这个字典应包括学生的姓名、年龄、所在班级、兴趣爱好（列表形式），以及各科成绩（另一个字典）。
 1. 使用 `pickle` 将这个学生信息的字典序列化并保存到一个文件中。
 2. 读取这个文件，并使用 `pickle` 反序列化，恢复原始的学生信息。
 3. 打印反序列化后的数据，确保它与原始数据完全相同。
2. 定义一个简单的类 `Person`，包含姓名和年龄两个属性。
 1. 创建该类的一个实例。
 2. 使用 `pickle` 对这个实例进行序列化，保存到一个文件中。
 3. 从文件中读取数据并反序列化，恢复 `Person` 类的实例。
 4. 打印反序列化后实例的属性，确保它与原始实例的属性相同。

6.3 Excel 文件

Excel 文件, 通常以 `.xlsx` 或 `.xls` 为文件扩展名, 是由 Microsoft 开发的电子表格程序。Excel 因其支持广泛的内置函数和公式, 提供多种图表和数据分析工具, 在金融行业中应用极为广泛, 专业的金融数据平台, 如 Bloomberg, Thomson Reuters (Refinitiv) 等, 都提供专门的 Excel 插件, 以方便用户直接在 Excel 环境中访问、分析和操作实时和历史数据。这些插件强化了 Excel 的功能, 使其成为一个更加强大的金融分析工具。

在 Python 中, 对 Excel 文件的处理和操作主要是通过第三方的工具包来实现的, 常用的有如下两个工具包:

6.3.1 openpyxl

openpyxl 是一个用于读/写 Excel 2010 xlsx/xlsm 文件的 Python 库, 支持创建新表格、读取表格、修改已有的 xlsx 文档。

A 安装

使用 pip 安装 openpyxl, 可以执行如下操作:

```
1 | pip install openpyxl
```

B 使用简介

- 读取数据

```
1 | from openpyxl import load_workbook
2 | ## 加载现有的 xlsx 文件
3 | wb = load_workbook('example.xlsx')
4 | ## 激活当前活跃的工作表
5 | ws = wb.active
6 | ## 读取特定单元格的值
7 | print(ws['A1'].value)
```

- 写入数据

```
1 | from openpyxl import Workbook
2 | ## 创建一个新的工作簿
```

```
3 | wb = Workbook()
4 | ## 获取当前活跃的工作表
5 | ws = wb.active
6 | ## 向特定单元格写入数据
7 | ws['A1'] = 'Hello, World!'
8 | ## 向特定索引位置的单元格写入数据
9 | c1 = ws.cell(row=2, column=2)
10 | c2.value = 'Hello, B2'
11 |
12 | ## 保存工作簿到文件
13 | wb.save('example.xlsx')
```

想了解更多 `openpyxl` 的功能，访问其 [在线教程](#)。

6.3.2 XlsxWriter

`xlsxwriter` 是一个创建新的 Excel 文件并写入数据的库，非常适合于生成复杂的 Excel 报表。`xlsxwriter` 支持添加图表、图片、自定义格式等高级功能。

A 安装

```
1 | pip install XlsxWriter
```

B 使用简介

```
1 | import xlsxwriter
2 | ## 创建一个新的 Excel 工作簿
3 | workbook = xlsxwriter.Workbook('example.xlsx')
4 | ## 添加一个工作表
5 | worksheet = workbook.add_worksheet()
6 | ## 在 A1 单元格写入 'Hello'
7 | worksheet.write('A1', 'Hello')
8 | ## 在指定索引位置的表格写入数据
9 | worksheet.write(2, 0, 123)
10 | worksheet.write(3, 0, 123.456)
11 | ## 插入图片
12 | worksheet.insert_image('B5', 'python.png')
13 |
14 | ## 关闭工作簿，保存文件
15 | workbook.close()
```

想了解更多 `XlsxWriter`，请访问其 [在线说明](#)。

Python 中还有其它工具包支持 Excel 文件的操作和读写，如处理旧的.xls 格式文件，`xlrd` 用于读取 Excel 文件，而 `xlwt` 用于写入 Excel 文件。`pyexcel` 提供了一个简化的接口来读写多种扩展名的表格文件（包括 `xlsx`, `xls`, `csv`）。

选择哪个库主要取决于你需要处理的 Excel 文件类型（`xls` 还是 `xlsx`），以及你希望执行的操作（如是否需要支持复杂的单元格格式化、图表等）。对于大多数基本操作，`openpyxl` 已经足够使用。对于复杂的报表生成，`xlsxwriter` 提供了广泛的功能来满足需求。

