

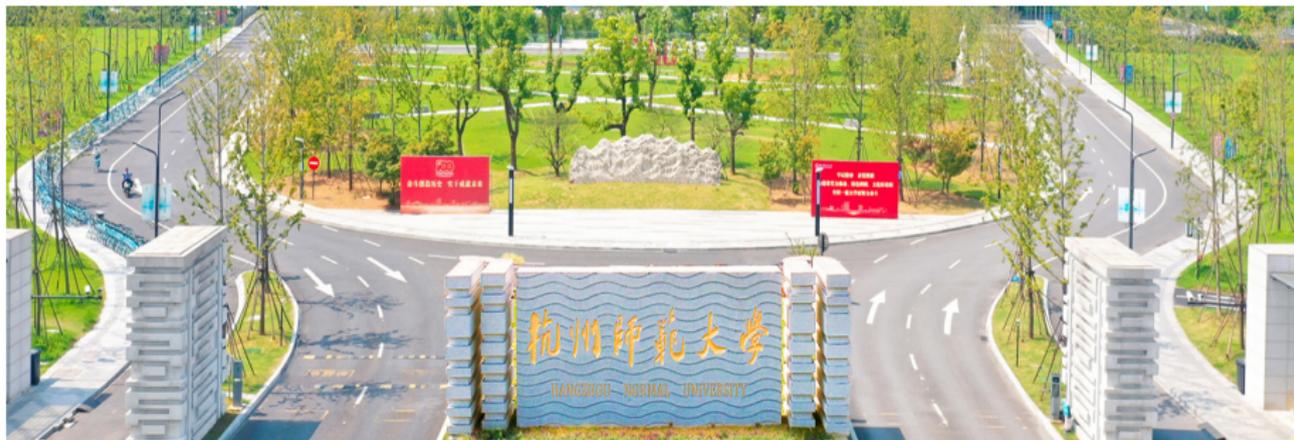
# Python 程序设计-2024

张建章

阿里巴巴商学院

杭州师范大学

2024-09



- 1 课程考核说明
- 2 关于课程
- 3 计算思维
- 4 计算机与计算机语言
- 5 Python 编程环境配置
- 6 Python 初学者指南
- 7 Python 编程实例
- 8 Python 交互式解释器
- 9 Jupyter Notebook
- 10 表达式
- 11 变量
- 12 常见的 Python 数据类型
- 13 函数
- 14 输入和输出
- 15 模块

- 16 流程控制
- 17 保存和执行程序
- 18 写程序注意事项
- 19 序列概述
- 20 创建列表
- 21 列表的基本操作
- 22 列表的常用方法
- 23 列表推导式
- 24 比较两个列表
- 25 多维列表
- 26 常用的操作列表的内置函数
- 27 常见的可迭代对象
- 28 创建元组
- 29 元组的基本操作
- 30 元组的常用方法

- 31 序列的通用操作
- 32 创建字符串
- 33 字符串的基本操作
- 34 字符串的常用方法
- 35 字符串格式化
- 36 string 模块
- 37 特殊字符
- 38 字符串的高级格式化设置
- 39 输出字面上的% 和 {} 占位符
- 40 正则表达式
- 41 创建集合
- 42 集合的基本操作
- 43 集合常用方法
- 44 frozenset 与 set 的区别
- 45 创建字典

- 46 字典的基本操作-增删改查
- 47 复制字典
- 48 其他常见操作
- 49 字典的字符串格式化使用
- 50 流程控制
- 51 条件语句
- 52 循环语句
- 53 嵌套
- 54 流程控制中常用的语句和函数
- 55 抽象的概念及意义
- 56 自定义函数的定义与调用
- 57 函数的参数
- 58 局部变量与全局变量
- 59 函数式编程
- 60 递归函数

- 61 文件打开与关闭
- 62 文件的读写方法和模式
- 63 读取模式
- 64 写入方法
- 65 写入模式
- 66 文件路径
- 67 文件随机读写
- 68 异常的概念
- 69 异常传播机制
- 70 Python 的内置异常类
- 71 异常处理语句
- 72 Python 代码组织的基本概念
- 73 常用的 Python 标准库模块
- 74 模块的定义与使用
- 75 第三方库

根据教学大纲要求，本课程的考核办法为：

$$\begin{aligned} \text{总成绩} = & \text{期末成绩} \times 50\% + \text{日常作业} \times 30\% \\ & + \text{日常考勤} \times 10\% + \text{课堂表现} \times 10\% \end{aligned}$$

其中，期末考试采用**上机考试**形式。

课程名称：《Python 程序设计》

课程目标：

- ① 掌握 Python 编程语言；
- ② 培养“计算思维”；
- ③ 通过程序设计高效解决实际问题。

授课方式：上机实验为主，主要基于 Jupyter-lab 交互式编程教学

作业提交：坚果云在线提交 Jupyter Notebook 文件 (后缀为.ipynb)

商科同学为什么要学 Python——赋能数字经济研究与实践：



## 什么是计算思维

**计算思维 (computational thinking):** 计算思维 (Computational Thinking, CT) 是一种通过运用计算机科学的基本概念来解决问题、设计系统和理解人类行为的过程。计算思维的核心包括以下四个方面:

- **分解 (Decomposition):** 将复杂问题分解为更小、更易于处理的部分, 从而使问题更加可理解和可管理;
- **模式识别 (Pattern Recognition):** 识别问题中的模式和规律, 找出相似性, 从而简化解决方案的开发过程;
- **抽象 (Abstraction):** 提取问题的关键信息, 忽略不必要的细节, 以简化问题的表示和解决过程;
- **算法思维 (Algorithmic Thinking):** 设计一个明确的、有步骤的解决方案 (算法), 使其可以被计算机执行, 或为解决问题提供一种系统化的方法。

## 计算思维实例

## 案例:规划一次度假

假设你正在计划一次家庭度假,这个过程可以应用计算思维的四个核心方面:

1. **分解 (Decomposition)**: 首先,将度假计划分解为更小的任务。例如,你需要选择目的地、确定预算、预订住宿和安排交通。每一个任务都是一个独立的子问题,可以分别解决。
2. **模式识别 (Pattern Recognition)**: 在分解任务后,下一步是识别模式。例如,你可能注意到,每次度假都涉及类似的步骤,如选择住宿和交通工具。这些模式帮助你更有效地进行计划,因为你可以利用以前的经验来简化当前的任务。
3. **抽象 (Abstraction)**: 在规划过程中,你需要忽略不相关的细节,专注于关键因素。例如,在选择目的地时,你可能不需要考虑所有可能的天气条件,而只需要关注主要的气候类型和旅游季节。这种简化使得问题更加可控。
4. **算法思维 (Algorithmic Thinking)**: 最后,根据分解、模式识别和抽象的结果,制定一个明确的步骤来完成度假计划。这包括决定什么时候出发、如何预订机票和酒店,以及每天的活动安排。这些步骤应当详细到足以让任何人根据这些步骤顺利完成计划。

## 总结

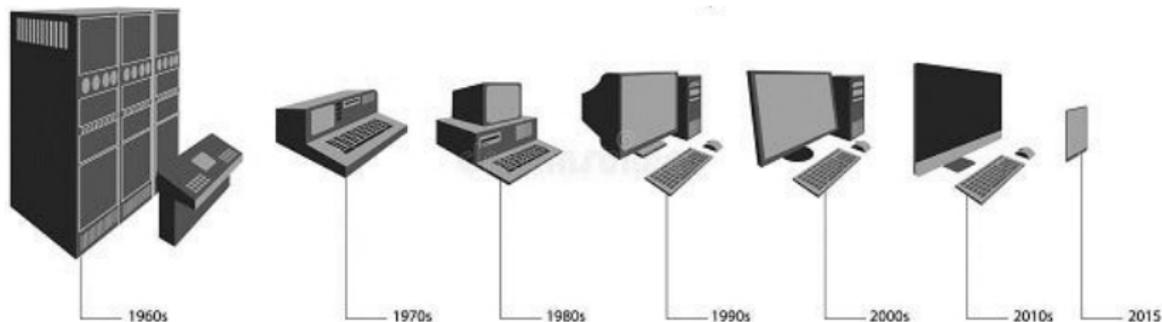
通过应用计算思维的四个核心方面,度假规划变得系统化和高效。首先,通过分解,将复杂问题拆分为可管理的部分;然后,通过模式识别,利用已知的解决方法加速当前任务;通过抽象,聚焦于重要信息而忽略无关细节;最后,通过算法思维,制定一个可执行的计划。

## 培养计算思维的重要性

- **提升问题解决能力：**在商业环境中，经常需要应对复杂的业务问题，如市场分析、财务建模和供应链管理等。通过培养计算思维，可以学会如何将复杂问题分解成更小的部分，找到问题中的模式，并设计出有效的解决方案；
- **数据分析和决策制定：**在大数据时代，商业决策越来越依赖于对大量数据的分析和解读。计算思维能够提升数据处理能力，使得在数据中识别模式、预测市场趋势或评估财务风险时更加高效和准确；
- **创新与自动化：**计算思维不仅帮助解决现有问题，还能激发创新。通过理解和应用算法设计，可以开发新的工具和方法来自动化重复性的业务流程，提高工作效率和创造力；
- **跨学科应用：**计算思维的核心方法，如分解、抽象和模式识别，能够在不同的学科和领域之间迁移和应用；

## 计算机

计算机是能够被编程以自动执行一系列算术或逻辑操作的机器，即，能够按照事先存储的程序（可编程性），自动、高速地对数据进行输入、处理、输出和存储的系统（功能性）。



- 早期计算设备：查尔斯·巴贝奇在 1820 年代设计的差分机；
- 电子计算机：1940 年代，电子数值积分计算机（ENIAC）问世；
- 个人计算机和互联网：80 年代初，个人计算机出现，21 世纪初，互联网普及；
- 现代计算机：向更高的速度、更大的存储容量和更高的集成度发展，如云计算和量子计算。

# 计算机语言

计算机语言是一种用于向计算机传达指令的符号系统。通过这些语言，人们可以编写计算机程序，这些程序告诉计算机如何执行特定的任务。

- **Python:** 一种通用的编程语言，广泛用于数据分析、机器学习、人工智能、web 开发等；
- **JavaScript:** 一种主要用于前端开发的语言，常用于构建动态和交互式网页；
- **Java:** 是一种面向对象的编程语言，广泛用于企业级应用、安卓移动应用和大型系统开发；
- **SQL:** 结构化查询语言，用于管理和操作关系数据库，是处理大规模数据集和进行数据分析的重要工具。

### 常用编程语言



图 1: 常用编程语言

### 计算机语言的特点

- **精确性和确定性：**计算机语言是高度精确的，用于向计算机传达明确的指令；
- **语法和语义：**计算机语言的语法和语义固定不变，专门用于消除歧义并确保计算机能准确执行任务；
- **表达能力有限：**主要用于执行明确的计算任务和操作，缺乏对情感和复杂人类思想的表达能力；
- **结构和抽象：**计算机语言通常使用固定的词汇和严格的语法规则来定义操作，让程序员能够更好地控制计算机硬件。

## 静态语言与脚本语言

静态语言通常使用编译器并进行类型检查以确保代码的安全性和性能。而脚本语言往往使用解释器，具有动态类型，可以更方便地进行快速开发和迭代。

**静态语言 (Static Languages):** 静态语言通常是指在编译时需要明确类型的编程语言。变量的类型在编写代码时就已经确定，并且在整个程序的生命周期中不会改变。这种类型的语言在编译阶段会进行类型检查，这有助于在代码运行之前捕获潜在的错误。例如，C、C++ 和 Java 都是典型的静态语言。

**脚本语言 (Scripting Languages):** 脚本语言是一种解释执行的编程语言，通常用于自动化任务、处理数据或在 web 开发中生成动态内容。与静态语言不同，脚本语言通常具有动态类型，这意味着变量的类型是在运行时确定的。脚本语言一般不需要预编译为机器码，而是通过解释器逐行执行。例如，Python、JavaScript 和 PHP 都是常见的脚本语言。

## 编译与解释

编译和解释是两种不同的程序执行方式。编译器在执行之前将整个源代码一次性转换为机器码，而解释器逐行执行代码。编译语言通常具有更好的性能，而解释语言则更具灵活性和开发速度。

**编译 (Compilation):** 编译是将高级编程语言代码转换为机器码的过程。编译器会在程序运行之前将整个源代码翻译为机器可执行的文件。

**解释 (Interpretation):** 解释是逐行读取和执行源代码的过程。解释器在代码运行时实时翻译和执行代码，这使得开发者能够快速测试和修改代码，因为不需要在每次修改后重新编译整个程序。



编译



解释

## 编程的基本原则

编写程序的主要原则是确保代码简洁、可读、易于维护，并能有效地执行任务。以下是一些核心编程原则，这些原则有助于开发人员编写高质量的代码，减少错误，提高代码的可读性和可维护性：

- KISS (Keep It Simple, Stupid!);
- DRY (Don't Repeat Yourself);
- 单一职责原则 (Single Responsibility Principle): 每个模块或函数应只负责一个特定功能，这样可以减少耦合，提高代码的可维护性；
- 文档化 (Document Your Code): 编写清晰的代码注释，帮助其他开发者理解代码的意图和功能，尤其在团队合作中尤为重要。

## Python 编程语言简介

Python 是一种高级、通用的编程语言，由 Guido van Rossum 于 1991 年首次发布。它以简洁、易读的语法和广泛的应用领域而闻名。

- 易于学习和使用；
- 广泛的应用领域：数据科学、人工智能、机器学习、Web 开发、自动化脚本、数据分析等；
- 丰富的库和社区支持：Python 拥有庞大的标准库和第三方库，这些库为图形处理、数据分析、机器学习、Web 开发等提供了强大的支持；
- 跨平台兼容性；
- 动态类型和自动内存管理：变量类型在运行时确定使 Python 更灵活，能更快地开发和测试代码。Python 还具有自动内存管理功能，开发者不需要手动管理内存分配，这减少了编程中的常见错误。

## 4. 计算机与计算机语言

Python 是一种非常适合商科学生学习的编程语言，通过学习 Python，商科学生不仅能够提升数据分析能力和自动化水平，还能获得在现代商业环境中所需的技术技能，为未来的职业发展打下坚实的基础。



图 2: 2024 年度 IEEE Spectrum 编程语言排行榜

## 本地编程环境

Anaconda 是一个面向科学计算、数据科学和机器学习的 Python 发行版。它不仅包括 Python，还集成了用于数据分析、机器学习和科学计算的超过 1500 个开源软件包。Anaconda 的主要工具包括 Conda、Jupyter Notebook、Spyder IDE 等。使用 Anaconda 便于更专注于学习 Python 编程和数据分析技能，而不必担心环境配置和依赖管理的问题。

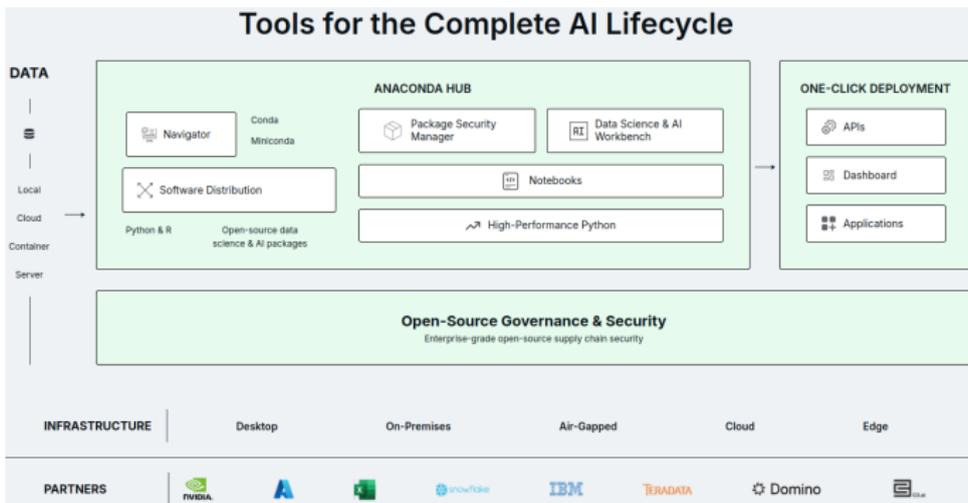


图 3: Anaconda 面向 AI 生命周期的工具集  
课件合集

## Windows 安装 Anaconda

### 判断 32 位和 64 位 Windows



在线视频

## Jupyter-lab 基本用法



[在线视频, 文字说明](#) (For Windows); [在线视频, 文字说明](#) (For Mac)

### 在线编程环境

在线 Python 编程环境（如魔搭社区、Kaggle、Google Colab 等）提供了多种独特的优势，尤其是对于初学者和需要灵活工作环境的用户。这些环境可以通过标准的网络浏览器直接访问，无需在本地安装 Python 或相关工具，极大地降低了设置的复杂性。一些平台还支持高级计算资源（如 GPU 和 TPU），非常适合需要大量计算能力的任务，如机器学习模型的训练。

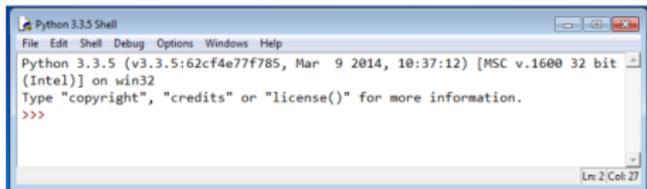


#### ModelScope 魔搭社区

共享 · 共创 · 共进，构建持续创新的 AI 开源生态

加入社区

### 让 Python 编程更高效的工具



原生 Python 和 IDE



集成开发环境



能跑的车架子



跑车

## 为了学 Python 我需要什么样的电脑



没有必要购买高端电脑，如外星人



市面上普通的电脑即可用于本课程学习

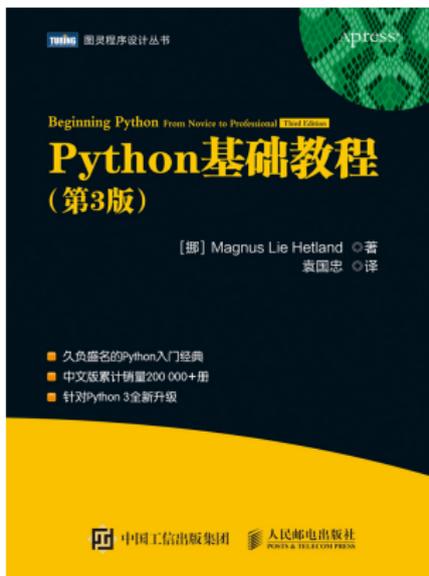
## 学习资源



《程序设计基础》课程网站

课程讨论区 

课堂和课后互动



参照课本勤奋练习



善于搜索互联网

## 善用生成式人工智能工具

生成式人工智能（Generative AI）工具，如 ChatGPT、Claude、Copilot、Cursor，能够极大地提升 Python 初学者的学习效率。这些工具可以自动生成代码片段，帮助初学者快速理解如何实现特定功能，并学习 Python 的语法和最佳编码实践。我国近年来也涌现出一批出色的生成式人工智能工具，如，阿里巴巴的通义千问、月之暗面的 Kimi、字节跳动的豆包、百度的文心一言等。



## 计算机编程的基本方法

- ① 输入 (Input): 终端命令行交互, 文件, 网络;
- ② 处理 (Processing): 对输入数据进行计算并产生输出结果的过程;
- ③ 输出 (Output): 通过终端命令行, 文件, 网络等输出结果。



## 计算实例: 体质指数

$$\text{体质指数 (Body Mass Index, BMI)} = \frac{\text{体重 (kg)}}{\text{身高 (m)}^2}。$$

BMI 值是一个中立而可靠的指标，是国际上常用的衡量人体胖瘦程度以及是否健康的一个标准。

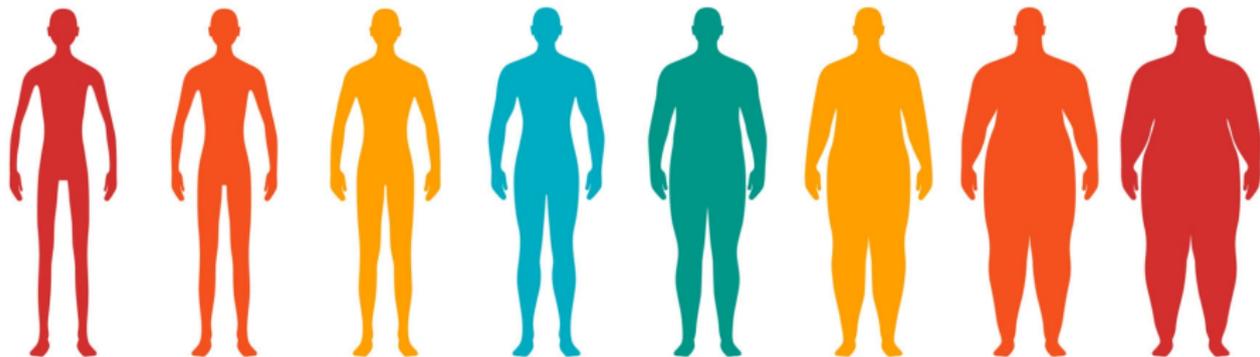
**计算机:**

- ① 接收用户输入的身高、体重数据;
- ② 根据公式计算;
- ③ 输出体质指数。

**用户:**

- ① 输入身高、体重数据;
- ② 查看计算结果，对照量表;

## BMI 对照表

BODY MASS INDEX (kg/m<sup>2</sup>)

&lt; 16

Severe  
Thinness

16 - 17

Moderate  
Thinness

17 - 18.5

Mild  
Thinness

18.5 - 25

Normal

25 - 30

Overweight

30 - 35

Obese  
Class I

35 - 40

Obese  
Class II

&gt; 40

Obese  
Class III

## 用 Python 计算 BMI

```
Height = float(input(" 请输入身高 (m): "))
Weight = float(input(" 请输入体重 (kg): "))

BMI = round(Weight/Height**2, 2)

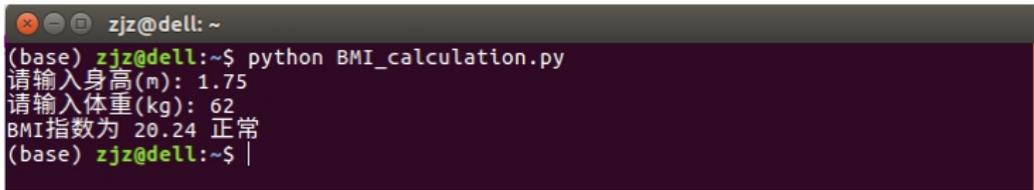
if BMI>=25:
    print("BMI 指数为", BMI, " 体质偏重")
elif BMI<=18.5:
    print("BMI 指数为", BMI, " 体质偏轻")
else:
    print("BMI 指数为", BMI, " 正常")
```

## 两种代码执行方式

## 交互式

```
[1]: # 计算BMI指数
[2]: Height = float(input("请输入身高(m): "))
      请输入身高(m): 1.75
[3]: Weight = float(input("请输入体重(kg): "))
      请输入体重(kg): 62
[4]: BMI = round(Weight/Height**2, 2)
[5]: if BMI>=23.9:
      print("BMI指数为", BMI, "体质偏重")
      elif BMI<=18.5:
      print("BMI指数为", BMI, "体质偏轻")
      else:
      print("BMI指数为", BMI, "正常")
      BMI指数为 20.24 正常
```