练习

1. Excel 数据处理与报表生成：

a. 读取数据

- (1) 下载或创建一个 Excel 文件（例如名为 `sales_data.xlsx`），该文件包含一个工作表：`Sales`。
- (2) `Sales` 工作表具有以下列：`Date`, `Product`, `Quantity`, `Unit Price`。

b. 处理数据

- (1) 计算每个产品的总销售额（`Quantity * Unit Price`）。
- (2) 按产品分类汇总销售额。

c. 生成报表

- (1) 在 `Summary` 工作表中创建报表，该报表展示每个产品的总销售额。
- (2) 为每个产品添加一行，包含产品名称和对应的总销售额。

d. 样式设置

- (1) 设置标题行的字体为粗体。
- (2) 设置货币列的格式为货币格式。

e. 保存和关闭文件

- (1) 保存对 Excel 文件的更改。
- (2) 安全关闭工作簿。

代码框架如下，请补充完整：


```
1 from openpyxl import load_workbook
2 from openpyxl.styles import Font, Alignment, numbers
3
4 ## 加载工作簿
5 wb = load_workbook('sales_data.xlsx')
6
7 ## 读取 Sales 工作表
8 sales_sheet = wb['Sales']
9
10 ## 创建一个字典来存储每个产品的总销售额
11 product_sales = {}
12
13 ## 遍历数据（跳过标题行）
14 for row in sales_sheet.iter_rows(min_row=2, min_col=1, max_col=4,
15     values_only=True):
16     date, product, quantity, unit_price = row
17     # TODO: 计算总销售额并累加到 product_sales 字典
18
19 ## 读取或创建 Summary 工作表
20 if 'Summary' in wb.sheetnames:
21     summary_sheet = wb['Summary']
22     # TODO
23
24 else:
25     summary_sheet = wb.create_sheet('Summary')
26
27 ## TODO: 将汇总数据写入 Summary 工作表
28 ## TODO: 填写表头名称
29
30
31 ## TODO: 设置标题行的样式
32
33 ## TODO: 填写产品的总销售额数据并设置表示货币的单元格格式
34
35
36 ## 保存并关闭工作簿
37 wb.save('updated_sales_data.xlsx')
38 wb.close()
39
40 print(" 报表已生成并保存！ ")
```

6.4 PDF 文件

PDF (Portable Document Format, 便携式文档格式) 是由 Adobe 在 1993 年开发的一种文件格式, 用于以一种独立于应用软件、硬件和操作系统的方式呈现文档内容, 包括文本格式和图像。得益于其安全性、格式保持一致性和易于共享的特点, PDF 文件在金融行业中应用广泛, 特别是银行、投资公司和其他金融机构常用 PDF 格式发送账户对账单、年度报告、季度报告和投资总结。使用 PDF 可以确保所有接收者看到的格式相同, 无论他们使用什么设备或操作系统。

Python 中有众多的工具包支持对 PDF 文件的读写, 但各工具包都有各自的特点, 有侧重页面内容 (文本、图片、表格) 提取的, 有专长于页面操作 (分离、合并、裁剪、旋转), 还有的支持复杂交互式表单处理和动态内容生成。本节主要介绍能帮助处理财务报告需要的 PDF 文件读写操作的两种工具包。

6.4.1 PyPDF2

PyPDF2 是一个免费且开源的纯 Python PDF 库, 能够拆分、合并、裁剪以及转换 PDF 文件的页面。它还可以向 PDF 文件添加自定义的标注、水印等。PyPDF2 也能从 PDF 文件中提取文本、图片和元数据。完整的功能请查阅  PyPDF2 在线文档。

A 安装

```
1 | pip install pypdf2
```

B 使用简介

读取 PDF 文档

从 <https://berkshirehathaway.com/2023ar/2023ar.pdf> 下载财务报告, 并提取指定页的文本。

```
1 | from PyPDF2 import PdfReader
2 |
3 | # 下载 https://berkshirehathaway.com/2023ar/2023ar.pdf
4 | reader = PdfReader("2023ar.pdf")
5 | number_of_pages = len(reader.pages)
6 | page = reader.pages[0]
```

```
7 text = page.extract_text()
8
9 print(text)
```

输出结果如下：

```
1 BERKSHIRE HATHAWAY INC.
2 2023
3 ANNUAL REPORT
```

拆分 PDF 文件

```
1 from PyPDF2 import PdfWriter, PdfReader
2
3 reader = PdfReader("2023ar.pdf")
4
5 for page_number in range(2, 4): # 将 PDF 文件的第 2,3 页拆分为单独的页面文件
6     writer = PdfWriter()
7     writer.add_page(reader.pages[page_number])
8
9     with open(f"page_{page_number + 1}.pdf", "wb") as output_pdf:
10         writer.write(output_pdf)
```

合并 PDF 文件

```
1 from PyPDF2 import PdfMerger
2
3 merger = PdfMerger()
4
5 # 添加多个 PDF 文件
6 merger.append("page_3.pdf")
7 merger.append("page_4.pdf")
8
9 # 写入合并后的文件
10 merger.write("merged.pdf")
11 merger.close()
```

6.4.2 fpdf2

fpdf2 是 FPDF 的一个改进版，提供了更丰富的功能和更好的 Unicode 支持。它用于生成 PDF 文件，适合快速创建简单的 PDF 文档。支持页面管理、文本处理、图像插入、意见、多语言文本等，可以满足基本金融图文报告自动生成的需求。需要对 fpdf2 更多的了解，可以访问 [用户手册](#)。

A 安装

```
1 | pip install fpdf2
```

B 使用实例

```
1 | from fpdf import FPDF
2 |
3 | # 创建 PDF 对象
4 | pdf = FPDF()
5 |
6 | # 添加一页
7 | pdf.add_page()
8 |
9 | # 添加中文字体
10 | pdf.add_font('Songti', '', '/Users/Name/Library/Fonts/SimHei.ttf', uni=True)
11 | # 如上设置使用于 MacOS, Windows 系统需要修改字体文件路径
12 | pdf.set_font('Songti', size=12)
13 |
14 | # 添加文本
15 | pdf.cell(200, 10, txt=" 示例 PDF 文档", ln=True, align='C')
16 |
17 | # 添加更多文本
18 | pdf.cell(200, 10, txt=" 这是一个使用 fpdf2 创建的 PDF 示例。",
19 |         ln=True, align='L')
20 |
21 | # 插入图片
22 | pdf.image("example.png", x=10, y=30, w=50)
23 |
24 | # 保存 PDF 文件
25 | pdf.output("simple_example.pdf")
```

练习

1. 创建包含公司信息的 PDF

使用 fpdf2 创建一个 PDF 文件，包含以下内容：

a. 公司名称和业务简介：

- 选择一家上市公司。
- 编写一段简短的业务简介。

b. 公司图标：

- 下载公司的图标（例如，PNG 格式）。
- 将图标插入到 PDF 文件中。

c. 公司股价图：

- 从财务网站下载公司过去一年股价数据，采用 Excel 绘制股价图。
- 将图形保存并插入到 PDF 文件中。

7

异常与错误处理

异常是指在程序运行过程中出现的非正常、非预期的事件，它们通常是可以被捕获和处理的。异常通常是由程序中的错误、用户输入不当或其他不可预见的情况引起的。例如，试图除以零、访问不存在的文件或超出数组边界等都可能引发异常。

错误通常指程序中的严重问题，往往是无法在运行时被程序处理的。这些问题通常是由编程错误、环境问题或系统故障引起的。错误包括但不限于内存不足 (`OutOfMemoryError`)、栈溢出 (`StackOverflowError`) 等。

在大规模的程序编写和运行过程中，异常和错误通常都是不可避免的。了解不同的异常类型和处理不同的错误情况是保证代码稳定运行的要求。

7.1 Python 异常类型

Python 提供了许多内置的异常类型，帮助开发者更好地捕获和处理各种错误。常见的异常和错误都定义在 `__builtins__` 模块中，`dir(__builtins__)` 可以查到包含异常和错误的列表。

以下是一些常用的异常类型：

7.1.1 BaseException

`BaseException` 是所有异常的基类。所有内置异常都继承自这个类。通常不直接使用这个类，而是使用其子类。

7.1.2 Exception

`Exception` 是所有非系统退出异常的基类。大多数用户自定义异常也应该继承自这个类。

7.1.3 常见内置异常

A `ArithmeticError`

这是所有数值计算错误的基类，包括以下子类：

- `ZeroDivisionError`: 当除法或模运算的第二个操作数为零时引发。

```
1 | try:
2 |     result = 10 / 0
3 | except ZeroDivisionError as e:
4 |     print(f"Error: {e}")
```

- `OverflowError`: 当数值运算的结果超出表示范围时引发。

B `AttributeError`

当尝试访问对象的不存在的属性时引发。

```
1 | try:
2 |     obj = None
3 |     obj.some_method()
4 | except AttributeError as e:
5 |     print(f"Error: {e}")
```

C `ImportError`

当导入模块失败时引发。

```
1 | try:
2 |     import non_existent_module
3 | except ImportError as e:
4 |     print(f"Error: {e}")
```