## 脚本式



```
zjz@dell: ~
(base) zjz@dell:~$ python BMI_calculation.py
请输入身高(m): 1.75
请输入体重(kg): 62
BMI指数为 20.24 正常
(base) zjz@dell:~$ |
```

# 推荐使用交互式解释器学习 Python 程序设计

安装 Python 3.X 并启动: 本课程安装的 Anaconda 3.X 已内含 Python 3.X, 启动 Jupyter-lab 即启动 Python;



## 初学者使用交互式解释器的一些常见问题

## (1) jupyter-lab 启动后无法成功运行

- ① 在使用 jupyter-lab 的过程中，确保 cmd 黑框框处于打开状态；
- ② 如果 cmd 黑框框显示 **Bad File Descriptor** 错误，类似下图

```
To access the notebook, open this file in a browser:
file:///C:/Users/%E5%B8%85%E5%B8%85%E9%A3%9E%E7%8C%AA/AppData/Roaming/j
Or copy and paste one of these URLs:
http://localhost:8888/?token=47cf2aaa44780278c4e644e8c277c5088e44a5cca0
or http://127.0.0.1:8888/?token=47cf2aaa44780278c4e644e8c277c5088e44a5cca0
[I 18:56:16.832 NotebookApp] 302 GET / (:::1) 0.000000ms
[W 18:56:44.320 NotebookApp] 404 GET /nbextensions/widgets/notebook/js/extension
00ms referer=http://localhost:8888/notebooks/TFPractise/Untitled.ipynb
Bad file descriptor (C:\projects\libzmq\src\epoll.cpp:100)
Bad file descriptor (C:\projects\libzmq\src\epoll.cpp:100)
```

解决办法：关闭当前 cmd 黑框框，重新打开 cmd 黑框框，在确保网络连通的情况下，依次执行如下两条命令 `pip uninstall pyzmq`；`pip install pyzmq==19.0.2 --user`，两条命令成功运行后（运行时没有出现 Error 信息），重新启动 jupyter-lab 即可。

### (2) jupyter-lab 的浏览器界面需要输入 token

请复制你 cmd 黑框框中的 http 开头的网址 (两个网址中的任意一个, 类似下图) 到浏览器打开。

```
To access the notebook, open this file in a browser:
  file:///C:/Users/Administrator/AppData/Roaming/jupyter/runtime/nbserver-58048-open.html
Or copy and paste one of these URLs:
  http://localhost:8888/?token=adda914e1a9f67dc96d78601ae42e1dae7cb10efa3bc06b3
  or http://127.0.0.1:8888/?token=adda914e1a9f67dc96d78601ae42e1dae7cb10efa3bc06b3
[W 23:49:14.572 LabApp] Could not determine jupyterlab build status without nodejs
[I 23:49:16.360 LabApp] Kernel started: bde3aa9e-ea4a-4fa0-a89c-e46129c63b26
```

### (3) 运行 BMI\_calculation 代码时, 输入身高体重后无法继续计算

确保在输入身高体重信息后, 按 Enter 键确认, 因为 input 函数在接受键盘输入后, 需要用户确认输入, 以继续运行后续程序代码。

### (4) 明明在 jupyter-lab 里写了代码, 却在本机上找不到

请在启动 jupyter-lab 后, 进入到桌面 Desktop, 然后新建 Notebook, 重命名为有意义的英文名字, 再写代码, 写代码过程中, 一定要多按保存键, 快捷键为 Ctrl + S。

## (5) 课程配套代码的打开与运行

**本地：**①将课程网站的代码下载到本机，建议下载到桌面，便于查找；②启动 `jupyter-lab`，从左侧文件导航栏找到本地保存的课程代码，双击后打开；③ 运行选中的当前代码行，即当前选中的 `cell`，点击形如播放的按钮，运行全部代码，点击形如快进的按钮。

**云端：**①将课程网站的代码下载到本机，建议下载到桌面，便于查找；② 打开魔搭在线 `jupyter` 环境，左侧文件导航栏上方，点击向上的箭头，上传课程代码到云端；③启动 `jupyter-lab`，从左侧文件导航栏找到本地保存的课程代码，双击后打开；④ 运行选中的当前代码行，即当前选中的 `cell`，点击形如播放的按钮，运行全部代码，点击形如快进的按钮。

**注意：**一定要自己练习课程提供的代码，切忌只看代码，而不动手写代码和运行自己写的代码。

## 表达式和语句

**表达式** (Expression) 是由操作符和操作数组成的代码单元, 用于计算和返回一个值。

```
3 + 5 # 是一个算术表达式, 使用了加法操作符 +
```

```
abs(-7) # abs(-7) 是一个函数调用表达式, 结果为 7
```

表达式可以包含常量、变量、函数调用和操作符等。根据使用的操作符类型, 表达式可以分为不同的类别。

**语句** (Statement) 是一条完整的指令, 通常执行操作或更改程序的状态, 语句不一定返回值。

```
x = 5 # 赋值语句
```

## 算数表达式

算术表达式涉及数字和算术操作符（如 +、-、\*、/等），用于执行数学运算。

表 1: Python 常用算术操作符

操作符	含义	优先级	示例	结果
( )	括号（最高优先级）	最高	<code>(3 + 2) * 4</code>	20
**	幂运算	高	<code>2 ** 3</code>	8
*	乘法	中	<code>3 * 4</code>	12
/	除法	中	<code>10 / 2</code>	5.0
//	整数除法	中	<code>10 // 3</code>	3
%	取余	中	<code>10 % 3</code>	1
+	加法	低	<code>3 + 5</code>	8
-	减法	低	<code>10 - 4</code>	6

## 算数表达式

# 涉及括号、指数运算、整数除法、取余和多级嵌套运算

```
result = (3 + 5 * 2) ** 2 // (4 % 3 + 2 * (5 - 3))  
print(result)
```

# 该表达式包含多个优先级相同的运算符，需要理解从左到右的关联性

```
result = 5 ** 3 // 7 % 4 + 6 * (7 // 2)  
print(result)
```

# 该表达式包含多个不同优先级的操作符，需要掌握优先级表并正确使用括号

```
result = (2 ** 3) * (3 // 2 + 4 % 3) ** 2 - 5 / 2 + 7  
print(result)
```

## 关系表达式

关系表达式 (Relational Expressions) 用于比较两个操作数, 并返回一个布尔值, 即 `True` 或 `False`。

表 2: Python 常用关系操作符

操作符	含义	示例	结果
<code>==</code>	等于	<code>5 == 5</code>	<code>True</code>
<code>!=</code>	不等于	<code>5 != 3</code>	<code>True</code>
<code>&gt;</code>	大于	<code>10 &gt; 5</code>	<code>True</code>
<code>&lt;</code>	小于	<code>3 &lt; 7</code>	<code>True</code>
<code>&gt;=</code>	大于或等于	<code>4 &gt;= 4</code>	<code>True</code>
<code>&lt;=</code>	小于或等于	<code>6 &lt;= 8</code>	<code>True</code>

## 逻辑表达式

逻辑操作符用于对布尔表达式进行运算，最终返回 `True` 或 `False`。

表 3: Python 中的逻辑操作符

操作符	含义	优先级
<code>and</code>	逻辑与：仅当所有操作数为 <code>True</code> 时返回 <code>True</code>	中等
<code>or</code>	逻辑或：只要有一个操作数为 <code>True</code> 就返回 <code>True</code>	低
<code>not</code>	逻辑非：将 <code>True</code> 变为 <code>False</code> ，反之亦然	高

表达式 `not (True or False and True)` 会按照以下顺序计算：

- ① 首先执行 `and`，因为它的优先级高于 `or`，结果是 `False`。
- ② 然后执行 `or`，其结果为 `True`。
- ③ 最后，`not` 将结果反转为 `False`。

## 逻辑表达式

# 逐步解析每个条件表达式，讨论`not`、`and`和`or`的结合使用如何影响最终结果

```
x = 8;y = 20;z = 15
```

```
result = not (x > 5 and y == 20) or (z < 10)
```

# 详细分析每一步的计算过程，并说明在多种逻辑操作符混合使用时，

↪ 如何决定最终结果

```
p = 3;q = 6;r = 9
```

```
result = (p + q > r) and (q != r or p <= q)
```

# 解释该表达式的执行顺序，特别是如何处理短路运算

↪ （即在逻辑表达式中尽量减少计算），推导并验证最终结果

```
x = 0;y = 4;z = 7
```

```
result = x > y or (y < z and z > x) or not y
```

## 身份运算符

Python 提供两个身份运算符：`is` 和 `is not`。它们用于判断两个对象是否是内存中的同一对象，而不仅仅是比较它们的值。

- `is` 运算符：当两个变量引用同一个对象时，`is` 返回 `True`。例如：

```
a = [1, 2, 3]
b = a
print(a is b) # 输出: True
```

这里，`a` 和 `b` 引用的是同一个列表对象，所以 `a is b` 为 `True`。

- `is not` 运算符：用于判断两个变量是否引用不同的对象：

```
c = [1, 2, 3]
print(a is not c) # 输出: True
```

即使 `a` 和 `c` 的内容相同，但它们在内存中是不同的对象。

## 成员关系运算符

成员关系运算符用于检查元素是否存在于某个序列中，如列表、元组或字符串。Python 提供了 `in` 和 `not in` 运算符。

- `in` 运算符：用于检查某个值是否是序列的成员。如果该值存在，返回 `True`：

```
fruits = ['apple', 'banana', 'cherry']  
print('banana' in fruits) # 输出: True
```

- `not in` 运算符：用于检查某个值是否不存在于序列中。如果该值不存在，返回 `True`：

```
print('mango' not in fruits) # 输出: True
```

## 操作符的优先级

在 Python 中，操作符的优先级决定了在表达式中各个操作符的计算顺序。高优先级的操作符会先计算，除非使用括号明确改变运算顺序。以下是 Python 中所有操作符的优先级表，从最高到最低优先级。

表 4: Python 操作符优先级 (从高到低)

优先级	操作符	描述
1	<code>()</code>	圆括号用于改变优先级
2	<code>**</code>	指数运算 (从右到左)
3	<code>+x</code> , <code>-x</code> , <code>~x</code>	一元加、减, 按位取反
4	<code>*</code> , <code>/</code> , <code>//</code> , <code>%</code>	乘法、除法、取整除、取余
5	<code>+</code> , <code>-</code>	加法、减法
6	<code>&lt;&lt;</code> , <code>&gt;&gt;</code>	位移运算符
7	<code>&amp;</code>	按位与
8	<code>^</code>	按位异或
9	<code> </code>	按位或
10	<code>==</code> , <code>!=</code> , <code>&gt;</code> , <code>&lt;</code> , <code>&gt;=</code> , <code>&lt;=</code> , <code>is</code> , <code>is not</code> , <code>in</code> , <code>not in</code>	比较运算符、身份运算符、成员运算符
11	<code>not</code>	逻辑非
12	<code>and</code>	逻辑与
13	<code>or</code>	逻辑或

**变量**是用于存储数据的容器。每个变量都指向存储在计算机内存中的特定值。变量可以存储不同类型的数据，包括数字、字符串、布尔值等。

```
x = 10          # 整数
name = "Alice" # 字符串
is_active = True # 布尔值
```

通过赋值操作符 `=`，可以将某个值赋给变量。比如：

```
age = 25
```

Python 允许在一行中同时为多个变量赋值：

```
a, b, c = 1, 2, 3
```

## 变量命名

变量的名称可以是字母、数字和下划线的组合，但不能以数字开头。此外，变量命名遵循一系列最佳实践和约定，以确保代码的可读性、可维护性以及团队协作时的易理解性。

- **使用描述性和有意义的名称：** 变量名称应清晰地描述变量的用途或内容。避免使用简短、不具描述性的名称（如 `x`、`y`），而应使用如 `student_age` 这样的名称，便于他人阅读和理解。
- **遵循命名约定：** Python 推荐使用 **snake\_case** 作为变量命名规范，即用小写字母并以下划线分隔单词，例如 `first_name`。
- **避免使用保留关键字：** 不要使用 Python 的保留字（如 `if`、`else`、`for`）作为变量名，以免导致语法错误。使用下面代码查看保留关键字。

```
import keyword
keyword.kwlist
```

## 变量命名

**问题:** 根据 Python 的变量命名规则, 找出每个变量名中的问题并进行纠正。提示: 变量名不能以数字开头、不能包含特殊字符 (如 # 和空格), 并且避免使用 Python 的保留关键字。

```
2ndPlaceWinner = "John"  
total#ofBooks = 150  
User Email = "example@domain.com"  
float = 7.5  
VARiable = 100
```

## 数字-整数

1. **二进制 (Binary)** 通过在数字前加前缀 `0b` 或 `0B` 表示二进制。例如，`0b1010` 表示二进制数 1010，其十进制值为 10。
2. **八进制 (Octal)** 使用前缀 `0o` 或 `0O` 来表示八进制数。例如，`0o123` 表示八进制数 123，对应的十进制值为 83。
3. **十进制 (Decimal)** 默认情况下，整数以十进制形式表示，无需添加任何前缀。例如，`123` 即表示十进制数 123。
4. **十六进制 (Hexadecimal)** 前缀 `0x` 或 `0X` 用于表示十六进制数。例如，`0x1A` 表示十六进制数 1A，对应的十进制值为 26。

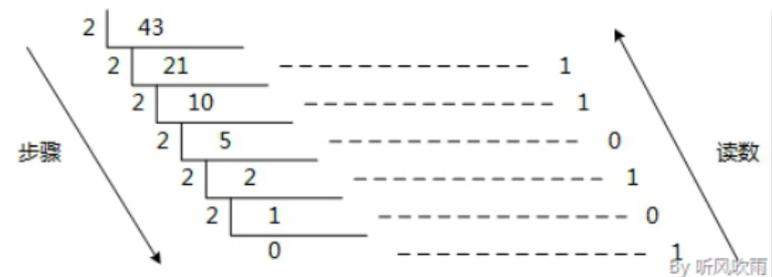


图 4: 除 n 取余法 (43→101011)

Python 还提供了内置函数来进行进制转换：

- `bin()`：将整数转换为二进制字符串表示，例如 `bin(10)` 结果为 `'0b1010'`。
- `oct()`：将整数转换为八进制字符串表示，例如 `oct(83)` 结果为 `'0o123'`。
- `hex()`：将整数转换为十六进制字符串表示，例如 `hex(26)` 结果为 `'0x1a'`。

此外，`int()` 函数可以将字符串形式的数值转换为指定进制的整数，例如：

```
int("0b1010", 2) # 结果为 10
int("0o123", 8)  # 结果为 83
int("0x1A", 16)  # 结果为 26
```

## 数字-浮点数

浮点数 (float) 用于表示带有小数部分的实数, 由于浮点数在计算机中是用二进制表示的, 某些十进制小数在二进制中不能精确表示。例如, `0.1` 在 Python 内部的表示并非精确的 `0.1`, 而是一个近似值, 这可能在数值比较时导致问题。

```
a = 3.14      # 直接赋值
b = float(5)  # 将整数转换为浮点数

1-0.9 == 0.1 # False

import math
math.isclose(0.1,1-0.9) # True
```

科学计数法的形式为  $N \times 10^M$ , 在 Python 中表示为 `NeM` 或 `NEM`, 例如 `1.23e4` 表示的数值是  $1.23 \times 10^4$  (即 12300)。当数值大于 `1e16` 或小于 `1e-4` 时, Python 会自动将浮点数以科学计数法显示。

## 数字-复数

复数表示为  $a + bj$  的形式，其中  $a$  是实部， $b$  是虚部， $j$  表示虚数单位（即  $\sqrt{-1}$ ）。Python 中用  $j$  代替了传统的  $i$  来表示虚部。可以通过直接赋值和 `complex` 函数两种方式在 Python 中定义复数，使用 `.real` 和 `.imag` 属性，分别访问复数的实部和虚部。虚部为 1 时，不可以省略不写。

```
# 直接赋值
```

```
z = 3 + 4j
```

```
# 使用 complex() 函数，该函数接受两个参数，分别表示实部和虚部
```

```
z = complex(3, 4)
```

```
z = 3 + 4j
```

```
print(z.real) # 输出: 3.0
```

```
print(z.imag) # 输出: 4.0
```

## 布尔值

布尔值 (Boolean) 是一种表示逻辑真值的数据类型, 只有两个可能的取值: `True` 和 `False`。

布尔值可以通过比较运算符 (如 `<`, `>`, `==` 等) 来产生。例如, 表达式 `5 > 3` 会返回 `True`, 而 `2 == 3` 则返回 `False`。

除了直接的布尔值, Python 中的任何数据类型都可以被转换为布尔值, 非零数字、非空对象会被视为 `True`, 而 `0`、`None` 和空对象 (如空字符串、空列表) 会被视为 `False`。

```
bool(1)      # 返回 True
bool(0)      # 返回 False
bool("")     # 返回 False
bool("abc")  # 返回 True
```

布尔值参加算数运算时, `True` 和 `False` 分别被视作数字 1 和 0。

## 其他重要数据类型

- ① 字符串: 用于存储文本数据, 例如, `name = "Alice"`
- ② 列表: 有序的、可变的元素集合, 例如,  
`fruits = ["apple", "banana", "cherry"]`
- ③ 元组: 有序的、不可变的元素集合, 例如, `coordinates = (10, 20)`
- ④ range: 用于生成一系列数字的范围对象, 例如 `range(0, 10)`
- ⑤ 字典: 用于存储键值对, 例如,  
`person = {"name": "Alice", "age": 25}`
- ⑥ 集合: 无序且唯一的元素集合, 例如,  
`unique_numbers = {1, 2, 3}`
- ⑦ 不可变的集合: 不可变的集合, 元素不可更改的集合。
- ⑧ None: 表示“空”或“无值”的特殊对象。

## 类型转换

类型转换是通过调用内置函数手动将一种数据类型转换为另一种常用的数据类型，例如 `int()`、`float()`、`str()` 等。类型转换通常用于需要确保数据的类型正确时，特别是在处理用户输入或进行不同类型的数据运算时。例如，将字符串转换为整数：

```
num_str = "123"  
num_int = int(num_str) # 将字符串 '123' 转换为整数 123
```

通过类型转换，可以明确指定期望的数据类型，以防止由于类型不匹配导致的错误。常见的类型转换函数包括：

- `int()`：将其他类型转换为整数；
- `float()`：将其他类型转换为浮点数；
- `str()`：将其他类型转换为字符串。

**函数**是一个用于组织和重用代码的核心工具。函数将相关的代码逻辑封装在一个命名的代码块中，使程序更加模块化和易于维护。

函数通过 `def` 关键字定义，后面跟随函数名称和圆括号。圆括号中可以包含**参数**，这些参数是函数在调用时需要传入的值。以下是一个简单的函数示例：

```
# 定义函数
def add(a, b):
    return a + b

# 调用函数
result = add(5, 3) # result 的值为 8
```

## 常用内置函数

Python 中的内置函数提供了许多常用的功能，帮助简化编程任务。

表 5: 常用 Python 内置函数

函数	描述	示例
<code>print()</code>	将对象打印到控制台输出	<code>print("Hello, World!")</code>
<code>input()</code>	从用户输入读取一行字符串	<code>name = input("Enter your name: ")</code>
<code>abs()</code>	返回数字的绝对值	<code>abs(-7)</code> 返回 7
<code>pow()</code>	计算第一个参数的幂次,支持模运算	<code>pow(2, 3)</code> 返回 8
<code>len()</code>	返回对象的长度(如字符串、列表等)	<code>len("Python")</code> 返回 6
<code>sum()</code>	返回可迭代对象中所有元素的和	<code>sum([1, 2, 3])</code> 返回 6
<code>min()</code>	返回可迭代对象中的最小值	<code>min([5, 3, 9])</code> 返回 3
<code>max()</code>	返回可迭代对象中的最大值	<code>max([5, 3, 9])</code> 返回 9
<code>round()</code>	将浮点数四舍五入到指定的小数位数	<code>round(3.456, 2)</code> 返回 3.46
<code>type()</code>	返回对象的数据类型	<code>type(42)</code> 返回 <code>&lt;class 'int'&gt;</code>

输入与输出是程序与用户交互的基础功能。Python 提供了内置的 `input()` 和 `print()` 函数，分别用于接收用户输入和向屏幕输出结果。

`input()` 函数用于从用户处接收输入。调用该函数时，程序会暂停运行并等待用户输入，直到用户按下回车键。输入的内容会作为字符串返回。如果需要将用户输入转换为其他类型（如整数或浮点数），可以使用类型转换函数，例如：

```
age = int(input("Enter your age: ")) # 将输入转换为整数
```

`print()` 函数用于向控制台输出结果。它可以接收多个参数，并将它们以空格分隔输出。`print()` 还可以使用可选的 `sep` 和 `end` 参数，来控制输出的分隔符和结束符。例如：

```
print("Hello", "World", sep=", ", end="!") # 输出: Hello, World!
```

模块是 Python 中一种将相关功能组织在一起的方式。通过导入模块，可以使用模块中定义的函数、常量和对象。例如，Python 自带的 `math` 模块就是一个常用的数学模块，它提供了基本的数学运算函数和常量。

要使用 `math` 模块，首先需要通过 `import` 语句将其导入。导入后，可以通过 `math.` 前缀访问模块中的函数和常量。

```
import math

result = math.sqrt(16) # 使用 math 模块计算平方根
print(result) # 输出: 4.0
```

**流程控制 (Control Flow)** 是指程序代码的执行顺序。通常情况下，Python 程序会从上到下、逐行执行代码。但通过使用流程控制语句，程序可以根据特定条件、循环结构等来控制代码的执行顺序。Python 中的主要流程控制结构包括：顺序执行、选择（条件语句）和重复（循环语句）。

**条件语句**是 Python 编程中控制流程的基础，它允许程序根据特定条件执行不同的代码块。常见的条件语句包括 `if`、`elif` 和 `else`。

## 条件语句

`if` 语句用于检查条件是否为真，如果条件成立，则执行相应的代码块：

```
if condition:  
# 条件为真的情况下执行的代码
```

当需要处理多个条件时，可以使用 `elif` 来扩展条件判断。  
`else` 则用于在所有条件都不成立时执行默认代码块：

```
if condition1:  
# 条件 1 为真时执行  
elif condition2:  
# 条件 2 为真时执行  
else:  
# 所有条件为假时执行
```

假设我们在一个电子商务平台上，根据销售金额对产品进行分类。可以使用条件语句判断产品属于“低销量”、“中等销量”还是“高销量”：

```
sales = 15000 # 假设这是产品的销售金额

if sales > 20000:
    print(" 高销量产品")
elif 10000 <= sales <= 20000:
    print(" 中等销量产品")
else:
    print(" 低销量产品")
```

## 循环语句

**for** 循环和 **while** 循环是 Python 中常用的控制流程结构，广泛用于商业数据分析中的多种任务，如遍历数据集或动态处理数据。

**for** 循环用于遍历一个序列（如列表、元组或字符串），每次迭代提取一个元素。常见的使用场景包括逐行读取数据、遍历数据集中每一条记录等。

**while** 循环根据条件执行代码块，直到条件不再满足为止。它通常用于需要不确定的迭代次数时，例如数据监控或数据收集过程。

例如，假设有一个代表公司季度销售额的列表，可以用 `for` 循环来计算总销售额：

```
sales = [12000, 18000, 25000, 30000]
total_sales = 0

for sale in sales:
    total_sales += sale

print(f" 总销售额为: {total_sales}")
```

例如，假设我们想持续监控某一产品的库存量，当库存低于某个阈值时自动停止销售：

```
stock = 50 # 初始库存
threshold = 10 # 库存阈值

while stock > threshold:
    print(f" 当前库存: {stock}, 继续销售...")
    stock -= 5 # 每次销售 5 个
print(" 库存低于阈值, 停止销售")
```

- 命令行运行 **Python** 脚本:

```
python BMI_calculation.py
```

- 使用 **IDE** 运行 **Python** 脚本
- 使用 **Jupyter Notebook** 编写和运行代码

## 唯手熟尔

- 避免拼错标识符，如变量名，函数，语句等；
- 避免使用中文符号，如引号，逗号，括号等；
- 引号、括号通常成对使用，如，有左括号也要有右括号，左边有引号，右边引号也别漏；
- 注意书写格式 (冒号，缩进，对齐)。

## 19. 序列概述

在 Python 中，序列（**sequence**）数据类型是一类用于存储有序数据的容器，能够通过整数索引访问其元素。常见的序列类型包括字符串（**string**）、列表（**list**）、元组（**tuple**）。这些序列类型有一些共同特征：它们的元素是有序的，可以通过索引进行访问。

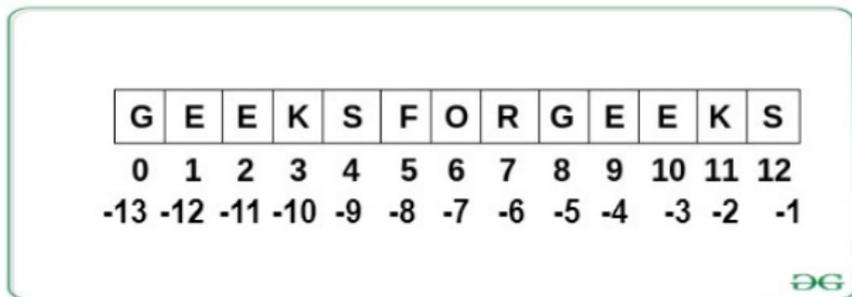


图 5: 序列的正向索引和反向索引

Python 的序列类型提供了丰富的操作，如切片（提取子序列）、连接（使用 **+** 运算符连接多个序列）、重复（使用 **\*** 运算符重复序列）和成员资格测试（使用 **in** 和 **not in** 测试元素是否在序列中）。

## 定义列表

**列表 (list)** 是一种**有序且可变**的数据结构，能够存储多个元素，并允许对这些元素进行动态操作，如添加、删除或修改。

Python 列表使用**方括号**定义，并通过逗号分隔元素。它可以包含各种类型的数据，例如数字、字符串、甚至其他列表。这使得列表能够存储复杂的数据结构，如订单记录、财务数据或客户反馈等。

```
# 示例：存储销售订单数据
orders = [" 订单 A", " 订单 B", " 订单 C"]
orders.append(" 订单 D") # 添加新订单
print(orders) # 输出 ['订单 A', '订单 B', '订单 C', '订单 D']
```

## list 函数

`list()` 是 Python 中的内置函数，用于将可迭代对象（如字符串、元组、集合等）转换为列表。该函数可以创建一个新的空列表，或者通过传递一个可迭代对象来初始化列表。

```
# 创建空列表
empty_list = list()
print(empty_list) # 输出: []
# 从字符串创建列表
string = "hello"
char_list = list(string)
print(char_list) # 输出: ['h', 'e', 'l', 'l', 'o']
# 从元组创建列表
tuple_data = (1, 2, 3)
list_from_tuple = list(tuple_data)
print(list_from_tuple) # 输出: [1, 2, 3]
```

## 列表的多维结构

Python 列表还支持多维结构，即列表的元素可以是另一个列表，使得它在表示复杂的商业数据时非常有用。例如，在一个订单系统中，每个订单可能包含多个产品，每个产品又有自己的属性（如名称、价格、数量）。使用嵌套列表可以很好地表示这种结构：

```
# 示例：存储订单中包含的产品信息
order_details = [
    ["产品 A", 100, 2], # 产品名称、单价、数量
    ["产品 B", 200, 1],
    ["产品 C", 150, 5]
]
print(order_details[0]) # 输出 ['产品 A', 100, 2]
```

在这个示例中，每个子列表代表一个产品的详细信息，而整个列表表示一个订单的产品清单。这种嵌套结构非常适合用于管理诸如采购订单、库存列表等复杂的数据。

## 索引操作

### 1. 使用正索引访问元素

Python 列表中的元素可以通过方括号 `[]` 内的索引值进行访问。

例如，有一个包含水果的列表：

```
fruits = ['apple', 'banana', 'mango', 'orange']
# 访问第二个元素
print(fruits[1]) # 输出: 'banana'
# 访问最后一个元素
print(fruits[-1]) # 输出: 'orange'
```

`fruits[1]` 访问的是列表中的第二个元素（索引从 0 开始）。

### 2. 使用负索引访问元素

负索引用于从列表的末尾开始计数，`-1` 表示最后一个元素，`-2` 表示倒数第二个元素，以此类推。这种方式在不确定列表长度时特别有用，方便访问列表末尾的元素。

### 3. 修改列表中的元素

列表是可变的数据结构，因此可以通过索引直接修改其中的元素。例如，修改上例中第二个元素为 `'strawberry'`：

```
fruits[1] = 'strawberry'  
print(fruits) # 输出: ['apple', 'strawberry', 'mango', 'orange']
```

同样地，也可以使用负索引来修改末尾的元素。

## 切片操作

在 Python 中，列表切片是从一个列表中提取部分元素的常用操作。其基本语法是：

```
list[start:stop:step]
```

其中，`start` 表示切片的起始索引（包含该索引），`stop` 表示结束索引（不包含该索引），而 `step` 表示每次跳过的步长。

### 1. 基本切片操作

最常见的切片是使用 `start` 和 `stop` 两个参数，从指定的起始位置到结束位置提取子列表。

```
fibonacci_sequence = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
sliced_list = fibonacci_sequence[2:5]
print(sliced_list) # 输出: [1, 2, 3]
```

这里，从索引 2 开始提取，到索引 5 结束（不包括索引 5）。

## 2. 省略 start 或 stop 参数

如果省略 start 参数，默认从列表的第一个元素开始；如果省略 stop，则切片会一直到列表的末尾。

```
sliced_list = fibonacci_sequence[:4]
print(sliced_list) # 输出: [0, 1, 1, 2]
```

此时提取的是从列表开头到索引 4 之前的所有元素。

## 3. 使用 step 参数

step 参数允许我们指定切片时的步长，从而可以跳过一些元素。例如，以下代码每隔一个元素提取一次：

```
sliced_list = fibonacci_sequence[1:8:2]
print(sliced_list) # 输出: [1, 2, 5, 13]
```

step 为 2，因此在指定范围内每隔一个元素提取一次。

#### 4. 负索引和反转

Python 允许使用负索引来从列表末尾进行切片。此外，可以通过负 `step` 来反转列表：

```
reversed_list = fibonacci_sequence[::-1]
print(reversed_list) # 输出: [34, 21, 13, 8, 5, 3, 2, 1, 1, 0]
```

这种方式可以轻松实现列表的反转。

#### 5. 使用切片插入元素

切片可以用来在列表中插入元素，而不替换现有元素。通过设置起始索引和结束索引相同的方式，可以在指定位置插入新元素。

```
numbers = [1, 2, 3, 6, 7]
numbers[3:3] = [4, 5] # 在索引 3 处插入元素
print(numbers) # 输出: [1, 2, 3, 4, 5, 6, 7]
```

通过 `[3:3]` 在索引 3 的位置插入了元素 `[4, 5]`。

## 6. 使用切片替换元素

切片也可以用来替换列表中的一部分元素，只需将指定范围内的元素替换为新的值。

```
colors = ['red', 'orange', 'yellow', 'green', 'blue']  
colors[1:3] = ['purple', 'pink'] # 替换索引 1 到 2 的元素  
print(colors) # 输出: ['red', 'purple', 'pink', 'green', 'blue']
```

在此操作中，列表中索引 1 和 2 的元素（'orange' 和 'yellow'）被新值 'purple' 和 'pink' 替换。

## 7. 使用切片删除元素

将某一范围内的元素替换为空列表，可以删除列表中的一部分元素。

```
colors = ['red', 'orange', 'yellow', 'green', 'blue']  
colors[1:3] = [] # 删除索引 1 到 2 的元素  
print(colors) # 输出: ['red', 'green', 'blue']
```

## 列表拼接

在 Python 中，使用加法运算符 `+` 将两个列表拼接在一起是一种简单、直接的方式。通过这个操作，两个列表会合并为一个新的列表，而原始列表不会被修改。

```
list1 = ['a', 'b', 'c']
list2 = ['d', 'e', 'f']

combined_list = list1 + list2
print(combined_list) # 输出: ['a', 'b', 'c', 'd', 'e', 'f']
```

**注意：**使用 `+` 运算符时，会生成一个新的列表对象。因此，对于非常大的列表，可能会消耗额外的内存。对于需要频繁拼接的大型数据集，可以考虑使用其他方法，如 `extend()` 方法，它直接修改现有列表，避免创建新的列表。

## 列表乘法

列表乘法是一种通过重复列表中的元素来生成新列表的操作。它使用星号运算符（`*`）来实现，基本语法如下：

```
my_list = [1, 2, 3]
new_list = my_list * 3
print(new_list) # 输出: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

当列表包含可变对象（如嵌套列表）时，乘法操作会重复引用而不是创建独立的副本。这意味着修改其中一个元素会影响所有引用的对象。

```
grid = [[0] * 3] * 4
print(grid) # 输出: [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
# 修改第一个子列表
grid[0][0] = 1
print(grid) # 所有子列表的第一个元素都被修改了
# 输出: [[1, 0, 0], [1, 0, 0], [1, 0, 0], [1, 0, 0]]
```

## 修改列表元素

1. **通过索引直接赋值**：最常见的修改方法是使用列表的索引来替换特定位置的元素。

```
my_list = [1, 2, 3, 4, 5]
my_list[1] = 10
print(my_list) # 输出: [1, 10, 3, 4, 5]
```

通过 `my_list[1]` 访问列表中的第二个元素，并将其修改为 10。这种方法简单直接，适用于已知索引的位置。

2. **使用切片修改多个元素**：

```
my_list = [1, 2, 3, 4, 5]
my_list[3:] = [6, 7]
print(my_list) # 输出: [1, 2, 3, 6, 7]
```

通过切片，可以直接修改指定范围内的多个元素。

## 删除列表元素

删除列表中的元素可以通过多种方法实现，主要包括以下几种常用方式：

### 1. 使用 `remove()` 方法

`remove()` 方法根据元素的值来删除列表中的第一个匹配项。如果列表中有重复的元素，`remove()` 只会删除第一个出现的值。如果指定的值不在列表中，则会抛出 `ValueError`。

```
# 创建一个包含多个元素的列表
fruits = ['apple', 'banana', 'cherry', 'banana']

# 删除第一个 'banana'
fruits.remove('banana')

# 输出更新后的列表
print(fruits) # 输出: ['apple', 'cherry', 'banana']
```

## 2. 使用 `pop()` 方法

`pop()` 方法通过索引删除元素。默认情况下，它删除并返回列表的最后一个元素。如果提供了索引参数，则删除并返回对应位置的元素。

```
# 创建一个列表
numbers = [10, 20, 30, 40]

# 删除并返回索引为 2 的元素
removed_item = numbers.pop(2)

# 输出被删除的元素和更新后的列表
print(removed_item) # 输出: 30
print(numbers)     # 输出: [10, 20, 40]
```

### 3. 使用 `del` 语句

`del` 语句可以通过索引删除列表中的元素，也可以删除整个列表的一部分或全部。

```
# 创建一个列表
languages = ['Python', 'Java', 'C++', 'Ruby']

# 删除索引为 1 的元素
del languages[1]

# 输出更新后的列表
print(languages) # 输出: ['Python', 'C++', 'Ruby']
```

#### 4. 使用切片删除多个元素

如果需要删除列表中多个连续的元素，可以使用切片赋值空列表方式或者结合 `del` 语句与切片操作。

```
# 创建一个列表
nums = [1, 2, 3, 4, 5, 6]

# 删除索引 2 到 4 的元素
nums[2:5] = [] # 等价于 del nums[2:5]

# 输出更新后的列表
print(nums) # 输出: [1, 2, 6]
```

## 增加列表元素

向列表添加元素的常用方法有多种，分别适用于不同的场景。以下为几种基本语法的介绍：

### 1. `append()` 方法

`append()` 用于向列表末尾添加一个元素。该元素可以是任意数据类型，如字符串、整数、列表等。示例代码如下：

```
players = ["player1", "player2", "player3"]
players.append("player4")
print(players)
# 输出: ['player1', 'player2', 'player3', 'player4']
```

该方法会直接修改原列表，但不返回新的列表。

## 2. `extend()` 方法

`extend()` 用于将另一个列表中的每个元素依次添加到当前列表中，而不是作为单个元素附加。

```
nums = [1, 2, 3]
nums.extend([4, 5, 6])
print(nums)
# 输出: [1, 2, 3, 4, 5, 6]
```

此方法适合在需要一次性添加多个元素时使用。

### 3. insert() 方法

`insert()` 允许在列表的指定索引位置插入元素。它接受两个参数：第一个是要插入的位置索引，第二个是要插入的元素。

```
nums = [1, 3, 4]
nums.insert(1, 2)
print(nums)
# 输出: [1, 2, 3, 4]
```

此方法可以用于精确控制元素插入的位置。

### 4. + 运算符

`+` 运算符可以将两个列表合并为一个新列表，不会修改原始列表。

```
nums1 = [1, 2, 3]
nums2 = [4, 5, 6]
combined = nums1 + nums2
print(combined) # 输出: [1, 2, 3, 4, 5, 6]
```

## 列表排序

`list.sort()` 方法用于对列表进行原地排序，这意味着它会直接修改原列表，而不会返回新的列表。该方法的基本语法为：

```
list.sort(key=None, reverse=False)
```

其中，`key` 和 `reverse` 是两个可选参数：

**key:** 用于指定一个函数，该函数会为列表中的每个元素生成一个用于比较的值。默认情况下，元素会被直接比较。

**reverse:** 用于指定排序顺序。默认值为 `False`，即升序排序。如果设置为 `True`，列表将按降序排序。

### 1. 升序排序（默认）

```
numbers = [4, 2, 9, 1]
numbers.sort()
print(numbers) # 输出: [1, 2, 4, 9]
```

### 2. 降序排序

```
numbers = [4, 2, 9, 1]
numbers.sort(reverse=True)
print(numbers) # 输出: [9, 4, 2, 1]
```

### 3. 使用 `key` 参数进行自定义排序

通过 `key` 参数可以实现根据元素的特定属性进行排序，例如根据字符串的长度排序：

```
words = ["apple", "banana", "cherry", "date"]
words.sort(key=len)
print(words) # 输出: ['date', 'apple', 'cherry', 'banana']
```

在此示例中，`len` 函数作为 `key` 的值，列表按照字符串的长度升序排序。

## 列表复制

复制列表可以通过多种方法实现。最常见的方式之一是使用 `copy()` 方法，即，`new_list = original_list.copy()`。

```
# 原始列表
original_list = [1, 2, 3]

# 使用 copy() 方法复制列表
new_list = original_list.copy()

# 修改新列表
new_list.append(4)

# 输出结果
print(" 原列表:", original_list)    # 输出: 原列表: [1, 2, 3]
print(" 新列表:", new_list)         # 输出: 新列表: [1, 2, 3, 4]
```

在此示例中，`copy()` 方法返回一个新的列表对象，但修改新列表不会影响原列表。这对于需要保留原始数据时非常有用。

除了 `copy()` 方法，Python 还支持通过切片 `[:]` 或使用 `list()` 构造函数来复制列表：

```
# 使用切片复制列表
new_list = original_list[:]

# 使用 list() 构造函数复制列表
new_list = list(original_list)
```

## 深复制和浅复制

在 Python 中，浅复制与深复制主要区别在于复制过程中处理对象嵌套结构的方式。

浅复制会创建一个新的对象，但不会递归复制其中的嵌套对象。相反，新的对象中的嵌套元素依然引用原来的对象。因此，当嵌套对象发生变化时，浅复制的副本与原对象都会受到影响。例如，假设有一个嵌套列表：

```
list1 = [[1, 2, 3], [4, 5, 6]]
list2 = list1.copy()
list3 = list1[:]
list2[0][0] = 100
print(list1) # 输出: [[100, 2, 3], [4, 5, 6]]
print(list2) # 输出: [[100, 2, 3], [4, 5, 6]]
print(list3) # 输出: [[100, 2, 3], [4, 5, 6]]
```

**深复制**则递归地复制所有的嵌套对象，从而确保副本与原对象完全独立，任何修改只会影响复制出的新对象，不会影响原始对象。例如：

```
import copy
list1 = [[1, 2, 3], [4, 5, 6]]
list2 = copy.deepcopy(list1)
list2[0][0] = 100
print(list1) # 输出: [[1, 2, 3], [4, 5, 6]]
print(list2) # 输出: [[100, 2, 3], [4, 5, 6]]
```

在这个例子中，`list2` 与 `list1` 完全独立，修改 `list2` 中的嵌套对象不会影响 `list1`。

### 区别总结

- ① 浅复制只复制了最外层的对象，嵌套对象仍然与原始对象共享引用，因此修改嵌套对象会影响原始对象，浅复制可以使用 `copy()` 方法和列表的切片操作实现。
- ② 深复制递归地复制所有对象，副本与原对象完全独立，修改副本不会影响到原始对象，深复制可以通过 `copy.deepcopy()` 实现。

## 元素成员判断

检查列表中是否包含某个元素可以使用关键字 `in`，如果元素在列表中，则返回 `True`；如果不在，则返回 `False`。此外，也可以使用 `not in` 来检查元素不在列表中的情况，返回 `True` 表示元素不在列表中。

```
# 定义一个列表
my_list = [1, 2, 3, 4, 5]

# 检查数字 3 是否在列表中
if 3 in my_list:
    print("3 is in the list")

# 检查数字 6 是否不在列表中
if 6 not in my_list:
    print("6 is not in the list")
```

在数据分析中，Python 列表的常用方法扮演着关键角色，它们不仅简化了数据操作，还提供了高效的解决方案。

表 6: Python 列表的常用方法

方法	描述	代码示例
<code>append(x)</code>	在列表末尾添加元素 <code>x</code> 。	<code>my_list.append(5)</code>
<code>extend(iter)</code>	将可迭代对象中的元素添加到列表末尾。	<code>my_list.extend([6, 7, 8])</code>
<code>insert(i, x)</code>	在索引 <code>i</code> 处插入元素 <code>x</code> 。	<code>my_list.insert(2, 'a')</code>
<code>remove(x)</code>	删除列表中第一个值为 <code>x</code> 的元素。	<code>my_list.remove(3)</code>
<code>pop([i])</code>	移除并返回索引 <code>i</code> 处的元素, 默认为最后一个。	<code>my_list.pop()</code>
<code>clear()</code>	移除列表中的所有元素。	<code>my_list.clear()</code>
<code>index(x)</code>	返回列表中第一个值为 <code>x</code> 的元素的索引。	<code>my_list.index(4)</code>
<code>count(x)</code>	返回列表中值为 <code>x</code> 的元素个数。	<code>my_list.count(2)</code>
<code>sort()</code>	对列表就地排序, 默认为升序。	<code>my_list.sort(reverse=True)</code>
<code>reverse()</code>	将列表中的元素反转。	<code>my_list.reverse()</code>

列表推导式（List Comprehension）是 Python 中一种简洁的语法，用于通过对已有的可迭代对象进行操作创建新的列表。相比传统的 `for` 循环，列表推导式不仅能够使代码更紧凑，而且在许多情况下具有更高的执行效率。

列表推导式的基本形式为：

```
[表达式 for 元素 in 可迭代对象 if 条件]
```

其中：

- `表达式` 是对每个元素进行的操作，生成新的列表元素；
- `元素` 是从可迭代对象中获取的每一个值；
- `可迭代对象` 可以是列表、字符串、范围（`range()`）等；
- `if 条件` 是可选项，用于过滤元素，只有满足条件的元素才会被包含在新列表中。

## 1. 简单示例：创建一个平方数列表

通过列表推导式，可以很容易地创建一个包含平方数的列表：

```
numbers = [1, 2, 3, 4, 5]
squares = [num ** 2 for num in numbers]
print(squares) # 输出: [1, 4, 9, 16, 25]
```

该示例中，`num ** 2` 是表达式，表示对每个 `numbers` 列表中的元素进行平方操作。

## 2. 带条件的列表推导式：筛选列表中的偶数

列表推导式也可以结合条件筛选元素。例如，生成一个仅包含偶数的列表：

```
even_numbers = [num for num in range(10) if num % 2 == 0]
print(even_numbers) # 输出: [0, 2, 4, 6, 8]
```

这里的 `if num % 2 == 0` 用于筛选偶数。

### 3. 多重条件和 `if...else` 的使用

使用 `if...else` 可以在不同条件下生成不同的结果。例如：

```
results = ["Even" if num % 2 == 0 else "Odd" for num in range(6)]  
print(results) # 输出: ['Even', 'Odd', 'Even', 'Odd', 'Even',  
↪ 'Odd']
```

该代码根据每个数字的奇偶性生成不同的字符串。

### 4. 嵌套列表推导式：矩阵转置

列表推导式也支持嵌套，例如可以用于对矩阵进行转置：

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
transpose = [[row[i] for row in matrix] for i in  
↪ range(len(matrix[0]))]  
print(transpose) # 输出: [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

该嵌套列表推导式通过两层循环，完成矩阵的转置操作。

在 Python 中，使用关系运算符可以直接比较两个列表。比较时，会逐元素进行，从第一个元素开始，若发现不相等的元素则返回比较结果；若所有元素都相等，则返回 `True`。对于列表的比较，可以使用以下运算符：

1. **相等运算符** `==`：若两个列表的所有元素相同，则返回 `True`。
2. **不相等运算符** `!=`：若两个列表的至少一个对应元素不相等，则返回 `True`。
3. **大于运算符** `>`：若第一个列表的第一个不等元素比第二个列表的对应元素大，则返回 `True`；若无差异则继续比较下一个元素。
4. **小于运算符** `<`：逻辑与大于相反。

```
list1 = [1, 2, 3]
list2 = [1, 2, 3]
list3 = [1, 2, 4]
list4 = [1, 2]

print(list1 == list2) # 输出: True
print(list1 != list3) # 输出: True
print(list1 > list3)  # 输出: False
print(list1 < list4)  # 输出: False
```

上述代码演示了如何利用关系运算符进行列表比较。值得注意的是，在 Python 3 中，不同类型的元素不能进行比较，例如，无法将字符串与整数进行比较。

在数据分析中，多维列表（或称为嵌套列表）是处理复杂数据结构的有效工具。多维列表可以用来表示矩阵、表格或任何形式的多层数据。其基本语法在 Python 中简单直观，通过将列表嵌套在其他列表中来实现。

## 1. 定义二维列表

在 Python 中，二维列表可以通过如下方式定义：

```
two_dimensional_array = [  
    [1, 2, 3],  
    [4, 5, 6],  
    [7, 8, 9]  
]
```

在这个例子中，`two_dimensional_array` 是一个 3x3 的矩阵，其中每个内部列表代表矩阵的一行。

可以通过双重索引来访问其中的元素，例如：

## 2. 迭代访问和操作