D `IndexError`

当使用的索引超出序列的范围时引发。

```
1 try:
2     lst = [1, 2, 3]
3     print(lst[5])
4 except IndexError as e:
5     print(f"Error: {e}")
```

E KeyError

当使用的键在字典中不存在时引发。

```
1 try:
2     d = {'key': 'value'}
3     print(d['non_existent_key'])
4 except KeyError as e:
5     print(f"Error: {e}")
```

F TypeError

当操作或函数应用于不适当类型的对象时引发。

```
1 try:
2     result = 'string' + 10
3 except TypeError as e:
4     print(f"Error: {e}")
```

G ValueError

当操作或函数接收到具有正确类型但不合适的值时引发。

```
1 try:
2     int_val = int('abc')
3 except ValueError as e:
4     print(f"Error: {e}")
```

H FileNotFoundError

当试图打开的文件不存在时引发。

```
1 try:
2     with open('non_existent_file.txt', 'r') as file:
3         content = file.read()
4 except FileNotFoundError as e:
5     print(f"Error: {e}")
```

I StopIteration

当迭代器没有更多项目时引发，通常在 `next()` 函数调用时。

```
1 try:
2     it = iter([1, 2, 3])
3     while True:
4         print(next(it))
5 except StopIteration:
6     print("Iteration finished")
```

J NameError

当局部或全局名字（变量）找不到时引发。

```
1 try:
2     print(non_existent_var)
3 except NameError as e:
4     print(f"Error: {e}")
```

K 自定义异常

除了内置异常，Python 还允许开发者定义自己的异常。自定义异常通常继承自 `Exception` 类。

```
1 class CustomError(Exception):
2     pass
3
4 try:
5     raise CustomError("This is a custom error")
6 except CustomError as e:
7     print(f"Error: {e}")
```

7.2 异常处理机制

异常是在程序运行过程中发生的错误情况，会导致程序的正常执行流程被打断。如果异常没有被处理，程序将会终止并显示错误信息。

7.2.1 捕获和处理异常

对于异常通常采用 `try` 和 `except` 块来捕获并处理。基本语法如下：


```
1 try:
2     # 可能引发异常的代码
3 except 异常类型1 as e:
4     # 处理异常的代码
5     ...
6 else:
7     # 没有异常发生时执行的代码
8 finally:
9     # 代码无论是否发生异常都会执行
10
```

A 示例

```
1 def divide(x, y):
2     try:
3         result = x / y
4     except ZeroDivisionError:
5         print("division by zero!")
6     else:
7         print("result is", result)
8     finally:
9         print("executing finally clause")
10
11 divide(2, 1)
12
13 divide(2, 0)
14
15 divide("2", "1")
```

许多 Python 库和工具包都定义了自己的异常，以便更清晰地表示特定错误情况。这些自定义异常通常继承自 Python 内置的异常类，例如 `Exception` 或 `RuntimeError`。处理这些自定义异常的办法通常包括捕获、记录和响应错误，并根据需要采取相应处理措施。

7.2.2 抛出异常

可以使用 `raise` 语句显式地抛出异常。

A 示例

```
1 def divide(x, y):
2     if y == 0:
3         raise ValueError("Division by zero is not allowed")
```

```
4     return x / y
5
6 try:
7     result = divide(10, 0)
8 except ValueError as e:
9     print(f"Caught an exception: {e}")
```

7.3 错误调试技巧

在 Python 编程中，调试是确保代码正确性和效率的关键步骤。以下是一些常用的调试技巧和方法，帮助你更有效地发现和解决代码中的错误。

7.3.1 使用 print 语句

最简单的调试方法是使用 `print` 语句输出变量的值和程序执行的流程。

```
1 def add(a, b):
2     print(f"a: {a}, b: {b}")
3     return a + b
4
5 result = add(2, 3)
6 print(f"Result: {result}")
```

7.3.2 使用 assert 语句

`assert` 语句用于在代码中添加检查点，以确保某些条件为真。如果条件为假，会引发 `AssertionError`。

```
1 def divide(a, b):
2     assert b != 0, "b cannot be zero"
3     return a / b
4
5 print(divide(10, 2))
6 print(divide(10, 0)) # This will raise an AssertionError
```

7.3.3 使用 logging 模块

`logging` 模块提供了比 `print` 更强大的日志记录功能，可以设置不同的日志级别（DEBUG、INFO、WARNING、ERROR、CRITICAL）。

```
1 import logging
2
3 logging.basicConfig(level=logging.DEBUG)
4
5 def add(a, b):
6     logging.debug(f"Adding {a} and {b}")
7     return a + b
8
9 result = add(2, 3)
10 logging.info(f"Result: {result}")
```

7.3.4 使用 pdb 模块

`pdb` 是 Python 的内置调试器，允许你逐步执行代码、查看变量和设置断点。建议详细阅读 [pdb 文档](#)，了解 `pdb` 的主要功能和使用方法。

A 启动调试器

在代码中插入 `pdb.set_trace()` 来启动调试器或插入内置函数 `breakpoint()`。

```
1 import pdb
2
3 def add(a, b):
4     pdb.set_trace()
5     return a + b
6
7 result = add(2, 3)
8 print(f"Result: {result}")
```

B 常用命令

- `n` 或 `next`: 执行下一行代码。
- `c` 或 `continue`: 继续执行，直到遇到下一个断点。
- `l` 或 `list`: 显示当前行及其周围的代码。
- `p` 或 `print`: 打印变量的值。
- `q` 或 `quit`: 退出调试器。

7.3.5 使用 IDE 的调试工具

现代集成开发环境（IDE）如 PyCharm、Visual Studio Code 和 Eclipse 等都提供了强大的调试工具。这些工具通常包括：

- 设置断点
- 逐步执行代码
- 监视变量
- 调用堆栈查看

使用这些调试工具可以更直观和高效地进行调试。

7.3.6 单元测试和测试驱动开发

编写单元测试可以帮助你捕捉和修复错误。使用框架如 `unittest`、`pytest` 可以方便地进行测试。

A unittest

`unittest` 是 Python 标准库中的一个单元测试框架，它提供了一套完整的工具来创建和运行测试。

```
1 | # test_script.py
2 | import unittest
3 |
4 | def add(a, b):
5 |     return a + b
6 |
7 | class TestMath(unittest.TestCase):
8 |     def test_add(self):
9 |         self.assertEqual(add(2, 3), 5)
10 |        self.assertEqual(add(-1, 1), 0)
11 |        self.assertEqual(add(-1, -1), -2)
12 |
13 | if __name__ == '__main__':
14 |     unittest.main()
```

运行测试：

```
1 | python test_script.py
```

B pytest

`pytest` 是一个功能强大且易于使用的测试框架，支持简单的单元测试和复杂的功能测试。它有很多扩展和插件，可以极大地扩展其功能。通过 `pip install pytest` 安装该工具包。

```
1 | # test_math.py
2 | def add(a, b):
3 |     return a + b
4 |
5 | def test_add():
6 |     assert add(2, 3) == 5
7 |     assert add(-1, 1) == 0
8 |     assert add(-1, -1) == -2
```

运行测试：

```
1 | pytest test_math.py
```

7.3.7 寻求帮助

在 Python 编程中遇到错误或问题时，有许多途径可以寻求帮助。以下是一些常见且有效的途径：

A Google

Google 是查找编程问题答案的首要工具。通过在 Google 搜索框中输入错误信息或相关问题描述，通常可以找到解决方案或相关讨论。有如下使用要点：

- 使用具体的错误信息进行搜索。
- 添加“Python”关键字以确保相关性。
- 查看前几页的搜索结果，通常会找到有用的资源。

B Stack Overflow

Stack Overflow 是一个广受欢迎的问答网站，专注于编程问题。你可以在上面搜索已有的问题和答案，或者提问以获得社区的帮助。主要使用技巧：

- 在提问前，先搜索是否已有类似的问题。
- 提问时提供详细的错误信息、代码片段和环境描述。
- 标记问题以便相关领域的专家可以看到。

C GitHub Issues

如果你在使用某个特定的库或工具时遇到问题，可以查看该项目的 GitHub 仓库的 Issues 部分。许多项目在这里记录已知问题和解决方案。使用要点：

- 搜索已有的 **Issues**，看看是否已有相同或类似的问题。
- 若没有找到，可以新建一个 **Issue**，描述问题并附上相关信息。

D 官方文档和教程

Python 及其生态系统中的许多库都有详细的官方文档和教程。这些资源通常包含示例代码和常见问题的解决方案。

- 🐍 **Python** 官方文档
- 各个库的官方文档，如 🐍 **Pandas**，🐍 **Matplotlib**

E ChatGPT 和其他 AI 助手

🐍 **ChatGPT** 及其他 **AI** 助手可以快速提供建议和解决方案。你可以描述你的问题，**AI** 通常会提供有用的代码示例或解释。使用技巧：

- 提供详细的错误信息和上下文。
- 如果问题复杂，可以分步描述和询问。