```
element_5 = two_dimensional_array[1][1] # 访问第二行第二列的元素
```

可以使用嵌套循环遍历二维列表中的每个元素。例如：

```
for row in two_dimensional_array:
    for element in row:
        print(element, end=" ")
    print()
```

该代码会逐行打印二维列表中的所有元素。

在 Python 中，列表方法和内建函数在操作列表时的行为存在显著区别。列表方法通常直接修改原始列表，而内建函数则返回一个新值，保持原始列表不变。

例如，使用列表方法 `append()` 和 `sort()` 可以直接修改列表：

```
# 使用 append() 方法
my_list = [7, 2, 3]
my_list.append(4)
print(my_list) # 输出: [7, 2, 3, 4]

# 使用 sort() 方法
my_list.sort()
print(my_list) # [2, 3, 4, 7]
```

相对而言，使用内建函数 `sorted()` 和 `len()` 不会修改原始列表：

```
# 使用 sorted() 函数
original_list = [3, 1, 2]
new_sorted_list = sorted(original_list)
print(original_list) # 输出: [3, 1, 2], 原列表未变
print(new_sorted_list) # 输出: [1, 2, 3]

# 使用 len() 函数
length = len(original_list)
print(length) # 输出: 3
```

由此可见，列表方法会对原始列表进行修改，而内建函数则返回新的值并不影响原列表。

## 26. 常用的操作列表的内置函数

表 7: Python 中常用的操作列表的内置函数

函数名	功能描述	用法示例
<code>len()</code>	返回对象的长度或元素个数	<code>len([1, 2, 3, 4])</code> 返回 4
<code>sum()</code>	返回列表中所有元素的和	<code>sum([1, 2, 3, 4])</code> 返回 10
<code>max()</code>	返回列表中最大值	<code>max([1, 2, 3, 4])</code> 返回 4
<code>min()</code>	返回列表中最小值	<code>min([1, 2, 3, 4])</code> 返回 1
<code>sorted()</code>	返回列表的排序副本	<code>sorted([4, 1, 3, 2])</code> 返回 <code>[1, 2, 3, 4]</code>
<code>reversed()</code>	返回列表的反向迭代器	<code>list(reversed([1, 2, 3]))</code> 返回 <code>[3, 2, 1]</code>
<code>all()</code>	判断列表所有元素是否为真	<code>all([True, True, False])</code> 返回 <code>False</code>
<code>any()</code>	判断列表中是否至少有一个真值	<code>any([False, False, True])</code> 返回 <code>True</code>

`type()` 函数, `dir()` 函数和 `help()` 函数是 Python 中非常实用的内建函数, 常用于探索对象的类型、属性和方法、用法。

在 Python 中，列表（`list`）、`range`、`zip` 和 `enumerate` 都是可迭代对象，都支持迭代操作，即可以逐个访问元素，但它们在概念和用途上有明显的区别。

- 列表是直接存储元素的序列，而 `range`、`zip` 和 `enumerate` 则是生成惰性迭代器，通常不会直接生成所有元素，而是按需生成，可以提高内存利用效率，用于更高效地处理和遍历数据；
- 选择它们取决于具体应用场景，如在需要内存效率时优先使用迭代器，而在需要灵活数据操作时则使用列表。

## range

`range`：生成一个整数序列，通常用于循环中。与列表不同，`range` 返回一个惰性迭代器对象，它不直接存储所有数值，而是按需生成。这使其在处理大量数据时更加高效，因为它节省了内存。

1. 当 `range()` 只有一个参数时，这个参数表示序列的结束值（不包含该值），起始值默认为 `0`。例如：

```
for i in range(5):  
    print(i)  
# 输出: 0 1 2 3 4
```

2. 在使用两个参数时，第一个参数表示起始值，第二个参数表示结束值（不包含该值）。例如：

```
for i in range(1, 6):  
    print(i)
```

3. 第三种形式允许指定步长（`step`），即每次迭代时增加或减少的值。步长可以为负数，以创建递减的序列。例如：

```
for i in range(10, 0, -2):  
    print(i)  
# 输出: 10 8 6 4 2
```

在这个示例中，`range()` 函数以 `-2` 为步长，从 `10` 递减到 `2`。

`range()` 在 Python 中生成的是一个惰性对象，不直接存储所有元素，而是按需生成。这种特性使其在处理大范围数据时更为高效，因为它减少了内存占用。若需要将 `range` 对象转换为列表，可以使用 `list()` 函数，如 `list(range(5))` 将返回 `[0, 1, 2, 3, 4]`。

## enumerate

`enumerate`：为可迭代对象中的每个元素提供一个索引，生成一个包含索引和值的元组迭代器。`enumerate` 适合在需要访问元素及其位置的循环中使用，并且它与列表不同，不会直接创建一个包含所有索引值的完整序列。其基本语法如下：

```
enumerate(iterable, start=0)
```

- `iterable`：一个支持迭代的对象，如列表、元组或字符串。
- `start`（可选）：指定索引的起始值，默认为 `0`。

`enumerate()` 函数通常与 `for` 循环一起使用，以便在遍历时同时获取元素及其索引。例如：

```
fruits = ['apple', 'banana', 'cherry']
for index, fruit in enumerate(fruits):
    print(index, fruit)
# 输出：
# 0 apple
# 1 banana
# 2 cherry
```

`enumerate()` 还可以通过设置 `start` 参数来更改计数的起始值。例如：

```
for index, fruit in enumerate(fruits, start=1):
    print(index, fruit)
# 输出：
# 1 apple
```

**zip**: 将多个可迭代对象（例如列表、元组等）中的元素配对组合成元组，并返回一个迭代器。这个迭代器中的每个元组包含来自各个可迭代对象对应位置的元素。**zip** 的长度取决于最短的输入对象，因此它不会像列表那样存储所有可能的组合，而是逐个生成。其基本语法如下：

`zip(*iterables)`

- **iterables**: 可以是一个或多个可迭代对象，例如列表、元组、字符串等。

## 示例 1: 组合两个列表

```
x = [1, 2, 3]
y = ['one', 'two', 'three']
result = zip(x, y)
print(list(result))
# 输出: [(1, 'one'), (2, 'two'), (3, 'three')]
```

在这个例子中, `zip()` 函数将列表 `x` 和 `y` 中对应位置的元素组合成元组。

## 示例 2: 组合多个列表

```
x = [1, 2, 3]
y = ['one', 'two', 'three']
z = ['I', 'II', 'III']
result = zip(x, y, z)
print(list(result))
# 输出: [(1, 'one', 'I'), (2, 'two', 'II'), (3, 'three', 'III')]
```

### 示例 3：长度不等的可迭代对象

当传入的可迭代对象长度不同时，`zip()` 会在最短的可迭代对象耗尽时停止配对：

```
x = [1, 2, 3, 4]
y = ['a', 'b']
result = zip(x, y)
print(list(result))
# 输出: [(1, 'a'), (2, 'b')]
```

如上所示，`zip()` 函数在 `y` 耗尽时停止，忽略了 `x` 中的剩余元素。

`zip()` 返回的是一个迭代器而非列表，因此在需要看到完整结果时，可以使用 `list()` 将其转换为列表。该特性使其在处理大数据集时更为高效。

元组 (Tuple) 是 Python 中一种重要的数据类型，用于存储多个数据项。与列表类似，元组也是一种序列类型，但与列表的可变性不同，元组是**不可变的**，一旦创建，元组中的元素不能被修改。这使得元组特别适合用于表示那些在程序运行期间不应改变的数据集，如数据库记录、配置参数等。

元组通过一组**圆括号 ()** 来创建，内部的元素用**逗号**分隔。例如，以下代码创建了一个包含三个元素的元组：

```
my_tuple = (1, 2, 3)
print(my_tuple) # 输出: (1, 2, 3)
```

需要注意的是，**逗号**是定义元组的核心部分，即使只有一个元素的元组也需要逗号来区分。例如：

```
single_element_tuple = (1,) # 必须加逗号
print(single_element_tuple) # 输出: (1,)
```

如果省略逗号，Python 会将其视为普通的整数或其他类型，而不是元组。

元组还可以通过**不使用括号**的方式定义，但在多项运算或其他复杂场景中，为了提高可读性，通常建议加上圆括号：

```
my_tuple = 1, 2, 3
print(my_tuple) # 输出: (1, 2, 3)
```

`tuple()` 函数是一个用于创建元组的内置函数。使用 `tuple()` 可以将其他可迭代对象（如列表、字符串、字典等）转换为元组。

### 1. 创建空元组

```
t1 = tuple()
print(t1) # 输出: ()
```

### 2. 从列表创建元组

```
t2 = tuple([1, 4, 6])
print(t2) # 输出: (1, 4, 6)
```

### 3. 从字符串创建元组

```
t3 = tuple('Python')
print(t3) # 输出: ('P', 'y', 't', 'h', 'o', 'n')
```

元组是一种不可变的序列类型，常用于存储多个有序的值。元组的基本操作与列表类似，但由于其不可变性，不能对元组的元素进行修改。

### 1. 访问元组的元素

元组中的元素可以通过索引访问，索引从 0 开始。例如：

```
tuple1 = ('a', 'b', 'c')  
print(tuple1[0]) # 输出: 'a'
```

### 2. 元组的不可变性

元组是不可变的，无法修改其中的元素。例如，试图修改元组元素会引发错误：

```
tuple1 = (1, 2, 3)  
# tuple1[0] = 10 # 这将引发 TypeError
```

### 3. 嵌套元组

元组可以包含其他元组或可变对象。虽然元组本身是不可变的，但其包含的可变对象如列表等，仍然可以修改：

```
nested_tuple = (1, 2, [3, 4])
nested_tuple[2][0] = 'modified'
print(nested_tuple) # 输出: (1, 2, ['modified', 4])
```

### 4. 元组的切片操作

元组支持切片操作，可以获取元组中的子集：

```
tuple1 = (1, 2, 3, 4, 5)
print(tuple1[1:3]) # 输出: (2, 3)
```

## 5. 元组的长度

```
tuple1 = (1, 2, 3)
print(len(tuple1)) # 输出: 3
```

## 6. 元组的遍历

可以使用 **for** 循环遍历元组中的元素:

```
tuple1 = (1, 2, 3)
for item in tuple1:
    print(item)
```

## 7. 检查元素是否存在于元组

```
tuple1 = (1, 2, 3)
print(2 in tuple1) # 输出: True
```

元组虽然是不可变的，但提供了两种常用的内置方法：`count()` 和 `index()`，这些方法在处理元组数据时非常有用，特别是在分析和操作不需要修改的数据时。

### 1. `count()` 方法

`count()` 用于统计指定元素在元组中出现的次数。

```
# 创建包含重复元素的元组
vowels = ('a', 'e', 'i', 'o', 'i', 'u')

# 统计 'i' 出现的次数
count_i = vowels.count('i')
print(count_i) # 输出: 2
```

## 2. `index()` 方法

`index()` 用于查找指定元素在元组中的索引位置，并返回第一个匹配项的索引。如果元素不存在，则会抛出 `ValueError`。其语法为：

```
tuple.index(element, start, end)
```

```
# 创建一个元组
vowels = ('a', 'e', 'i', 'o', 'i', 'u')

# 查找 'i' 的索引
index_i = vowels.index('i')
print(index_i) # 输出: 2
```

在该示例中，`index()` 返回元组中第一个 'i' 的索引值

元组是不可变的序列类型之一，其支持许多适用于所有序列类型的操作。这些操作包括索引访问、切片、连接、重复、成员测试等，能够对序列进行常见的操作和查询，以下结合代码示例进行说明。

### 1. 索引访问 (Indexing)

元组的元素可以通过索引进行访问，索引从 0 开始。如果索引为负数，则表示从序列末尾开始计数。

```
my_tuple = (10, 20, 30, 40)
print(my_tuple[0]) # 输出: 10
print(my_tuple[-1]) # 输出: 40
```

## 2. 切片 (Slicing)

切片允许从序列中获取一个子序列，其格式为 `[start: end: step]`，其中 `start` 是起始索引，`end` 是结束索引（不包括），`step` 是步长（默认为 1）。

```
numbers = (0, 1, 2, 3, 4, 5)
subset = numbers[1:4] # 输出: (1, 2, 3)
reversed_tuple = numbers[::-1] # 输出: (5, 4, 3, 2, 1, 0)
```

切片操作返回一个新的元组，而不修改原始元组。

## 3. 连接 (Concatenation)

可以使用加号 (+) 将两个元组合并成一个新的元组。

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
concatenated_tuple = tuple1 + tuple2 # 输出: (1, 2, 3, 4, 5, 6)
```

## 4. 重复 (Repetition)

使用乘法符号 (\*) 可以将元组重复多次生成一个新元组。

```
original_tuple = (10,)
repeated_tuple = original_tuple * 3 # 输出: (10, 10, 10)
```

## 5. 成员测试 (Membership Testing)

使用 `in` 运算符可以检查一个值是否存在于元组中。

```
my_tuple = ('apple', 'banana', 'cherry')
print('banana' in my_tuple) # 输出: True
```

## 6. 打包与解包 (Packing and Unpacking)

打包 (packing) 和解包 (unpacking) 是处理序列 (如列表和元组) 的重要语法特性。打包是将多个值组合为一个序列, 而解包则是将序列中的值提取到单独的变量中。

打包操作通过将多个值用逗号分隔在一起, 可以创建一个元组。

```
# 打包
values = 1, 2, 3
print(values) # 输出: (1, 2, 3)
```

解包操作使用赋值语句, 将序列中的元素赋值给多个变量, 变量的数量必须与序列中的元素数量相匹配。

```
a, b, c = (1, 2, 3)
print(a) # 输出: 1
print(b) # 输出: 2
print(c) # 输出: 3
```

元组等序列可以被解包成多个变量。这种操作允许快速赋值，并且可以结合星号（\*）将剩余的值赋给一个列表。

```
values = (1, 2, 3, 4)
a, b, *c = values # a = 1, b = 2, c = [3, 4]
data = (1, 2, 3)
a, _, c = data # 这里使用了 _ 作为占位符来忽略中间的值
```

这种操作在处理多个返回值或动态参数传递时非常实用。

上述六种通用操作适用于所有 Python 中的序列类型，包括列表和字符串，而元组的不可变特性使其在某些情况下更具优势，例如用作函数的返回值或键值对。

字符串是**不可变的** (immutable)，一旦创建，字符串中的字符无法直接修改。Python 字符串支持**多种操作和方法**，如字符串的分割 (`split`)、替换 (`replace`)、查找 (`find`) 和大小写转换 (`upper`, `lower`) 等。这些操作在处理文本数据时非常有效，能够简化对大规模文本的分析和处理。

字符串是一种不可变的字符序列，创建字符串的基本语法较为简单，主要通过以下几种方式实现：

### 1. 单引号或双引号创建字符串

使用单引号 (`' '`) 或双引号 (`" "`) 可以创建字符串。例如：

```
string1 = 'Hello, World'
string2 = "Hello, Python"
print(string1) # 输出: Hello, World
print(string2) # 输出: Hello, Python
```

Python 允许在字符串中使用单双引号，只要引号的形式匹配。

## 2. 多行字符串

使用三重引号（`''' '''` 或 `""" """`）可以创建多行字符串。适用于较长的文本或需要保留格式的文本内容。例如：

```
message = """This is a multi-line  
string example."""  
print(message)
```

输出将保留原始的换行格式。

## 3. `str()`

使用 `str()` 函数创建空字符串或将其他数据类型转换为字符串。

**4. 字符串的不可变性:** 一旦字符串被创建, 字符串中的字符无法被修改。这意味着尝试修改字符串中的某个字符会导致错误:

```
s = "hello"
s[0] = 'H' # 报错: TypeError: 'str' object does not support item
          ↪ assignment
# 但可以通过创建新的字符串来变更其内容
new_s = 'H' + s[1:]
print(new_s) # 输出: Hello
```

**5. f-string 格式化:** Python 提供了 f-string 格式化方式, 允许在字符串中嵌入变量或表达式。例如:

```
name = "Alice"
age = 25
print(f"My name is {name} and I am {age} years old.")
```

f-string 不仅语法简洁, 还支持嵌入复杂的表达式。

字符串的基本操作包括通过索引访问字符、使用切片提取子字符串、计算长度、进行成员资格检查、连接与重复字符串操作，以及需注意字符串的不可变性，无法直接修改其中的元素。

## 1. 索引

假设有一个字符串 `s = "Python"`，可以通过索引访问各个字符：

```
s = "Python"
print(s[0])    # 输出: P
print(s[1])    # 输出: y
print(s[-1])   # 输出: n   (使用负索引)
print(s[-2])  # 输出: o   (倒数第二个字符)
```

如果尝试访问超出字符串长度的索引（例如 `s[10]`），会引发 `IndexError` 错误。

## 2. 切片

一种从字符串中提取子字符串的方式，语法同列表切片。

```
s = "ABCDEFGHI"  
print(s[2:7])    # 输出: CDEFG  
print(s[:5])    # 输出: ABCDE (从索引 0 开始, 步长为 1)  
print(s[4:])    # 输出: EFGHI (从索引 4 开始直到结束)  
print(s[::2])   # 输出: ACEGI (每隔一个字符提取)
```

负索引允许从字符串的末尾开始计数。例如：

```
print(s[-4:-1]) # 输出: FGHI (从倒数第4个字符开始提取到倒数第2个)  
print(s[::-1])  # 输出: IHGFEDCBA (反转字符串)
```

### 3. 计算长度

计算字符串的长度可以通过内置函数 `len()` 完成。该函数接受一个字符串作为参数，并返回其中字符的总数，包括空格和标点符号。

```
# 定义一个字符串
feedback = "Customer service was excellent!"
# 计算字符串的长度
length = len(feedback)
# 输出结果
print(length) # 输出: 31
```

4. 成员资格检查: 通过 `in` 和 `not in` 运算符完成，用于检测子字符串是否存在于给定的字符串中。

```
feedback = "The product quality is excellent."
result = "excellent" in feedback
print(result) # 输出: True
```

**5. 字符串连接:** 可以通过使用 `+` 操作符, 将多个字符串组合成一个新的字符串。

```
greeting = "Hello"  
name = "Alice"  
result = greeting + " " + name  
print(result) # 输出: Hello Alice
```

在这个例子中, `+` 操作符将两个字符串连接起来, 生成新的字符串 `"Hello Alice"`。

另一种高效的方式是使用 `join()` 方法, 将一个可迭代对象 (如列表) 中的元素连接为一个字符串, 尤其是在处理大量字符串时更为节省内存:

```
words = ["Python", "is", "fun"]  
result = " ".join(words)  
print(result) # 输出: Python is fun
```

6. 字符串重复字符串的重复可以使用 `*` 操作符实现，将一个字符串按指定次数重复。例如：

```
repeat_str = "ha " * 3
print(repeat_str) # 输出: ha ha ha
```

这种操作常用于生成格式化的分隔符或模式，例如：

```
line = "=" * 10
print(line) # 输出: =====
```

## 1. 字符串拆分: `split()`

`split()` 方法用于按照指定的分隔符将字符串拆分成子字符串列表。默认情况下, `split()` 会按照空格分隔字符串。如果需要, 可以通过传递参数指定不同的分隔符。

```
sentence = "Python is fun to learn"
words = sentence.split() # 按空格拆分
print(words) # 输出: ['Python', 'is', 'fun', 'to', 'learn']

# 使用指定分隔符拆分
sentence = "name,email,phone"
fields = sentence.split(',')
print(fields) # 输出: ['name', 'email', 'phone']
```

## 2. 字符串合并: `join()`

`join()` 方法用于将一个可迭代对象（如列表或元组）中的元素通过指定的分隔符连接成一个字符串。`join()` 方法调用时应在分隔符字符串上调用，并传入需要合并的字符串列表。

```
words = ['Python', 'is', 'fun']
sentence = ' '.join(words) # 使用空格连接
print(sentence) # 输出: Python is fun

# 使用自定义分隔符
fields = ['name', 'email', 'phone']
csv_format = ','.join(fields)
print(csv_format) # 输出: name,email,phone
```

这些操作在处理结构化文本数据（如 CSV 文件）或构建文本报告时非常有用。

### 3. 字符串查找：find() 方法

`find()` 方法用于在字符串中查找子字符串的索引位置。如果找到匹配的子字符串，返回其起始索引；否则返回 `-1`。可以指定可选的起始和结束索引，限制查找的范围。

```
text = "Revenue for the year is estimated at $5 million."  
position = text.find("estimated")  
print(position) # 输出: 24
```

在该示例中，`find()` 方法返回子字符串 `"estimated"` 在字符串中的位置。

## 2. 字符串查找: `index()` 方法

字符串的 `index()` 方法用于查找子字符串在主字符串中的位置。其基本语法为:

```
str.index(sub[, start[, end]])
```

- `sub`: 要搜索的子字符串。
- `start`: 可选, 搜索的起始位置。
- `end`: 可选, 搜索的结束位置。

如果找到该子字符串, `index()` 返回其在主字符串中的最低索引; 若未找到, 则抛出 `ValueError` 异常。以下是几个示例代码:

```
sentence = "Hello, world!"  
position = sentence.index("world")  
print(position) # 输出: 7
```

`start` 和 `end` 参数含义同字符串切片：

```
# 使用起始参数
phrase = "Python is great. Python is versatile."
position = phrase.index("Python", 10)
print(position) # 输出: 17
# 使用结束参数
phrase = "Python is great. Python is versatile."
position = phrase.index("le", 10, 35) # ValueError
position = phrase.index("le", 10, 36)
print(position) # 输出: 34
```

`index()` 方法在处理字符串搜索时非常有效，尤其是在确定子字符串存在的情况下。

### 3. 字符串替换: `replace()` 方法

`replace()` 方法用于将字符串中的某个子字符串替换为另一个子字符串。它的基本语法是:

```
str.replace(old, new, count)
```

其中 `old` 是要替换的子字符串, `new` 是替换后的字符串, `count` 是可选参数, 表示替换的次数。如果不指定 `count`, 将替换所有出现的子字符串。

```
report = "The profit margin was low. The profit margin needs  
↪ improvement."  
new_report = report.replace("profit margin", "revenue")  
print(new_report)  
# 输出: The revenue was low. The revenue needs improvement.
```

在此示例中, `replace()` 方法将所有出现的 "profit margin" 替换为 "revenue", 生成了一个新的字符串。

## 4. 大小写转换

字符串的大小写转换可以通过以下几种常用的内置方法完成，包括 `upper()`、`lower()`、`capitalize()` 和 `swapcase()`，这些方法在处理文本数据时非常有用，尤其是在标准化、数据清洗和文本分析场景中。

`upper()` 方法将字符串中的所有字母转换为大写：

```
text = "python is fun"
upper_text = text.upper()
print(upper_text) # 输出: PYTHON IS FUN
```

`lower()` 方法用于将字符串中的所有字母转换为小写：

```
text = "Hello, WORLD!"
lower_text = text.lower()
print(lower_text) # 输出: hello, world!
```

`capitalize()` 方法将字符串的第一个字母转换为大写，其他字母转换为小写，适用于标题或句子的首字母格式化：

```
text = "python programming"  
capitalized_text = text.capitalize()  
print(capitalized_text) # 输出: Python programming
```

`swapcase()` 方法将字符串中的大写字母转换为小写，小写字母转换为大写：

```
text = "PyThOn PrOgRaMmInG"  
swapped_text = text.swapcase()  
print(swapped_text) # 输出: pYtHoN pRoGrAmMiNg
```

## 5. 去除空白字符

去除字符串中的空白符可以使用三种常见的方法：`strip()`、`rstrip()` 和 `rstrip()`。这些方法分别用于去除字符串两端或特定一端的空白符或其他字符。

`strip()` 方法用于去除字符串开头和结尾的所有空白符（包括空格、换行符、制表符等）。示例如下：

```
text = "  Python is great!  "  
trimmed_text = text.strip()  
print(trimmed_text)  
# 输出: "Python is great!"
```

此方法不会影响字符串中间的空白符，只会去除两端的空白符。

`rstrip()` 方法用于去除字符串右侧的空白符，左侧保持不变：

```
text = " Python is great! "  
left_trimmed_text = text.rstrip()  
print(left_trimmed_text)  
# 输出: "Python is great!"
```

`lstrip()` 方法去除字符串左侧的空白符，右侧保持不变：

```
text = " Python is great! "  
right_trimmed_text = text.lstrip()  
print(right_trimmed_text)  
# 输出: " Python is great!"
```

## 6. 计数

字符串的 `count()` 方法用于计算指定子字符串在目标字符串中出现的次数。该方法非常适合用于文本处理和字符串分析任务，尤其是在需要统计某个字符或子字符串出现频率时。

```
string.count(substring, start=..., end=...)
```

- `substring`: 必选参数，表示需要计数的子字符串。
- `start` (可选): 指定搜索的起始索引，默认为字符串的开头。
- `end` (可选): 指定搜索的结束索引，默认为字符串的末尾。

该方法返回一个整数，表示子字符串在指定范围内出现的次数。如果未找到子字符串，则返回 0。

`start` 和 `end` 参数含义同字符串切片。

## 示例 1: 计数字符串中某字符的出现次数

```
message = 'python is popular programming language'  
print(message.count('p')) # 输出: 4
```

在上述代码中, 'p' 在字符串中总共出现了 4 次。

## 示例 2: 使用 start 和 end 参数

```
string = "Python is awesome, isn't it?"  
substring = "i"  
count = string.count(substring, 8, 25)  
print("The count is:", count) # 输出: 1  
count = string.count(substring, 8, 26)  
print("The count is:", count) # 输出: 2
```

在这个示例中, 计数从索引 8 开始, 到索引 25 结束, 因此只找到 1 次 'i' 字符。

## 7. 类型验证方法

字符串有多个以 `is` 开头的方法，这些方法用于对字符串内容进行各种类型的验证，返回布尔值（`True` 或 `False`）。如下表

表 8: 常见的字符串内容类型验证方法

方法	含义	示例代码	输出
<code>isalnum()</code>	判断字符串是否只包含字母和数字	<code>"Hello123".isalnum()</code>	<code>True</code>
<code>isalpha()</code>	判断字符串是否只包含字母	<code>"Hello".isalpha()</code>	<code>True</code>
<code>isdigit()</code>	判断字符串是否只包含数字	<code>"12345".isdigit()</code>	<code>True</code>
<code>isdecimal()</code>	判断字符串是否只包含十进制字符	<code>"12345".isdecimal()</code>	<code>True</code>
<code>islower()</code>	判断字符串是否全为小写字母	<code>"hello".islower()</code>	<code>True</code>
<code>isupper()</code>	判断字符串是否全为大写字母	<code>"HELLO".isupper()</code>	<code>True</code>
<code>istitle()</code>	判断字符串是否每个单词首字母大写)	<code>"Hello World".istitle()</code>	<code>True</code>
<code>isspace()</code>	判断字符串是否只包含空白字符	<code>" ".isspace()</code>	<code>True</code>

字符串还有许多实用的常用方法，参见讲义中的表 5.1，可以结合 `help` 函数自行学习其他常用的字符串方法的用法。

字符串格式化是一项重要的技能，特别是在处理动态文本输出时，如生成报告、用户提示或数据展示。Python 提供了多种格式化字符串的方法，包括旧式的百分号格式化 (%), `str.format()` 方法，以及较新的 F 字符串格式化 (`f-strings`)。

## 1. 百分号格式化

Python 最早的字符串格式化方式，使用 % 符号作为占位符。例如：

```
name = "Alice"
age = 30
print("Hello, my name is %s and I am %d years old." % (name,
↪ age))
```

上例中，%s 表示字符串占位符，%d 表示整数占位符。该方法虽然简洁，但可读性和灵活性较低，已逐渐被 `str.format()` 和 `f-strings` 所取代。

## 2. `str.format()` 方法

`str.format()` 引入了更加灵活的字符串格式化方式。使用大括号 `{}` 作为占位符，支持位置参数和关键字参数。例如：

```
name = "Bob"  
score = 95.5  
message = "Student: {} | Score: {:.2f}".format(name, score)  
print(message)
```

`{}` 占位符被 `name` 替换，而 `:.2f` 将 `score` 格式化为保留两位小数的浮点数。

### 3. F 字符串格式化 (f-strings)

Python 3.6 引入了 F 字符串格式化，这是目前推荐的格式化方式。它允许在字符串中直接嵌入变量和表达式，使代码更加简洁明了。例如：

```
name = "Eve"  
gpa = 3.8  
message = f"Student: {name} | GPA: {gpa:.2f}"  
print(message)
```

变量 `name` 和 `gpa` 直接嵌入到字符串中，并且可以通过 `{gpa:.2f}` 将 `gpa` 格式化为两位小数的浮点数。F 字符串不仅支持变量插值，还能嵌入复杂的表达式。

## str.format() 方法的位置参数和关键字参数

`str.format()` 方法可以通过位置参数和关键字参数来进行字符串格式化，灵活控制字符串的内容替换。

### 1. 位置参数

使用位置参数时，根据参数在 `format()` 方法中的顺序将值插入到字符串的占位符中，参数的顺序由大括号中的数字索引来决定。例如：

```
message = "Hello, {0}. You are {1} years old.".format("Alice",  
↪ 25)  
print(message)
```

在这个例子中，`{0}` 和 `{1}` 分别表示 "Alice" 和 25 两个位置参数。

## 2. 关键字参数

关键字参数允许通过名称引用参数值，这样使代码更加清晰。例如：

```
message = "Hello, {name}. You are {age} years  
↳ old.".format(name="Bob", age=30)  
print(message)
```

通过使用关键字参数 `name` 和 `age`，可以指定各自的值，使得格式化更加直观。

可以混合使用位置参数和关键字参数，但要注意，位置参数**必须**在关键字参数之前。例如：

```
message = "Hello, {0}. Your balance is  
↳ {balance}.".format("David", balance=230.23)  
print(message)
```

Python 的 `string` 模块提供了一系列用于处理字符串的常量和函数。该模块包含常用的字符集合，如字母、数字、标点符号等，简化了字符串操作。此外，`string` 模块还提供了诸如 `capwords()`、`translate()` 等实用函数，能够实现字符转换、格式化等功能，特别适合在数据处理和文本清理中使用。

`string` 模块中，常量提供了一些预定义的字符集合，用于简化字符串处理。以下是一些常用常量及其基本用法。

```
import string
# 输出所有小写字母
print(" 小写字母:", string.ascii_lowercase)
# 输出所有大写字母
print(" 大写字母:", string.ascii_uppercase)
# 输出所有字母（包含大写和小写）
print(" 所有字母:", string.ascii_letters)
print(" 数字字符:", string.digits) # 输出数字字符
# 输出标点符号
print(" 标点符号:", string.punctuation)
```

`translate()` 函数用于基于一个翻译表（translation table）替换或删除字符串中的字符。该翻译表可以通过 `str.maketrans()` 方法创建，`translate()` 函数结合此表高效地执行字符映射操作。这个功能常用于数据清理或字符串替换等场景。

```
# 导入 string 模块
import string

# 创建一个翻译表，替换字符并移除特定字符
translation_table = str.maketrans("abc", "123", "d")

# 应用 translate 函数
text = "abcdef"
translated_text = text.translate(translation_table)

# 输出结果
print(" 原始文本:", text)
print(" 翻译后的文本:", translated_text)
```

## 37. 特殊字符

在 Python 字符串处理过程中，特殊字符（special characters）是指那些不能直接表示或具有特殊含义的字符。为了在字符串中正确使用这些字符，通常需要使用转义字符（escape character）来避免语法错误或实现特定功能。转义字符以反斜杠（\）为前缀，后跟一个特定字符，来表示一个特殊的含义。常见的 Python 特殊字符和用法如下所示。

表 9: 常见的 Python 转义字符及其用法

转义字符	含义	用法示例
\\	反斜杠	<code>print("C:\\Users\\Path")</code> 输出为 <code>C:\Users\Path</code>
\'	单引号	<code>print('It\'s a test')</code> 输出为 <code>It's a test</code>
\"	双引号	<code>print("She said, \"Hello\"")</code> 输出为 <code>She said, "Hello"</code>
\n	换行	<code>print("Line1\nLine2")</code> 输出为两行: <code>Line1</code> 和 <code>Line2</code>
\t	制表符	<code>print("A\tB")</code> 输出为 <code>A B</code> （插入一个水平制表符）
\b	退格	<code>print("ABC\bD")</code> 输出为 <code>ABD</code> （删除 C）
\r	回车	<code>print("Hello\rWorld")</code> 输出为 <code>World</code> （光标回到行首并覆盖）
\v	垂直制表符	<code>print("A\vB")</code> 输出为 <code>A</code> 和 <code>B</code> 分别位于两行,前面带有垂直制表符

1. 换行符 `\n`: 用于在字符串中插入一个换行。

```
print("Hello\nWorld")
```

2. 制表符 `\t`: 用于插入一个水平制表符。

```
print("Hello\tWorld")
```

3. 单引号 `'` 和双引号 `"`: 当字符串使用单引号或双引号时，如果需要在字符串中包含相同类型的引号，需要使用转义字符。

```
print('It\'s a beautiful day')  
print("He said, \"Python is awesome!\"")
```

4. 反斜杠 `\`: 用于表示一个实际的反斜杠, 因为单个反斜杠在 Python 中是转义字符。

```
print("This is a backslash: \\")
```

5. 回车符 `\r` 和退格符 `\b`: `\r` 用于将光标移到行首, `\b` 则是退格符, 删除前一个字符。

```
print("Hello\rWorld") # 输出为 "Worldo"  
print("Hello\b World") # 输出为 "Hell World"
```

6. 原始字符串 `r` 或 `R`: 在需要保留反斜杠的情况下, 可以通过在字符串前加 `r` 或 `R`, 使反斜杠不被解释为转义字符。

```
print(r"C:\new_folder\test.txt")
```

```
# 使用转义字符打印带有引号的字符串
print("He said, \"Python is fun!\")
# 输出: He said, "Python is fun!"

# 打印包含路径的字符串
print(r"C:\Users\username\Desktop")
# 输出: C:\Users\username\Desktop

# 使用换行符和制表符格式化输出
print("Name:\tJohn\nAge:\t25")
# 输出:
# Name:   John
# Age:   25
```

Python 中，字符串的高级格式化功能为处理复杂的文本输出提供了强大的工具，尤其是在打印表格或整齐的输出时非常重要。主要方法包括 `str.format()` 和 F 字符串（f-strings），它们都支持 Python 的“格式化迷你语言”（formatting mini-language），允许对字符串进行精确控制，例如对齐、填充、宽度设定和精度设置。

Python 的格式化迷你语言是一套强大的工具，允许开发者在格式化字符串时精确控制输出。无论是 `str.format()` 还是 F 字符串（f-strings），都支持这种迷你语言，可以指定输出的宽度、对齐方式、数值格式等。

格式化迷你语言的通用格式为 `{填充内容: 格式参数}`，全部格式参数如下：

```
[fill] [align] [sign] [#] [0] [width] [,] [.precision] [type]
```

常用的几个参数定义如下：

- **fill**：指定用于填充空白的字符，默认是空格。
- **align**：控制对齐方式，`<` 表示左对齐，`>` 表示右对齐，`^` 表示居中对齐。
- **sign**：用于数值的符号处理，`+` 表示始终显示正负号，`-` 表示仅对负数显示符号，空格则在正数前加空格。
- **width**：指定输出字段的最小宽度。
- **precision**：用于控制浮点数的小数位或字符串的最大长度。
- **type**：定义数据类型，例如整数（`d`）、浮点数（`f`）、二进制（`b`）、十六进制（`x`）等。

## 1. 对齐和填充示例

通过设置 `fill` 和 `align` 可以灵活控制字符串的对齐和填充字符:

```
text = "Hello"  
print(f"{text:<10}")    # 左对齐, 宽度 10  
print(f"{text:^10}")    # 居中对齐, 宽度 10  
print(f"{text:*>10}")   # 右对齐, 宽度 10, 用 '*' 填充
```

## 2. 数值符号处理

`sign` 参数用于控制数字的符号显示。其基本语法包括三种选项:

`+`、`-` 和  (空格)。使用 `+` 时, 无论数字为正或负, 都会在前面加上正负号; 使用 `-` 时, 仅在负数前加上负号, 这是默认行为; 而使用空格时, 正数前会加一个空格以便与负数对齐。以下代码示例展示了这些用法:

```
print("{:+} {}".format(58, -58)) # 输出: +58 -58  
print(":{-} {}".format(58, -58)) # 输出: 58 -58  
print("{: } {}".format(58, -58)) # 输出: 58 -58
```

### 3. 参数 # 和 0

# 和 0 这两个参数用于控制数字的格式和输出样式。

# 参数用于指示在数字格式化时添加前缀。例如，当格式化为二进制、八进制或十六进制时，# 将会在结果前添加相应的前缀（如 0b、0o、0x）。# 通常和 type 参数一起使用，例如：

```
print('{:#b}'.format(255))    # 输出: 0b11111111
print('{:#o}'.format(255))    # 输出: 0o377
print('{:#x}'.format(255))    # 输出: 0xff
```

0 参数用于在数字前进行零填充，以达到指定的宽度。当使用 0 时，如果数字的位数不足以满足给定的宽度，将会在左侧补零。例如：

```
print('{:05}'.format(42))     # 输出: 00042
print('{:02x}'.format(255))   # 输出: ff
print('{:#012b}'.format(255)) # 输出: 0b0011111111
```

## 4. width 参数

`width` 参数用于定义字段的最小宽度。它通过指定整数值控制输出时每个字段的最小字符数，确保格式统一和对齐。`width` 的设置可以结合对齐方式和填充字符一起使用。

`width` 参数的格式如为 `":width".format(value)`

这里的 `width` 为一个整数，表示最小字段宽度。例如，以下代码将输出带有最小宽度为 10 个字符的字符串：

```
print("{:10}".format("Hello"))
```

`width` 参数通常与对齐符号一起使用。使用 `<`、`>`、`^` 符号分别表示左对齐、右对齐和居中对齐。

```
print("{:<10}".format("Left"))  
print("{:>10}".format("Right"))  
print("{:^10}".format("Center"))
```

还可以指定填充字符，默认情况下为空格。通过在对齐符号之前添加填充字符，可以填充剩余的空白。

```
print("{:*<10}".format("Fill"))  
print("{:~^10}".format("Test"))
```

**width** 参数在格式化数字时同样有效。例如，将数字格式化为至少 5 个字符宽，并右对齐：

```
print("{:5d}".format(42))
```

## 5. , 参数

逗号参数 ( `,` ) 用于对数字进行分组, 以便增强可读性, 尤其是在处理大数时非常有用。其作用是数值添加千位分隔符。

```
number = 1234567890
print("{:,}".format(number))
```

此示例中, 逗号作为千位分隔符, 使得输出更容易阅读。此功能不仅适用于整数, 还可以与浮点数结合使用:

```
number = 1234567.89
print("{:,.2f}".format(number))
```

在此例中, `:.2f` 控制保留两位小数, 而逗号参数确保了千位分隔符的正确显示。此功能在会计和财务报表中非常有用, 因为大数通常需要以这种方式展示。

## 6. precision 参数

`precision` 参数用于控制浮点数或字符串的精度。其基本形式为在格式说明符中加入点号后跟一个数字，如 `:.2f`。该数字表示需要显示的小数位数或字符串的最大字符数。

**浮点数的精度控制：** `precision` 常用于限制浮点数的小数位数，采用四舍五入。例如，以下代码将一个浮点数截取到小数点后两位：

```
pi = 3.141592653589793
print("Pi to two decimal places: {:.2f}".format(pi))
```

此处，`:.2f` 将浮点数 `pi` 格式化为两位小数。

**字符串的精度控制：**在处理字符串时，`precision` 参数用于限制最大字符数。例如：

```
text = "Python"  
print("{:.3s}".format(text))
```

这里，`:.3s` 限制了字符串的长度为 3 个字符。

**综合应用：**可以将 `precision` 与其他格式化选项结合使用，如宽度、对齐等。例如：

```
num = 123.456789  
print("{:8.3f}".format(num))
```

此代码不仅将浮点数限制为三位小数，还将结果对齐到总宽度为 8 的字段。

## 7. type 参数

`type` 参数用于指定如何格式化不同类型的数据，如整数、浮点数、字符串等。常见的类型代码包括：

**整数格式化 `d`**：将数字格式化为十进制整数。例如：

```
print("{:d}".format(123)) # 输出: 123
```

**浮点数格式化 `f`**：将数字格式化为固定小数点形式，默认保留六位小数。例如：

```
print("{:.2f}".format(123.456789)) # 输出: 123.46
```

**科学计数法 `e` 或 `E`**：将数字格式化为科学计数法，小写 `e` 或大写 `E` 表示指数。例如：

```
print("{:e}".format(1234567)) # 输出: 1.234567e+06
```

## 进制格式化

**b**：将整数格式化为二进制。

**o**：将整数格式化为八进制。

**x** 或 **X**：将整数格式化为十六进制，**x** 为小写，**X** 为大写。

字符串格式化：**s**：将数据格式化为字符串。

百分比：**%**：将数字乘以 100 并显示为百分数。

```
print("{:x}".format(255)) # 输出: ff
print("{:s}".format("Hello")) # 输出: Hello
print("{:.2%}".format(0.25)) # 输出: 25.00%
```

通过组合使用 **type** 参数和其他格式化选项（如宽度、精度），可以灵活地控制输出格式，适用于不同的商业应用场景，如财务数据的显示和报告生成。

在 Python 字符串格式化中，输出字面上的占位符% 和大括号 {} 需要使用特定的转义方法。以下示例展示了如何实现这一功能。

```
# 使用% 格式化输出字面上的%
value = 50
percent_string = "The success rate is %d%%." % value
print(percent_string)
# 使用 {} 格式化输出字面上的 {}
name = "Alice"
braces_string = "Hello, {{name}}! Your score is
↳ {score:.1f}.".format(score=95.5)
print(braces_string)
```

1. **输出字面上的%：**在使用% 格式化时，双百分号%% 表示输出一个字面上的百分号。这里%d%% 中的第一个% 用于格式化整数，后两个% 用于显示字面值。

2. **输出字面上的 {}：**在使用 format() 方法时，双大括号 {{ }} 用于输出字面上的大括号。例如，{{name}} 将输出为 {name}，而不会被视为格式化占位符。

正则表达式（Regular Expression，简称 RegEx）是一种用于匹配文本模式的特殊字符序列。通过定义特定的模式，正则表达式能够高效地查找、匹配和操作字符串。Python 的内置模块 `re` 提供了强大的正则表达式功能，用于执行模式匹配操作，例如搜索、替换和分割字符串。

请参阅讲义第 5.9 节进行自学。

- 集合（Set）是 Python 中的一种内置数据类型，用于存储多个无序元素；
- 集合中的元素是**唯一**的，即不存在重复值，这使其非常适合执行诸如去重、成员测试等操作；
- Python 集合是**可变**的，即可以动态添加或删除元素，但集合中的每个元素必须是不可变的类型（如整数、字符串或元组）；
- 集合支持多种集合运算，如并集、交集、差集和对称差集。

创建集合有两种主要方法：使用花括号 `{}` 直接定义集合，或使用内置的 `set()` 函数。这两种方法均可用于构建一个包含唯一元素的集合。

1. 使用花括号定义集合：将元素放在花括号内并用逗号分隔即可。

```
fruits = {"apple", "banana", "cherry"}  
print(fruits) # 输出: {'apple', 'banana', 'cherry'}
```

集合不允许重复元素，如在定义时加入重复值，它们将被自动去除。

## 2. 使用 `set()` 函数创建集合

适合需要从其他可迭代对象（如列表或元组）转换为集合的情况。可以将一个可迭代对象作为参数传递给 `set()` 函数来创建集合：

```
numbers = set([1, 2, 3, 3, 4])
print(numbers) # 输出: {1, 2, 3, 4}
```

列表中的重复值 `3` 被移除，最终得到一个只包含唯一元素的集合。

**注意：**使用花括号创建空集合并不可行，因为 `{}` 默认创建一个空字典。若要创建一个空集合，必须使用 `set()` 函数：

```
empty_set = set()
print(empty_set) # 输出: set()
```

集合的基本操作包括并集、交集、差集和对称差集，这些操作可以通过运算符或集合的方法来实现。这些集合操作为数据分析和大数据管理中的数据去重、交叉比对及合并操作提供了高效工具。

### 1. 并集 (Union)

并集操作返回两个集合中所有不重复的元素。可以使用 `|` 运算符或 `union()` 方法：

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
print(set1 | set2) # 输出: {1, 2, 3, 4, 5}
print(set1.union(set2)) # 输出: {1, 2, 3, 4, 5}
```

运算符 `|` 和方法 `union()` 功能相同，但 `union()` 方法可以接受其他可迭代对象（如列表）。

## 2. 交集 (Intersection)

返回两个集合中的公共元素。可以使用 `&` 运算符或 `intersection()` 方法，两种方法均支持同时对多个集合进行操作。

```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}
print(set1 & set2) # 输出: {3, 4}
print(set1.intersection(set2)) # 输出: {3, 4}
```

## 3. 差集 (Difference)

差集操作返回一个集合中存在但另一个集合中不存在的元素。可以使用 `-` 运算符或 `difference()` 方法，差集运算结果依赖集合的顺序。

```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5}
print(set1 - set2) # 输出: {1, 2}
print(set1.difference(set2)) # 输出: {1, 2}
```

#### 4. 对称差集 (Symmetric Difference)

对称差集操作返回两个集合中不重复的元素。可以使用 `^` 运算符或 `symmetric_difference()` 方法：

```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}
print(set1 ^ set2) # 输出: {1, 2, 5, 6}
print(set1.symmetric_difference(set2)) # 输出: {1, 2, 5, 6}
```

该操作适用于查找在两个集合中互不相同的元素。

集合提供了多种内置方法来操作和管理数据，这些方法在数据分析、数据清洗和集合运算等场景中非常有用。

### 1. `add()`

`add()` 方法将一个元素添加到集合中，如果元素已经存在，则不会有任何变化，此方法适合在动态数据处理中逐个添加元素。

```
my_set = {1, 2, 3}
my_set.add(4)
print(my_set) # 输出: {1, 2, 3, 4}
```

## 2. update()

`update()` 方法用于将其他可迭代对象（如列表、元组、字典等）的元素添加到当前集合中。如果添加的元素在集合中已存在，则该元素只会出现一次，重复的会被忽略。

```
# 创建一个包含水果名称的集合
fruits = {'apple', 'banana'}
# 定义一个包含新水果的列表
new_fruits = ['cherry', 'date', 'apple']
# 使用 update() 方法将新水果添加到集合中
fruits.update(new_fruits)
```

在上述示例中，`new_fruits` 列表中的元素被添加到 `fruits` 集合中。由于集合不允许重复元素，`apple` 在添加时被忽略。

## 2. `remove()` 和 `discard()`

`discard()` 方法也用于从集合中删除指定的元素。如果该元素存在于集合中，则将其移除；如果元素不存在，则不会进行任何操作，也不会引发异常。

```
# 创建一个包含水果名称的集合
fruits = {'apple', 'banana', 'cherry'}

# 移除存在的元素 'banana'
fruits.remove('banana')

# 尝试移除不存在的元素 'orange'
fruits.remove('orange') # 引发 KeyError 异常

fruits.discard('orange') # 不进行任何操作，也不会引发异常
```

使用 `discard()` 更为安全，因为可以避免潜在的错误。

### 3. pop()

`pop()` 方法用于从集合中移除并返回一个任意元素。由于集合是无序的，因此无法预测 `pop()` 方法会移除哪个元素。如果集合为空，调用 `pop()` 方法将引发 `KeyError` 异常。

```
# 创建一个包含水果名称的集合
fruits = {'apple', 'banana', 'cherry'}

# 使用 pop() 方法移除并返回一个任意元素
removed_fruit = fruits.pop()
print(f" 被移除的水果: {removed_fruit}")
print(f" 剩余的水果集合: {fruits}")
```

#### 4. `issubset()` 和 `issuperset()`

`issubset()` 和 `issuperset()` 分别用于检查一个集合是否为另一个集合的子集或超集:

```
set1 = {1, 2}
set2 = {1, 2, 3}
print(set1.issubset(set2)) # 输出: True
print(set2.issuperset(set1)) # 输出: True
```

在数据分析中，这些方法可以帮助快速验证集合之间的包含关系。

#### 5. `clear()`

`clear()` 方法清空集合中的所有元素，用于重置集合。

```
my_set = {1, 2, 3}
my_set.clear()
print(my_set) # 输出: set()
```

`set` 和 `frozenset` 都是用来存储无序、唯一元素的集合数据类型，但 `set` 是可变的，而 `frozenset` 是不可变的。

`set`：适合在需要动态修改集合内容的场景下使用，当数据集合需要频繁更新或元素需要动态增删时，`set` 提供了灵活性。

`frozenset`：由于不可变性，它是哈希的，这使得它可以作为字典的键或其他集合的元素。

```
# 使用 set
mutable_set = {1, 2, 3}
mutable_set.add(4)
print(mutable_set) # 输出: {1, 2, 3, 4}

# 使用 frozenset
immutable_frozenset = frozenset([1, 2, 3])
print(immutable_frozenset) # 输出: frozenset({1, 2, 3})
# 试图修改 frozenset 将会引发错误
immutable_frozenset.add(4) # AttributeError: 'frozenset' object
↪ has no attribute 'add'
```

字典是一种用于存储键值对（key-value pairs）的数据结构，每个键都是**唯一且不可变**的对象（如字符串、整数、元组等）。这种结构使得字典能够快速、高效地进行数据查找和管理，适合用于表示需要关联数据的场景。

可以通过两种主要方法创建字典：使用花括号 `{}` 或 `dict()` 函数。这两种方法都能够方便地定义键值对，并将数据以字典形式存储。

### 1. 使用花括号 `{}`

这是创建字典最常用的方法，将键值对以 `key: value` 的形式放入花括号中，键和值之间用冒号分隔，多个键值对之间用逗号分隔。

```
student_info = {  
    'name': 'John Doe',  
    'age': 20,  
    'grade': 'A'  
}  
print(student_info)
```

2. 使用 `dict()` 函数 `dict()` 函数也可以用来创建字典，尤其适合从其他数据结构（如列表或元组）转换为字典。例如：

```
employee_data = dict([('name', 'Alice'), ('position',  
→ 'Manager')])  
print(employee_data)
```

在这个示例中，`dict()` 函数接收一个包含键值对的列表并返回一个字典。这种方法可以灵活地从其他结构构建字典。

除了创建包含初始数据的字典，还可以创建一个空字典并逐步添加键值对：

```
inventory = {}  
inventory['apples'] = 50  
inventory['bananas'] = 30  
print(inventory)
```

## 查找元素

查找字典元素的基本操作主要有两种方式：使用方括号 (`[]`) 和 `get()` 方法。

1. 使用方括号 (`[]`) 访问字典元素通过在方括号中指定键，可以直接获取字典中与该键对应的值。这种方式要求键在字典中存在，否则会抛出 `KeyError` 异常。示例如下：

```
# 定义一个字典
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}

# 使用方括号访问字典元素
print(my_dict['name']) # 输出: Alice
print(my_dict['age'])  # 输出: 25
```

这种方法适用于已知键一定存在的情况，因为它可以直接返回值且语法简洁高效。

## 2. 使用 `get()` 方法

`get()` 方法是一种更安全的访问字典元素的方式，它不仅可以返回键对应的值，还允许指定一个默认值，以防键不存在时避免异常：

```
# 使用 get() 方法访问字典元素
value = my_dict.get('name')
print(value) # 输出: Alice

# 访问不存在的键
value = my_dict.get('country', 'Not Found')
print(value) # 输出: Not Found
```

`get()` 方法的优势在于，当键不存在时，可以返回一个自定义的默认值（默认为 `None`），从而避免了异常的产生。这在处理大型或不明确结构的字典时尤为实用。

### 3. 使用 `setdefault()` 方法

`setdefault()` 方法在字典中查找指定键的值。如果键存在，返回对应的值；如果键不存在，则将该键与提供的默认值一起插入字典中，并返回这个默认值。如果未指定默认值，会插入键并将其值设为 `None`。

```
# 1. 键已存在的情况
person = {'name': 'Alice', 'age': 25}
# 键 'age' 存在于字典中
age_value = person.setdefault('age')
print(age_value) # 输出: 25
# 2. 键不存在的情况
person = {'name': 'Alice'}
# 键 'salary' 不存在, 因此插入该键, 值为默认的 None
salary_value = person.setdefault('salary')
print(person) # 输出: {'name': 'Alice', 'salary': None}
age_value = person.setdefault('age', 30) # 插入一个新键 'age',
→ 值为 30
print(person) # 输出: {'name': 'Alice', 'salary': None, 'age':
→ 30}
```

#### 4. 选择合适的方法

- 若键在字典中是必然存在的，直接使用方括号可以提高代码效率。
- 若键的存在与否不确定，且希望有默认返回值或避免异常处理，`get()` 方法和 `setdefault()` 是更好的选择。
- `setdefault()` 方法是一种在修改或初始化字典值时的便捷方式，尤其适用于避免重复检查键是否存在的情形。

这三种方式各有优劣，根据具体情况选择适当的方法，可以提高代码的健壮性和可读性。

## 增加元素

增加字典元素的基本语法主要有两种方式：直接赋值和使用 `update()` 方法。

### 1. 直接赋值法

直接赋值是最常见的方式。通过为字典中的新键赋值，即可添加新的键值对。如果该键已经存在，则会更新对应的值。

```
# 创建一个初始字典
my_dict = {"name": "Alice", "age": 25}

# 增加新键值对
my_dict["city"] = "New York"

print(my_dict)
# 输出: {'name': 'Alice', 'age': 25, 'city': 'New York'}
```

这种方法简单直观，适用于需要快速添加单个键值对的情况。

## 2. 使用 `update()` 方法

`update()` 方法可以一次添加或更新多个键值对。该方法接受一个字典或其他键值对的可迭代对象作为参数，将其内容添加到原字典中。

```
# 使用 update() 方法添加元素
my_dict.update({"country": "USA", "occupation": "Engineer"})

print(my_dict)
# 输出: {'name': 'Alice', 'age': 25, 'city': 'New York',
→ 'country': 'USA', 'occupation': 'Engineer'}
```

`update()` 方法的优点在于可以批量添加多个键值对，提高代码的可读性和效率。此外，如果传入的键已存在，方法会更新该键的值；如果键不存在，则会新增该键值对。

## 删除元素

删除字典元素的基本操作有几种常用的方法，包括 `del` 关键字、`pop()` 方法、`popitem()` 方法以及 `clear()` 方法。

### 1. 使用 `del` 关键字

`del` 关键字可以直接删除字典中的特定键值对。当指定的键存在时，这个操作会移除该键及其对应的值。如果键不存在，则会抛出 `KeyError` 异常。

```
# 创建一个字典
my_dict = {'a': 1, 'b': 2, 'c': 3}
# 删除键 'b'
del my_dict['b']
print(my_dict) # 输出: {'a': 1, 'c': 3}
```

这种方法适用于当确信要删除的键在字典中存在时使用，语法简洁高效。

## 2. 使用 `pop()` 方法

`pop()` 方法可以删除指定键的键值对，并返回被删除的值。

与 `del` 不同的是，`pop()` 允许设置一个默认值，如果键不存在，则返回该默认值而不是抛出异常：

```
# 使用 pop() 删除键 'a'，不设置默认值
my_dict.pop('a') # 若键不存在，则抛出 KeyError 异常
# 使用 pop() 删除键 'a'
del my_dict['a'] # 若键不存在，则抛出 KeyError 异常
# 使用 pop() 删除键 'a'，设置默认值
value = my_dict.pop('a', None) # 若键不存在，则返回 None

print(value)      # 输出: 1
print(my_dict)    # 输出: {'c': 3}
```

`pop()` 方法在希望获取被删除元素的值或避免异常时非常有用。

### 3. 使用 `popitem()` 方法

`popitem()` 用于删除字典中的最后一个键值对（LIFO 顺序，即“后进先出”）。此方法会返回被删除的键值对作为一个元组：

```
# 创建一个字典
my_dict = {'x': 10, 'y': 20}

# 删除并返回最后一个键值对
key, value = my_dict.popitem()

print(key, value) # 输出: y 20
print(my_dict)   # 输出: {'x': 10}
```

如果字典为空，`popitem()` 会抛出 `KeyError` 异常。此方法适合在处理最近添加的元素时使用。

## 4. 使用 `clear()` 方法

`clear()` 方法会清空字典中的所有元素，使其变为空字典：

```
# 清空字典
my_dict.clear()

print(my_dict)  # 输出: {}
```

`clear()` 适用于需要一次性移除所有字典元素的情况。

- `del`：适合直接删除特定键值对，但在键不存在时需要特别注意异常处理。
- `pop()`：推荐在需要返回被删除值或希望避免异常时使用。
- `popitem()`：用于删除最后一个键值对，适合处理最新添加的元素。
- `clear()`：用于清空整个字典。

## 修改元素

修改字典元素的基本语法主要有两种方法：使用方括号 (`[]`) 直接赋值和 `update()` 方法。

### 1. 使用方括号 (`[]`) 直接赋值

要修改字典中的特定元素，可以直接通过方括号引用该键并赋予新的值。如果该键存在，操作会更新值；如果不存在，则会添加一个新的键值对。

```
# 创建一个字典
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}
# 修改现有键的值
my_dict['age'] = 30
# 新增一个键值对
my_dict['country'] = 'USA'
print(my_dict)
```

这种方法直接且高效，适合修改单个键的值或添加新键值对的操作。

## 2. 使用 `update()` 方法

`update()` 方法可以同时更新或添加多个键值对。它接受一个字典作为参数，并将其中的键值对合并到目标字典中。这种方法尤其适合在一次操作中需要修改或添加多个键值对的情况：

```
# 使用 update() 方法更新字典
my_dict.update({'age': 35, 'city': 'Los Angeles'})

print(my_dict)
# 输出: {'name': 'Alice', 'age': 35, 'city': 'Los Angeles',
↪ 'country': 'USA'}
```

在这个例子中，`update()` 方法不仅更新了已有键（如 `age` 和 `city`），还可以添加新的键值对。该方法的优点在于灵活性强，可以高效地进行批量修改。

复制字典有多种方法，最常用的包括 `copy()` 方法、`=` 操作符以及 `deepcopy()` 函数。

## 1. 使用 `copy()` 方法

`copy()` 方法是 Python 字典对象的一个内置方法，用于创建字典的浅拷贝。浅拷贝仅复制字典本身及其键值对，但如果字典中包含可变对象（例如列表或其他字典），这些对象在新旧字典中依然共享引用。

```
# 创建一个字典
original_dict = {'a': 1, 'b': 2, 'c': [1, 2, 3]}
# 使用 copy() 方法进行浅拷贝
copied_dict = original_dict.copy()
print(copied_dict) # 输出: {'a': 1, 'b': 2, 'c': [1, 2, 3]}
```

在此例中，`copy()` 方法创建了一个浅拷贝。在修改浅拷贝中不可变类型（如整数和字符串）的值时，原始字典不会受到影响；但如果修改可变对象（如列表）的内容，原始字典和浅拷贝中的内容会同时更新。

## 2. 使用 = 操作符

简单地将一个字典赋值给另一个变量并不会创建真正的副本，而是生成一个对原始字典的引用。修改其中任何一个字典都会影响到另一个：

```
# 使用 = 操作符复制字典
dict_a = {'name': 'Alice', 'age': 25}
dict_b = dict_a

dict_b['age'] = 30

print(dict_a) # 输出: {'name': 'Alice', 'age': 30}
print(dict_b) # 输出: {'name': 'Alice', 'age': 30}
```

在这个例子中，`dict_b` 和 `dict_a` 共享相同的内存地址，修改其中之一会直接影响另一个。因此，`=` 操作符并不适合在需要独立副本的情况下使用。

### 3. 使用 `deepcopy()` 函数

若需要复制字典及其所有嵌套对象（即实现深拷贝），可以使用 `copy` 模块中的 `deepcopy()` 函数。此方法会递归复制所有嵌套对象，使得新字典完全独立于原字典：

```
from copy import deepcopy

# 创建一个包含可变对象的字典
original_dict = {'a': 1, 'b': [2, 3, 4]}
# 使用 deepcopy() 方法进行深拷贝
deep_copied_dict = deepcopy(original_dict)
# 修改深拷贝中的值
deep_copied_dict['b'].append(5)

print(original_dict)      # 输出: {'a': 1, 'b': [2, 3, 4]}
print(deep_copied_dict)  # 输出: {'a': 1, 'b': [2, 3, 4, 5]}
```

在这个例子中，修改深拷贝中的列表并不会影响到原始字典中的列表内容，因此 `deepcopy()` 适用于需要完全独立副本的场景。

除了基本的增删改查操作外，字典还提供了一些其他常见且有用的操作，例如获取字典长度、判断成员以及其他迭代方法。

### 1. 获取字典长度：len()

```
# 创建一个字典
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}
# 获取字典的长度
length = len(my_dict)
print(length) # 输出: 3
```

### 2. 成员判断：in 和 not in，检查字典中是否存在某个键。

```
# 判断键是否存在于字典中
if 'name' in my_dict:
    print(" 键 'name' 存在于字典中")
if 'email' not in my_dict:
    print(" 键 'email' 不存在于字典中")
```

### 3. 获取所有键、值或键值对: `keys()`、`values()` 和 `items()`

`keys()` 方法返回字典中所有键的视图对象，可以用于迭代或转换为列表:

```
# 获取所有键
keys = my_dict.keys()
print(list(keys)) # 输出: ['name', 'age', 'city']
```

`values()` 方法返回所有值的视图对象，也可以迭代或转换为列表:

```
# 获取所有值
values = my_dict.values()
print(list(values)) # 输出: ['Alice', 25, 'New York']
```

`items()` 方法返回键值对的视图对象，每个元素为一个包含键和值的元组：

```
# 获取所有键值对
items = my_dict.items()
for key, value in items:
    print(f"{key}: {value}")
# 输出:
# name: Alice
# age: 25
# city: New York
```

这些方法使得在不改变字典结构的情况下，可以方便地访问字典的各个组成部分，增强了字典的灵活性，提升了代码的简洁性和可读性。

`format_map()` 方法用于将字典中的值替换到字符串中定义的占位符。这一方法与 `format()` 方法类似，但 `format_map()` 直接接受一个映射（通常为字典）作为参数，而不使用解包操作符 `**`。这种方法在处理字典子类或需要动态填充字符串的场景中非常有用。

```
# 定义字典
data = {'name': 'Alice', 'age': 30}

# 使用 format_map() 方法进行格式化
print('My name is {name} and I am {age} years
      ↪ old'.format_map(data))
# 使用 format() 方法和解包操作符 ** 进行格式化
print('My name is {name} and I am {age} years
      ↪ old'.format(**data))
```

在上述代码中，字典 `data` 中的键 `'name'` 和 `'age'` 与字符串中的占位符匹配，成功替换为相应的值。如果字典中缺少某个占位符对应的键，`format_map()` 将抛出 `KeyError`。

流程控制是指在程序设计中，通过特定的语句和结构来控制程序执行的顺序和逻辑流向。

在商业数据处理领域，流程控制至关重要，因为它决定了数据处理的顺序、条件判断和循环操作，从而确保数据处理过程的准确性和效率。

例如，在处理客户订单时，使用条件语句可以根据订单状态采取不同的处理措施，而循环结构则可用于遍历大量数据记录进行批量处理。通过合理运用流程控制结构，能够构建出高效、可靠的数据处理流程，满足商业应用的需求。

Python 中的流程控制包括：顺序、选择、循环。

条件语句是程序设计中的基本控制结构，用于根据特定条件的真值判断，决定程序执行不同的代码块。在 Python 中，主要使用 `if`、`elif` 和 `else` 语句来实现条件判断。其基本语法如下：

```
if condition1:
    # 当 condition1 为 True 时执行的代码块
elif condition2:
    # 当 condition1 为 False 且 condition2 为 True 时执行的代码块
else:
    # 当上述条件均为 False 时执行的代码块
```

Python 使用缩进来表示代码块的范围。在条件语句中，缩进的代码块仅在条件为 `True` 时执行。

例如，以下代码根据输入的分数输出相应的成绩等级：

```
score = 85

if score >= 90:
    grade = 'A'
elif score >= 80:
    grade = 'B'
elif score >= 70:
    grade = 'C'
elif score >= 60:
    grade = 'D'
else:
    grade = 'F'

print(f"Your grade is: {grade}")
```

在此示例中，程序根据 `score` 的值，依次判断各个条件，输出相应的成绩等级。这种条件判断结构在数据处理、决策分析等领域广泛应用。

## 缩进

在 Python 编程中，代码块的定义依赖于缩进，而非其他编程语言中常见的花括号或关键字。缩进的使用不仅影响代码的可读性，更是 Python 语法的核心部分。根据 Python 官方的 PEP 8 风格指南，建议每一级缩进使用四个空格。

以下示例展示了如何在 Python 中使用缩进来定义代码块：

```
if number % 2 == 0:
    print(f"{number} 是偶数。")
else:
    print(f"{number} 是奇数。")
```

在上述代码中，`if` 和 `else` 语句后的代码块通过缩进来表示其从属关系。如果缩进不一致，Python 解释器将抛出 `IndentationError`，提示缩进错误。

## 单选语句

单选语句（即 `if` 语句）用于根据特定条件的真值判断，决定是否执行某段代码。其基本语法如下：

```
if condition:  
    # 当 condition 为 True 时执行的代码块
```

其中，`condition` 是一个布尔表达式。当 `condition` 为 `True` 时，缩进的代码块将被执行；否则，代码块将被跳过。

例如，以下代码根据输入的数字判断其正负性：

```
number = int(input(" 请输入一个整数: "))  
  
if number > 0:  
    print(" 该数字是正数。")
```

## 双选语句

双选语句（即 `if-else` 语句）用于根据条件的真值判断，决定程序执行不同的代码块。其基本语法如下：

```
if condition:
    # 当 condition 为 True 时执行的代码块
else:
    # 当 condition 为 False 时执行的代码块
```

例如，以下代码根据输入的年龄判断是否为成年人：

```
age = int(input(" 请输入年龄: "))

if age >= 18:
    print(" 您是成年人。")
else:
    print(" 您未成年。")
```

## 多选语句

多选语句（即 `if-elif-else` 语句）用于根据多个条件的真值判断，决定程序执行的代码块。其基本语法如下：

```
if condition1:
    # 当 condition1 为 True 时执行的代码块
elif condition2:
    # 当 condition1 为 False 且 condition2 为 True 时执行的代码块
elif condition3:
    # 当前面的条件均为 False 且 condition3 为 True 时执行的代码块
else:
    # 当上述所有条件均为 False 时执行的代码块
```

其中，`condition1`、`condition2`、`condition3` 等为布尔表达式。程序从上至下依次判断各条件，执行第一个为 `True` 的条件对应的代码块；如果所有条件均为 `False`，则执行 `else` 下的代码块。

循环结构是编程语言中的基本控制结构之一，用于重复执行特定代码块，直到满足指定的条件。在 Python 中，主要有两种循环结构：

`for` 循环和 `while` 循环。`for` 循环用于遍历序列（如列表、元组、字符串等）或其他可迭代对象，而 `while` 循环则在给定条件为真时反复执行代码块。

在商业数据处理中，循环结构具有重要作用。它们用于自动化重复性任务，如批量处理数据记录、迭代计算统计指标、遍历数据集以查找特定模式或异常等。通过使用循环结构，可以提高数据处理的效率和准确性，减少人工操作的错误率。

`for` 循环用于遍历序列（如列表、元组、字符串等）或其他可迭代对象。其基本语法如下：

```
for item in iterable:  
    # 执行的代码块
```

其中，`item` 是循环变量，在每次迭代中获取 `iterable` 中的下一个元素。`iterable` 是一个可迭代对象，如列表、元组、字符串、集合、字典或生成器（`range`、`zip`、`enumerate`）。

以下示例展示了如何使用 `for` 循环遍历列表中的元素：

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

`for` 循环依次将每个元素赋值给变量 `fruit`，并在循环体内打印该变量的值。

`while` 循环用于在指定条件为真时，反复执行代码块。基本语法：

```
while condition:  
    # 执行的代码块
```

其中，`condition` 是一个布尔表达式。当 `condition` 为 `True` 时，循环体内的代码将被执行；当 `condition` 为 `False` 时，循环终止，程序继续执行后续代码。

以下示例展示了如何使用 `while` 循环打印 1 到 5 的数字：

```
i = 1  
while i <= 5:  
    print(i)  
    i += 1
```

需要注意的是，确保循环条件最终会变为 `False`，以避免出现无限循环。在上例中，通过在循环体内递增 `i`，保证了循环的正常结束。

## break 语句和 continue 语句

在 Python 的循环结构中，`break` 和 `continue` 语句用于控制循环的执行流程。`break` 语句用于立即终止当前循环，跳出循环体，继续执行后续代码；`continue` 语句则用于跳过当前迭代，直接开始下一次循环。

### 1. `break` 语句的用法

`break` 语句通常用于在满足特定条件时退出循环。

```
# for 循环中的 break 示例
for number in range(1, 11):
    if number == 5:
        break
    print(number)
```

在此示例中，循环遍历数字 1 到 10。当 `number` 等于 5 时，`break` 语句被触发，循环立即终止，后续数字不再打印。

```
# while 循环中的 break 示例
i = 1
while i <= 10:
    if i == 5:
        break
    print(i)
    i += 1
```

在此示例中，`while` 循环不断增加变量 `i` 的值。当 `i` 等于 5 时，`break` 语句被触发，循环立即终止。

## 2. `continue` 语句的用法

`continue` 语句用于跳过当前迭代的剩余代码，直接进入下一次循环迭代。

```
# for 循环中的 continue 示例
for number in range(1, 6):
    if number == 3:
        continue
    print(number)
```

在此示例中，循环遍历数字 1 到 5。当 `number` 等于 3 时，`continue` 语句被触发，当前迭代剩余代码被跳过，数字 3 未被打印，循环继续进行。

```
# while 循环中的 continue 示例
i = 0
while i < 5:
    i += 1
    if i == 3:
        continue
    print(i)
```

在此示例中，`while` 循环将变量 `i` 的值从 1 增加到 5。当 `i` 等于 3 时，`continue` 语句被触发，跳过当次打印操作，直接进入下一次循环。数字 3 未被打印。

`break` 和 `continue` 语句在控制循环流程中极为有用，能够灵活地管理循环终止与跳过的条件，但需谨慎使用，以避免产生难以调试的逻辑错误，如，无限循环。

### 3. 使用 `while True` 和 `break` 实现特定的循环控制

使用 `while True` 与 `break` 语句相结合，可以实现特定的循环控制。`while True` 创建一个无限循环，而 `break` 语句用于在满足特定条件时终止该循环。这种结构在需要持续运行某个过程，直到满足特定条件时尤为有用。

#### 示例：用户输入验证

以下示例展示了如何使用 `while True` 和 `break` 语句实现用户输入验证，确保用户输入有效的整数：

```
while True:
    user_input = input(" 请输入一个整数: ")
    if user_input.isdigit():
        number = int(user_input)
        print(f" 您输入的整数是: {number}")
        break
    else:
        print(" 输入无效, 请输入一个整数。")
```

代码解析:

- `while True`: 创建一个无限循环, 持续提示用户输入。
- `input()` 函数: 获取用户输入, 并将其存储在 `user_input` 变量中。
- `if user_input.isdigit():`: 检查用户输入是否为数字字符串。
  - 如果是数字字符串:
    - 将其转换为整数类型, 并存储在 `number` 变量中。
    - 打印用户输入的整数。
    - 使用 `break` 语句终止循环。
  - 如果不是数字字符串:
    - 提示用户输入无效, 要求重新输入。

通过这种方式, 程序能够持续提示用户输入, 直到获得有效的整数输入为止。

## 1. 嵌套 for 循环

多重 `for` 循环（即嵌套 `for` 循环）用于在一个 `for` 循环内部再嵌套一个或多个 `for` 循环，以遍历多维数据结构或生成复杂的迭代模式。这种结构在处理二维数组、矩阵运算或生成特定模式时尤为常见。

基本语法：

```
for outer_element in outer_sequence:
    for inner_element in outer_element:
        # 执行的代码块
```

在上述结构中，外层 `for` 循环遍历 `outer_sequence` 中的每个元素。对于外层循环的每次迭代，内层 `for` 循环都会遍历 `inner_sequence` 中的所有元素。这种嵌套关系可以扩展到多层次，但应注意层次过多可能导致代码复杂性增加。

## 示例：生成乘法表

以下示例展示了如何使用多重 `for` 循环生成一个  $9 \times 9$  乘法表：

```
# 使用双重 for 循环打印 9x9 乘法表
for i in range(1, 10): # 外层循环，控制乘法表的行数，从 1 到 9
    for j in range(1, i + 1): # 内层循环，控制每行中列的输出范围
        print(f"{j}x{i}={i * j}", end="\t") #
        ↪ 打印当前的乘法表达式，并用制表符隔开
    print() # 每行结束后换行，进入下一行
```

## 2. 嵌套条件语句

嵌套条件语句是指在一个 `if` 或 `else` 块中包含另一个 `if` 语句。这在需要根据多个条件进行判断时非常有用。

```
x = 10
y = 5

if x > 0:
    if y > 0:
        print("x 和 y 都是正数")
    else:
        print("x 是正数, 但 y 不是正数")
else:
    print("x 不是正数")
```

### 3. 条件语句与循环的嵌套

在循环内部使用条件语句，或在条件语句内部使用循环，是实现复杂逻辑的常见方式。

```
numbers = [1, 2, 3, 4, 5]

for num in numbers:
    if num % 2 == 0:
        print(f"{num} 是偶数")
    else:
        print(f"{num} 是奇数")
```

在上述示例中，`for` 循环遍历列表中的每个数字，并使用 `if` 语句判断其奇偶性。

## 4. 嵌套循环与条件语句

在嵌套循环中使用条件语句，可以实现更复杂的逻辑控制。

```
for i in range(3):
    for j in range(3):
        if i == j:
            print(f"i 和 j 都是 {i}")
        else:
            print(f"i = {i}, j = {j}")
```

上述代码在 **i** 等于 **j** 时输出特定信息，否则输出 **i** 和 **j** 的值。

通过合理使用嵌套控制结构，可以编写出功能强大且灵活的程序，以满足复杂的业务需求。

`assert`、`pass`、`exec` 和 `eval` 是四个重要的语句或函数，分别用于不同的场景。这些语句和函数在流程控制中扮演辅助或特殊用途角色。通过 `assert` 验证流程条件、`pass` 填充流程结构、`exec` 和 `eval` 实现动态代码执行，均可以提升代码的灵活性和适应性，使流程控制更加丰富和动态。

### 1. `assert` 语句

`assert` 用于在程序中插入调试断言。当条件为 `False` 时，程序会引发 `AssertionError` 异常。这在测试和调试时非常有用。

```
x = 10
assert x > 0, "x should be positive"
```

在上述代码中，如果 `x` 不大于 0，程序将抛出 `AssertionError`，并显示消息“`x should be positive`”。

## 2. pass 语句

`pass` 是一个空操作，占位符语句。在需要语法上需要语句但不执行任何操作的地方使用。

```
for item in range(5):  
    if item % 2 == 0:  
        pass # 占位符，无操作，程序继续执行  
    else:  
        print(item)
```

上例中，使用 `pass` 作为占位符，当 `item` 为偶数时，程序不进行任何操作。

### 3. `exec` 函数

`exec` 用于动态执行储存在字符串或文件中的 Python 代码。它可以执行更复杂的 Python 代码。

```
code = """
for i in range(5):
    print(i)
"""
exec(code)
```

上述代码将动态执行字符串中的代码，输出 0 到 4。

`exec` 执行字符串里的 Python 代码，返回 `None`。

## 4. eval 函数

`eval` 用于计算存储在字符串中的简单表达式，并返回结果。

与 `exec` 不同，`eval` 只能处理单个表达式，不能执行复杂的代码结构。

```
expression = "3 * 4 + 5"  
result = eval(expression)  
print(result) # 输出 17
```

在此示例中，`eval` 计算字符串中的表达式，并返回结果 17。

`eval` 执行字符串里的表达式代码，返回表达式计算结果。

使用 `exec` 和 `eval` 时应谨慎，尤其是在处理不受信任的输入时，因为它们可能带来安全风险。

抽象是计算机科学中的核心概念，旨在隐藏复杂的实现细节，突出核心功能，从而提高程序的可读性、可维护性和扩展性。

在编程实践中，函数抽象是实现这一目标的主要手段之一。通过将重复的逻辑封装在函数中，开发者可以减少代码冗余，提升代码的模块化程度。

例如，考虑一个需要计算多个矩形面积的场景。如果不使用函数，可能会多次编写相同的计算逻辑：

```
# 计算第一个矩形的面积
width1 = 5
height1 = 10
area1 = width1 * height1

# 计算第二个矩形的面积
width2 = 3
height2 = 7
area2 = width2 * height2

# 计算第三个矩形的面积
```

上述代码存在明显的重复，增加了维护难度。通过定义一个计算矩形面积的函数，可以有效地抽象出重复的逻辑：

```
def calculate_area(width, height):  
    return width * height  
  
# 使用函数计算面积  
area1 = calculate_area(5, 10)  
area2 = calculate_area(3, 7)  
area3 = calculate_area(6, 9)
```

通过这种方式，计算面积的逻辑被封装在 `calculate_area` 函数中，调用者只需提供不同的参数即可复用该逻辑。这不仅减少了代码冗余，还提高了代码的清晰度和可维护性。

函数的定义使用 `def` 关键字，后跟函数名和括号中的参数列表。函数内部的代码块通常由缩进的语句组成，而 `return` 语句用于返回函数的结果。

## 1. 函数的定义与调用

函数的基本定义格式如下：

```
def function_name(parameters):  
    # 执行的代码块  
    return result
```

在定义函数时，`def` 后接函数名，再由圆括号包围的参数列表，可以为函数的参数指定默认值。如果函数没有参数，可以省略。如果函数需要返回一个结果，可以使用 `return` 语句将计算结果返回给调用者。

## 2. 参数的传递

Python 支持多种参数传递方式，包括位置参数、关键字参数和混合方式。位置参数 (调用时不写参数名字) 是最常见的类型，调用时传入的值按照位置匹配到相应的参数。

```
def add(a, b):  
    return a + b  
  
result = add(3, 5)  
print(result) # 输出 8
```

除了位置参数，Python 还支持使用关键字参数 (调用时写参数名字) 来进行函数调用，这使得传递参数时不受位置顺序的限制。

```
def describe_pet(animal_type, pet_name):  
    print(f"I have a {animal_type} named {pet_name}.")  
  
describe_pet(animal_type="dog", pet_name="Buddy")
```

### 3. 混合方式

混合使用位置参数和关键字参数进行函数调用时，位置参数必须在关键字参数前面。

```
def multiply(a, b):  
    return a * b  
  
print(multiply(4, b = 5)) # 输出 20  
print(multiply(a = 4, 5)) # 错误，位置参数必须在关键字参数前面
```

#### 4. 返回值的使用

函数可以返回值，`return` 语句用于指定返回值。如果函数没有 `return` 语句，它会默认返回 `None`。返回的值可以用于后续的计算。

```
def multiply(a, b):  
    return a * b  
  
result = multiply(4, 5)  
print(result) # 输出 20
```

## 5. 示例代码：带参数的函数

以下是一个包含多个参数、返回值以及默认参数的完整示例：

```
def calculate_area(length, width=1):  
    """ 计算矩形的面积，宽度参数有默认值 """  
    return length * width  
  
# 使用位置参数  
area1 = calculate_area(5, 3)  
print(f"Area 1: {area1}") # 输出 Area 1: 15  
  
# 使用默认参数  
area2 = calculate_area(5)  
print(f"Area 2: {area2}") # 输出 Area 2: 5
```

在此示例中，`width` 参数有默认值，因此在调用 `calculate_area(5)` 时，`width` 会自动取默认值 `1`。

## 6. 文档字符串

文档字符串（docstring）和函数注解（function annotation）是提高代码可读性和可维护性的关键工具。文档字符串用于描述模块、类或函数的功能和用法，而函数注解用于为函数的参数和返回值提供类型提示。

文档字符串是位于模块、类或函数定义内部的字符串字面量，通常使用三重引号（`""" """`）包裹。文档字符串应简洁明了，首行应为简短的描述，后续可包含更详细的说明。

```
def add(a, b):  
    """  
    返回两个数的和。  
  
    参数:  
    a (int): 第一个加数。  
    b (int): 第二个加数。  
  
    返回:  
    int: 两个数的和。  
    """  
    return a + b
```

在上述示例中，函数 `add` 的文档字符串清晰地描述了函数的功能、参数和返回值。这有助于用户快速理解函数的用途和使用方法。

## 7. 函数注解 (Function Annotation)

函数注解是 Python 3 引入的特性，用于为函数的参数和返回值添加元数据，通常用于类型提示。注解的语法是在参数名后使用冒号加类型提示，返回值注解则在参数列表后使用箭头加类型提示。函数注解仅用于提供信息，不会影响函数的实际行为。

```
def add(a: int, b: int) -> int:  
    return a + b
```

在此示例中，函数 `add` 的参数 `a` 和 `b` 以及返回值均被注解为整数类型。这为阅读代码的人提供了关于参数和返回值类型的有用信息。

## 1. 形式参数与实际参数

函数的定义和调用涉及两个关键概念：形式参数（formal parameters）和实际参数（actual parameters）。形式参数是在函数定义时指定的变量，用于接收传入的数据；实际参数则是在函数调用时提供的具体值或表达式。

```
def multiply(a, b):  
    return a * b  
  
result = multiply(4, 7)  
print(result) # 输出: 28
```

在此示例中，`a` 和 `b` 是形式参数，`4` 和 `7` 是实际参数。调用 `multiply(4, 7)` 时，实际参数 `4` 和 `7` 被传递给形式参数 `a` 和 `b`，函数返回它们的乘积 `28`。

## 2. 默认参数 (Default Arguments)

默认参数是函数定义时为参数指定的默认值。如果在调用时没有为该参数提供值，函数将使用默认值。

```
def greet(name, age=25):  
    print(f"Hello, {name}! You are {age} years old.")  
  
greet("Bob") # 使用默认值  
greet("Alice", 30) # 使用提供的值
```

在这个例子中，`age` 参数有一个默认值 `25`。如果在调用函数时没有传入 `age` 的值，默认值 `25` 将被使用。

**注意：**定义函数时，有默认值的参数在后，没默认值的参数在前。

### 3. 不定长参数 (Arbitrary Arguments)

当不确定传入函数的参数个数时，可以使用不定长参数。Python 提供了 `*args` 和 `**kwargs` 两种方式来处理这种情况。

- `*args` 用于传递任意数量的位置参数。
- `**kwargs` 用于传递任意数量的关键字参数。

```
def sum_numbers(*numbers):  
    total = sum(numbers)  
    print(f"Sum: {total}")  
  
sum_numbers(1, 2, 3, 4) # 传入多个位置参数
```

在这个例子中，`*numbers` 接受了多个位置参数并将它们放入一个元组 `numbers` 中，之后通过 `sum()` 函数计算它们的和。

```
def display_info(**info):  
    for key, value in info.items():  
        print(f"{key}: {value}")  
  
display_info(name="David", age=45, job="Engineer")
```

在这个例子中，`**info` 接受了多个关键字参数并将它们存储为一个字典 `info`，可以通过 `items()` 方法遍历字典中的每一对键值。

在 Python 编程中，变量的作用域决定了变量的可访问范围。主要分为局部变量和全局变量两种类型。

## 1. 局部变量 (Local Variables)

局部变量是在函数内部定义的变量，其作用域仅限于该函数内部。当函数被调用时，局部变量被创建；函数执行结束后，局部变量被销毁。局部变量在函数外部无法访问。

```
def example_function():  
    local_var = "I am a local variable"  
    print(local_var)  
  
example_function()  
print(local_var) # 试图在函数外部访问局部变量
```

上例中，`local_var` 是在函数 `example_function` 内部定义的局部变量。在函数内部打印该变量时，输出正常。然而，当尝试在函数外部访问 `local_var` 时，Python 抛出 `NameError`，提示未定义该变量。

## 2. 全局变量 (Global Variables)

全局变量是在函数外部定义的变量，其作用域覆盖整个模块 (整个.ipynb 文件或.py 文件)。全局变量可以在函数内部和外部访问。然而，在函数内部如果需要修改全局变量的值，必须使用 `global` 关键字声明，否则 Python 会将其视为新的局部变量。

```
global_var = "I am a global variable"

def example_function():
    print(global_var) # 在函数内部访问全局变量

example_function()
print(global_var) # 在函数外部访问全局变量
```

在上例中，`global_var` 是在函数外部定义的全局变量。在函数 `example_function` 内部和外部均可访问该变量，且输出结果一致。

### 3. 在函数内部修改全局变量

如果需要在函数内部修改全局变量的值，必须使用 `global` 关键字声明该变量。否则，Python 会在函数内部创建一个同名的局部变量，而不会影响全局变量的值。

```
global_var = "I am a global variable"

def example_function():
    global global_var
    global_var = "I have been modified"
    print(global_var)

example_function()
print(global_var) # 检查全局变量是否被修改
```

上例中，使用 `global` 关键字声明 `global_var`，表示在函数内部对全局变量进行修改。因此，函数内部和外部的 `global_var` 值均被修改。

函数式编程（Functional Programming）是一种编程范式，强调使用纯函数进行计算，避免副作用，并鼓励函数作为一等公民。在 Python 中，尽管其主要是面向对象的，但也提供了对函数式编程的支持。

## 1. 纯函数

纯函数是指在相同输入下总是产生相同输出且没有副作用的函数。这意味着函数的执行不依赖于外部状态，也不改变外部状态。

```
def add(a, b):  
    return a + b
```

上述函数 `add` 在相同的输入 `a` 和 `b` 下，总是返回相同的结果 `a + b`，且不影响外部状态，因此是一个纯函数。

## 2. lambda 表达式

lambda 表达式用于创建匿名函数，即没有名称的简短函数。其语法形式为：

```
lambda 参数列表: 表达式
```

参数列表可以包含多个参数，表达式是对这些参数进行操作并返回结果。lambda 表达式常用于需要简短函数且不想正式定义函数的场景。

```
# 定义一个 lambda 表达式，将输入值加 10  
add_ten = lambda x: x + 10  
print(add_ten(5)) # 输出: 15
```

上例定义了一个 lambda 表达式并将其赋值给变量 `add_ten`，接受一个参数 `x`，返回 `x` 加 10 的结果。调用 `add_ten(5)` 时，输出为 15。

### 3. 高阶函数

高阶函数是指接受一个或多个函数作为参数，或返回一个函数作为结果的函数。Python 内置的 `map`、`filter` 和 `reduce` 函数就是高阶函数的典型例子。

#### (1) `map` 函数

`map` 函数用于将指定的函数依次作用于可迭代对象的每个元素，返回一个包含结果的迭代器。其语法为：

```
map(function, iterable, ...)
```

- `function`：应用于每个元素的函数；
- `iterable`：一个或多个可迭代对象。

```
# 将列表中的每个元素平方
numbers = [1, 2, 3, 4, 5]
squared_numbers = map(lambda x: x**2, numbers)
print(list(squared_numbers)) # 输出: [1, 4, 9, 16, 25]
```

在此示例中，`map` 函数将匿名函数 `lambda x: x**2` 应用于列表 `numbers` 的每个元素，返回其平方值的迭代器。使用 `list` 函数将其转换为列表后，输出结果为 `[1, 4, 9, 16, 25]`。

`map` 可接受多个可迭代对象，如下例：

```
weights = [7, 9, 10, 5, 8, 4, 2, 1, 6, 3, 7, 9, 10, 5, 8, 4, 2]
ids = '110113198702121018'
sum(map(lambda x,y:x*int(y),weights,ids)) # 169
```

## (2) filter 函数

`filter` 函数用于过滤可迭代对象中的元素，保留使指定函数返回 `True` 的元素，返回一个迭代器。其语法为：

```
filter(function, iterable)
```

- `function`：用于判断每个元素是否保留的函数，返回布尔值。
- `iterable`：可迭代对象。

```
# 过滤出列表中的偶数
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(list(even_numbers)) # 输出: [2, 4, 6]
```

在上例中，`filter` 函数将匿名函数 `lambda x: x % 2 == 0` 应用于列表 `numbers` 的每个元素，保留偶数元素。返回一个 `filter` 类型迭代器。

### (3) reduce 函数

`reduce` 函数用于对可迭代对象中的元素进行累积操作，依次将前两个元素传递给指定函数，得到结果后，再与下一个元素继续进行累积，直到处理完所有元素，返回最终结果。在 Python 3 中，`reduce` 函数位于 `functools` 模块中，需要先导入。其语法为：

```
from functools import reduce
reduce(function, iterable[, initializer])
```

- `function`：用于累积操作的函数，接受两个参数。
- `iterable`：可迭代对象。
- `initializer`（可选）：初始值，若提供，则首先与可迭代对象的第一个元素进行累积。

```
from functools import reduce

# 计算列表元素的累积乘积
numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
print(product) # 输出: 120
# 初始值为 2
product = reduce(lambda x, y: x * y, numbers, 2)
print(product) # 输出: 240
```

在此示例中，`reduce` 函数将匿名函数 `lambda x, y: x * y` 依次应用于列表 `numbers` 的元素，计算其累积乘积。最终结果为 `120`。当初始值为 `2` 时，累积乘积为 `240`。

递归是一种通过重复调用自身来解决问题的方法，适用于具有自相似结构的问题，即可以分解为若干规模缩小的相似子问题。递归函数是一类通过调用自身来解决问题的函数，特别适用于以下几类问题：

- **数学归纳类问题**：像阶乘、斐波那契数列、幂运算等问题，具有明显的数学递归定义，因此使用递归实现代码简洁明了。
- **分治问题**：这类问题可以划分为若干小规模的同类型子问题，并通过递归解决后合并结果。例如，归并排序和快速排序都通过递归分解排序任务，分别解决子数组的排序问题。
- **树形或图形结构遍历**：树结构的遍历（如二叉树的前序、中序和后序遍历）天然适合递归。通过递归，函数可以直接处理每个节点并对其子节点继续递归调用。图结构的遍历如深度优先搜索（DFS）也可通过递归实现。
- **回溯问题**：对于路径选择和解空间搜索问题，递归是实现回溯算法的自然方式。例如，解决数独问题、生成排列组合以及八皇后问题等，递归函数能更清晰地表达解空间的遍历过程。

递归函数的基本结构通常如下所示：

```
def recursive_function(parameters):  
    if base_case_condition: # 基线条件  
        return base_case_value # 返回基线值  
    else:  
        return recursive_function(modified_parameters) # 递归调用
```

为避免无限循环和栈溢出，递归函数必须定义基线条件和递归条件。

- **基线条件：**基线条件是递归的终止条件，当问题规模足够小且可直接解决时，递归停止并返回结果。没有基线条件，递归会一直进行，导致栈溢出。
- **递归条件：**递归条件指函数在某些情况下调用自身，解决更小规模的子问题，直到达到基线条件。

### 示例 1: 计算阶乘

阶乘是一个经典的递归问题，定义为：

- $0! = 1$
- $n! = n \times (n - 1)!$ ，其中  $n > 0$

递归实现阶乘函数的代码如下：

```
def factorial(n):  
    if n <= 1: # 基线条件  
        return 1  
    else: # 递归条件  
        return n * factorial(n - 1)
```

在这个实现中：

- 基线条件是当  $n \leq 1$  时，返回 1。
- 递归条件是返回  $n \times factorial(n - 1)$ ，每次将问题规模减小 1。

## 示例 2: 斐波那契数列

斐波那契数列是另一个经典的递归问题，其定义为：

- $F(0) = 0$
- $F(1) = 1$
- $F(n) = F(n-1) + F(n-2)$ ，其中  $n > 1$

递归实现斐波那契数列的代码如下：

```
def fibonacci(n):  
    if n <= 0: # 基线条件  
        return 0  
    elif n == 1: # 基线条件  
        return 1  
    else: # 递归条件  
        return fibonacci(n - 1) + fibonacci(n - 2)
```

- 基线条件分别是当  $n = 0$  时返回 0，当  $n = 1$  时返回 1。
- 递归条件是返回  $fibonacci(n-1) + fibonacci(n-2)$ ，通过递归逐步减小问题规模。

## 使用递归函数的注意事项

- ❶ **基线条件（终止条件）**：确保函数定义了正确的基线条件，避免无限递归。基线条件是递归终止的必要条件，缺少基线条件可能导致栈溢出。
- ❷ **递归深度和性能**：递归会占用调用栈，每次递归调用都将使用额外的栈空间。因此，递归深度过大会导致栈溢出。Python 默认的递归深度是有限的，深度递归会触发 `RecursionError`。可以考虑通过迭代、缓存或动态规划来替代深度递归。
- ❸ **重复计算和效率优化**：某些递归算法可能导致大量重复计算，例如在计算斐波那契数列时，可对已计算的结果进行缓存（如通过 `functools.lru_cache` 实现）来提升效率。这种优化技术称为记忆化（Memoization），能显著降低时间复杂度。
- ❹ **可读性和平衡性**：在某些情况下，递归函数虽然简洁，但若过度依赖递归，可能会降低代码的可读性。编写时应权衡递归和迭代，选择更直观、效率更高的实现方式。

`open()` 函数是进行文件操作的基础。该函数用于打开一个文件，并返回一个文件对象，之后可对该对象进行读取或写入操作。其基本语法如下：

```
open(file, mode='r', buffering=-1, encoding=None)
```

其中，`file` 参数指定要打开的文件路径，`encoding` 参数指定编码格式，通常取值 `'utf-8'`，`mode` 参数指定文件的打开模式。常用的模式包括：

- `'r'`：以只读模式打开文件（默认值）。
- `'w'`：以写入模式打开文件，若文件已存在，则会覆盖其内容。
- `'a'`：以追加模式打开文件，数据将写入文件末尾。
- `'+'`：同时打开文件进行读写操作，与其他模式组合使用，如 `'r+'` 表示以读写模式打开文件。

以下是一个使用 `open()` 函数读取文件内容的示例：

```
# 打开文件 example.txt 进行读取
file = open('example.txt', 'r', encoding='utf-8')
content = file.read()
print(content)
file.close()
```

在上述代码中，`encoding='utf-8'` 参数指定文件的编码方式，以正确读取包含非 ASCII 字符的文件。`file.close()` 方法用于关闭文件对象，释放系统资源并确保数据完整性。

假设 `example.txt` 文件的内容如下：

```
Python 是一种广泛使用的高级编程语言。
它具有简洁的语法和强大的功能。
```

运行上述代码将输出文件的全部内容。

`open()` 函数还可用于写入文件。以下是一个写入文件的示例：

```
# 打开文件 example.txt 进行写入
file = open('example.txt', 'w', encoding='utf-8')
file.write('这是一个新的内容。')
file.close()
```

执行此代码后，`example.txt` 文件的内容将被替换为 `'这是一个新的内容。'`。

**注意：**使用 `'w'` 模式打开文件会覆盖原有内容，若需在文件末尾追加内容，应使用 `'a'` 模式。`file.close()` 方法确保写入的数据被刷新到磁盘，避免数据丢失。

在 Python 编程中，`with` 语句用于简化对资源的管理，特别是在文件操作中。使用 `with` 语句打开文件时，系统会在代码块执行完毕后自动关闭文件，无需显式调用 `close()` 方法，从而确保资源的正确释放并提高代码的可读性。

以下示例展示了如何使用 `with` 语句读取文件内容：

```
# 使用 with 语句打开文件 example.txt 进行读取
with open('example.txt', 'r', encoding='utf-8') as file:
    content = file.read()
    print(content)
```

在上述代码中，`with` 语句打开名为 `example.txt` 的文件，并将文件对象赋值给变量 `file`。在 `with` 代码块内，调用 `read()` 方法读取文件内容，并输出到控制台。当代码块执行完毕后，文件会被自动关闭。

以下示例展示了如何使用 `with` 语句将内容写入文件：

```
# 使用 with 语句打开文件
with open('example.txt', 'w', encoding='utf-8') as file:
    file.write('这是一个新的内容。')
# 无需显式调用 file.close()
```

推荐使用 `with` 语句读写文件。

文件的读取方法主要包括 `read()`、`readline()`、`readlines()` 和直接迭代文件对象。这些方法适用于不同的场景，选择合适的方法有助于提高代码的效率和可读性。

### 1. `read()` 方法：

`read()` 方法用于一次性读取整个文件的内容，并将其作为一个字符串返回。需要注意的是，若文件较大，使用 `read()` 可能导致内存占用过高。

```
# 打开文件 example.txt 进行读取
with open('example.txt', 'r', encoding='utf-8') as file:
    content = file.read()
    print(content)
```

## 2. `readline()` 方法:

`readline()` 方法用于读取文件的一行内容，包括行末的换行符。每次调用 `readline()` 都会返回文件的下一行，适用于逐行读取文件的场景。

```
# 打开文件 example.txt 逐行读取
with open('example.txt', 'r', encoding='utf-8') as file:
    line = file.readline()
    while line:
        print(line, end='') # end='' 防止重复添加换行
        line = file.readline()
```

运行上述代码将逐行输出文件内容。

### 3. `readlines()` 方法:

`readlines()` 方法用于读取文件的所有行，并将其作为一个列表返回，每个元素为文件中的一行。需要注意的是，若文件较大，使用 `readlines()` 可能导致内存占用过高。

```
# 打开文件 example.txt 读取所有行
with open('example.txt', 'r', encoding='utf-8') as file:
    lines = file.readlines()
    for line in lines:
        print(line, end='')
```

运行上述代码将输出文件的全部内容。

#### 4. 直接迭代文件对象：

文件对象本身是可迭代的，直接对文件对象进行迭代可逐行读取文件内容。这种方法内存占用较低，适用于大文件的读取。

```
# 打开文件 example.txt 直接迭代
with open('example.txt', 'r', encoding='utf-8') as file:
    for line in file:
        print(line, end='')
```

运行上述代码将逐行输出文件内容。

#### 注意

- 在读取文件时，建议使用 `with` 语句管理文件上下文。`with` 语句会在代码块执行完毕后自动关闭文件，确保资源的正确释放。

- 在读取包含非 ASCII 字符的文本文件时，需指定编码方式，如 `encoding='utf-8'`，以确保字符编码正确。

文件的读取模式决定了文件的打开方式和操作权限。常用的读取模式包括：

1. `'r'`（只读模式）：最常用。以只读方式打开文件，文件必须存在，指针位于文件开头。
2. `'rb'`（二进制只读模式）：以二进制格式只读方式打开文件，适用于非文本文件，如图片、音频等。
3. `'r+'`（读写模式）：以读写方式打开文件，文件必须存在，指针位于文件开头。
4. `'rb+'`（二进制读写模式）：以二进制格式读写方式打开文件，适用于需要读写非文本文件的情况。

以下示例展示了如何使用不同的读取模式读取文件内容。假设存在一个名为 `example.txt` 的文本文件，其内容如下：

Python 是一种广泛使用的高级编程语言。  
它具有简洁的语法和强大的功能。

- 使用 `'r'` 模式读取文件：

```
with open('example.txt', 'r', encoding='utf-8') as file:  
    content = file.read()  
    print(content)
```

- 使用 `'rb'` 模式读取文件：

```
with open('example.txt', 'rb') as file:  
    content = file.read()  
    print(content.decode('utf-8'))
```

- 使用 'r+' 模式读取并写入文件：

```
with open('example.txt', 'r+', encoding='utf-8') as file:  
    content = file.read()  
    print(content)  
    file.write('\n这是追加的内容。')
```

- 使用 'rb+' 模式读取并写入二进制文件：

```
with open('example.txt', 'rb+') as file:  
    content = file.read()  
    print(content.decode('utf-8'))  
    file.write('\n这是追加的内容.'.encode('utf-8'))
```

在上述示例中，`with` 语句用于确保文件在操作完成后自动关闭，`encoding='utf-8'` 参数指定了文件的编码方式，以正确处理包含非 ASCII 字符的内容。

文件的写入方法主要包括 `write()` 和 `writelines()`。这两种方法用于将数据写入文件，但适用场景有所不同。

### 1. `write()` 方法:

`write()` 方法用于将字符串写入文件。需要注意的是，`write()` 方法不会自动添加换行符，若需换行，需手动在字符串中加入 `\n`。

```
# 打开文件 example.txt 进行写入
with open('example.txt', 'w', encoding='utf-8') as file:
    file.write('这是第一行内容.\n')
    file.write('这是第二行内容.')
```

执行上述代码后，`example.txt` 文件的内容为:

```
这是第一行内容.
这是第二行内容.
```

## 2. writelines() 方法:

`writelines()` 方法用于将字符串列表写入文件。与 `write()` 方法类似, `writelines()` 不会自动添加换行符, 需在列表中的每个字符串末尾手动添加 `\n`。

```
# 打开文件 example.txt 进行写入
with open('example.txt', 'w', encoding='utf-8') as file:
    lines = ['这是第一行内容.\n', '这是第二行内容.\n',
            ↪ '这是第三行内容. ']
    file.writelines(lines)
```

执行上述代码后, `example.txt` 文件的内容为:

```
这是第一行内容.
这是第二行内容.
这是第三行内容.
```

### 3. `print()` 函数:

`print()` 函数不仅用于在控制台输出信息，还可用于将内容写入文本文件。通过指定 `print()` 函数的 `file` 参数，可以将输出重定向到文件对象，从而实现文件写入操作。

```
# 打开文件进行写入操作
with open('output.txt', 'w', encoding='utf-8') as file:
    # 使用 print() 函数将内容写入文件
    print("Hello, this is a sample text.", file=file)
    print("This text will be written to a file using the print
    ↪ function.", file=file)
```

**注意:** 在使用 `print()` 函数写入文件时，默认会在每次调用后添加换行符。如需避免换行，可设置 `end` 参数，

文件的写入模式决定了文件的打开方式和操作权限。常用的写入模式包括：

1. `'w'`（写入模式）：最常用。以写入方式打开文件，若文件已存在，则清空其内容；若文件不存在，则创建新文件。
2. `'a+'`（追加读写模式）：以追加和读写方式打开文件，若文件已存在，指针位于文件末尾，可读取和追加内容；若文件不存在，则创建新文件。
3. `'a'`（追加模式）：以追加方式打开文件，若文件已存在，指针位于文件末尾，新的内容将添加到现有内容之后；若文件不存在，则创建新文件。
4. `'w+'`（读写模式）：以读写方式打开文件，若文件已存在，则清空其内容；若文件不存在，则创建新文件。

以下示例展示了如何使用不同的写入模式操作文件。假设存在一个名为 `example.txt` 的文本文件，其初始内容如下：

Python 是一种广泛使用的高级编程语言。  
它具有简洁的语法和强大的功能。

- 使用 `'w'` 模式写入文件：

```
with open('example.txt', 'w', encoding='utf-8') as file:  
    file.write('这是使用 w 模式写入的新内容。')
```

执行上述代码后，`example.txt` 的内容将被替换为：

这是使用 `w` 模式写入的新内容。

- 使用 'a' 模式追加内容:

```
with open('example.txt', 'a', encoding='utf-8') as file:  
    file.write('\n这是使用 a 模式追加的内容。')
```

执行上述代码后, example.txt 的内容将变为:

这是使用 w 模式写入的新内容.  
这是使用 a 模式追加的内容.

- 使用 'w+' 模式读写文件:

```
with open('example.txt', 'w+', encoding='utf-8') as file:  
    file.write('这是使用 w+ 模式写入的新内容。')
```

执行上述代码后, example.txt 的内容为:

这是使用 w+ 模式写入的新内容。

- 使用 'a+' 模式追加并读取内容:

```
with open('example.txt', 'a+', encoding='utf-8') as file:  
    file.write('\n这是使用 a+ 模式追加的内容。')
```

执行上述代码后, example.txt 的内容为:

这是使用 w+ 模式写入的新内容。  
这是使用 a+ 模式追加的内容。

在上述示例中, with 语句用于确保文件在操作完成后自动关闭, encoding='utf-8' 参数指定了文件的编码方式, 以正确处理包含非 ASCII 字符的内容。

文件路径的指定方式主要分为相对路径和绝对路径。理解并正确使用这两种路径对于文件操作至关重要。

### 1. 相对路径：

相对路径是相对于当前工作目录（`current working directory, CWD`）指定的文件或目录位置。使用相对路径时，路径的起点是当前正在访问的文件夹。相对路径更灵活，易于在不同机器或文件夹结构间移植代码。但如果更改了工作目录，可能导致路径无效。

假设当前工作目录为 `/home/user/project`，目录结构如下：

```
/home/user/project/  
├─ main.ipynb  
└─ data/  
    └─ example.txt
```

在 `main.ipynb` 中，使用相对路径读取 `example.txt`：

```
with open('data/example.txt', 'r', encoding='utf-8') as file:  
    content = file.read()  
    print(content)
```

此时，`data/example.txt` 即为相对路径。

**注意：**在相对路径中 `.` 表述当前文件夹，`..` 表示上一层文件夹。

## 2. 绝对路径：

绝对路径是从文件系统的根目录开始的完整路径，提供了到达指定文件或目录的完整地址。绝对路径明确无误地指向文件位置，不受当前工作目录的影响。但不够灵活，当文件系统结构变化或在不同系统之间迁移代码时可能需要修改。

基于上述目录结构，使用绝对路径读取 `example.txt`：

```
with open('/home/user/project/data/example.txt', 'r',  
→ encoding='utf-8') as file:  
    content = file.read()  
    print(content)
```

`/home/user/project/data/example.txt` 即为绝对路径。

获取当前工作目录：使用 `os` 模块的 `getcwd()` 函数获取当前工作目录：

```
import os

current_directory = os.getcwd()
print(f"当前工作目录: {current_directory}") # 当前工作目录:
↪ /home/user/project
```

将相对路径转换为绝对路径：使用 `os.path` 模块的 `abspath()` 函数将相对路径转换为绝对路径：

```
import os

relative_path = 'data/example.txt'
absolute_path = os.path.abspath(relative_path)
print(f"绝对路径: {absolute_path}") # 绝对路径:
↪ /home/user/project/data/example.txt
```

在 Python 中，`seek()` 和 `tell()` 方法用于控制文件指针的位置，从而实现对文件的随机读写操作。`seek()` 方法用于移动文件指针到指定位置，`tell()` 方法用于获取当前文件指针的位置。

示例文件内容：

假设有一个名为 `example.txt` 的文本文件，内容如下：

```
Hello, this is a sample text file.  
It contains multiple lines of text.  
Each line serves as an example.
```

使用 `seek()` 和 `tell()` 进行文件操作的示例代码：

```
# 打开文件进行读写操作
file = open('example.txt', 'r+')
# 读取并打印第一行
first_line = file.readline()
print(f" 第一行内容: {first_line.strip()}")

# 获取当前文件指针位置
current_position = file.tell()
print(f" 当前文件指针位置: {current_position}")
```

代码解析：

1. 使用 `with open('example.txt', 'r+')` 打开文件，模式为 `'r+'`，表示以读写模式打开文件。
2. 使用 `readline()` 方法读取并打印文件的第一行内容。
3. 使用 `tell()` 方法获取当前文件指针的位置，并打印该位置。

```
# 移动文件指针到文件开头
file.seek(0)
print(f" 文件指针已移动到位置: {file.tell()}")

# 在文件开头插入新内容
new_content = "Inserted line at the beginning.\n"
original_content = file.read()
file.seek(0)
file.write(new_content + original_content)
```

4. 使用 `seek(0)` 将文件指针移动到文件开头，并验证指针位置。
5. 在文件开头插入新内容。为此，先读取原始内容，移动指针到开头，然后将新内容和原始内容写入文件。

```
# 移动文件指针到文件末尾
file.seek(0, 2)
print(f" 文件指针已移动到文件末尾位置: {file.tell()}")

# 在文件末尾追加新内容
file.write("\nAppended line at the end.")

# 关闭文件
file.close()
```

6. 使用 `seek(0, 2)` 将文件指针移动到文件末尾，`0` 表示偏移量，`2` 表示从文件末尾开始计算。
7. 在文件末尾追加新内容。
8. 关闭文件。

## 注意

- 使用 `seek()` 方法时，第二个参数 `whence` 的取值：`0` 表示从文件开头计算（默认值），`1` 表示从当前位置计算，`2` 表示从文件末尾计算。

- 在文本模式下（如 `'r+'`），`seek()` 和 `tell()` 的偏移量和位置是以字节 (byte) 为单位的。

**异常 (Exception)** 是指在程序执行过程中出现的错误或意外情况，导致程序无法按照预期继续运行。异常处理机制使程序能够捕获并处理这些错误，确保程序的稳健性和可靠性。

```
try:
    numerator = 10
    denominator = 0
    result = numerator / denominator
    print(result)
except ZeroDivisionError:
    print(" 错误：除数不能为零。")
```

在上述代码中，`try` 块包含可能引发异常的代码。当执行到 `result = numerator / denominator` 时，由于 `denominator` 为零，会引发 `ZeroDivisionError` 异常。此时，程序跳转到对应的 `except` 块，输出提示信息“错误：除数不能为零。”。通过这种方式，程序避免了因未处理的异常而崩溃。

异常传播机制指的是当异常在当前作用域未被捕获时，沿调用栈向上传递，直到被捕获或导致程序终止的过程。这种机制确保异常信息能够传递给适当的处理程序，以便采取相应的措施。

以下示例演示了异常的传播过程：

```
def function_a():  
    function_b()  
  
def function_b():  
    function_c()  
  
def function_c():  
    raise ValueError(" 发生了一个值错误")  
  
try:  
    function_a()  
except ValueError as e:  
    print(f" 捕获到异常: {e}")
```

在上述代码中，`function_c` 中显式引发了 `ValueError` 异常。由于 `function_c` 内部没有处理该异常，异常被传播到调用它的 `function_b`。同样地，`function_b` 也未处理该异常，异常继续传播到 `function_a`。最终，异常传播到 `function_a` 的调用者，即 `try` 块。在此处，存在匹配的 `except` 子句来捕获 `ValueError` 异常，因此异常被成功捕获，程序输出相应的提示信息。

异常处理是确保程序稳健性和可靠性的重要机制。Python 提供了丰富的内置异常类，用于捕获和处理各种错误情况。这些异常类均继承自 `BaseException`，并形成层次化的结构，方便开发者根据具体需求进行捕获和处理。以下是一些常见的 Python 内置异常类及其使用示例：

## 1. ZeroDivisionError

当尝试除以零时引发。

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print(" 错误：除数不能为零。")
```

## 2. ValueError

当函数接收到参数类型正确但值不合适时引发。

```
try:  
    number = int("abc")  
except ValueError:  
    print(" 错误：无法将字符串转换为整数。")
```

## 3. TypeError

当操作或函数应用于不适当类型的对象时引发。

```
try:  
    result = '2' + 2  
except TypeError:  
    print(" 错误：不能将字符串与整数相加。")
```

## 4. IndexError

当尝试访问序列中不存在的索引时引发。

```
try:
    numbers = [1, 2, 3]
    print(numbers[5])
except IndexError:
    print(" 错误: 索引超出列表范围。")
```

## 5. KeyError

当在字典中使用一个不存在的键时引发。

```
try:
    data = {'name': 'Alice'}
    print(data['age'])
except KeyError:
    print(" 错误: 键不存在于字典中。")
```

## 6. FileNotFoundError

当尝试打开一个不存在的文件时引发。

```
try:
    with open('nonexistent_file.txt', 'r') as file:
        content = file.read()
except FileNotFoundError:
    print(" 错误: 文件未找到。")
```

## 7. AttributeError

当尝试访问对象中不存在的属性时引发。

```
try:
    obj = None
    obj.method()
except AttributeError:
    print(" 错误: 对象没有该属性或方法。")
```

## 8. ImportError

当导入模块失败时引发。

```
try:
    import nonexistent_module
except ImportError:
    print(" 错误: 模块导入失败。")
```

了解并正确处理这些内置异常，有助于编写健壮的 Python 程序，确保在各种错误情况下程序能够优雅地处理并继续运行。表 11.1 列出了常见的 Python 内置异常类及其含义。

在 Python 编程中，异常处理机制通过 `try`、`except`、`else`、`finally` 和 `raise` 语句来管理程序运行时的错误。

- `raise` 语句允许程序员在特定条件下主动引发异常，以便在检测到错误或异常情况时中断正常的程序流程，并将控制权交给相应的异常处理器。
- `try` 块包含可能引发异常的代码；
- `except` 块用于捕获并处理这些异常；
- `else` 块在未发生异常时执行；
- `finally` 块无论是否发生异常都会执行；

## 1. raise 语句

`raise` 语句用于显式引发异常，以便在程序运行过程中处理特定的错误或异常情况。通过使用 `raise`，可以在代码中指定何时以及为何引发异常，从而提高程序的健壮性和可维护性。

### (1) 引发指定的异常：

```
raise Exception(" 这是一个自定义的异常消息")
```

上述代码将引发一个通用的 `Exception` 异常，并附带自定义的错误消息。

### (2) 引发特定类型的异常：

```
raise ValueError(" 无效的输入")
```

此示例引发一个 `ValueError` 异常，通常用于表示传入函数的参数具有有效类型但不在期望的值范围内。

## 2. try/except 语句

`try/except` 语句用于捕获和处理程序运行时可能发生的异常，确保程序的稳健性和可靠性。其基本结构如下：

```
try:  
    # 可能引发异常的代码  
except 异常类型:  
    # 处理异常的代码
```

在 `try` 块中编写可能引发异常的代码；如果发生指定类型的异常，程序将跳转到对应的 `except` 块执行处理代码。

以下示例演示了如何使用 `try/except` 语句处理除零错误:

```
def divide_numbers(a, b):  
    try:  
        result = a / b  
        return result  
    except ZeroDivisionError:  
        print(" 错误: 除数不能为零。")  
        return None  
  
# 测试代码  
num1 = 10  
num2 = 0  
output = divide_numbers(num1, num2)  
if output is not None:  
    print(f" 结果: {output}")
```

### 3. else 子句

`else` 子句紧跟在所有 `except` 子句之后，仅在 `try` 块未引发任何异常时执行。这对于将正常执行路径与异常处理逻辑分开非常有用。

```
try:
    result = 10 / 2
except ZeroDivisionError:
    print(" 除数不能为零。")
else:
    print(f" 结果是: {result}")
```

在上述代码中，`try` 块成功执行除法操作，未引发任何异常，因此执行 `else` 子句，输出计算结果。

## 4. finally 子句

`finally` 子句无论是否发生异常，都会执行，通常用于释放资源或执行清理操作。

```
try:
    file = open('example.txt', 'r')
    content = file.read()
except FileNotFoundError:
    print(" 文件未找到。")
else:
    print(content)
finally:
    file.close()
    print(" 文件已关闭。")
```

在上述代码中，`try` 块尝试打开并读取文件内容。如果文件存在且读取成功，执行 `else` 子句，打印文件内容。无论是否发生异常，`finally` 子句都会执行，确保文件被关闭。

`try/except` 语句可以捕获和处理程序运行时可能发生的异常。当需要捕获多个异常时，可以在单个 `except` 块中指定多个异常类型，或为每种异常类型定义独立的 `except` 块。

### 1. 在单个 `except` 块中捕获多个异常

可以在一个 `except` 块中通过元组指定多个异常类型。当 `try` 块中的代码引发这些异常之一时，程序将执行该 `except` 块。

```
try:
    # 可能引发异常的代码
except (TypeError, ValueError) as e:
    print(f" 捕获到异常: {e}")
```

以下示例演示了如何在单个 `except` 块中捕获 `TypeError` 和 `ValueError` 异常：

```
def process_data(data):  
    try:  
        # 尝试将数据转换为整数  
        number = int(data)  
        # 执行除法操作  
        result = 10 / number  
    except (ValueError, ZeroDivisionError) as e:  
        print(f" 处理数据时发生错误: {e}")  
    else:  
        print(f" 结果是: {result}")  
  
# 测试代码  
process_data("abc") # 将引发 ValueError  
process_data("0") # 将引发 ZeroDivisionError  
process_data("5") # 正常处理
```

## 2. 在多个 `except` 块中捕获不同的异常

如果需要对不同类型的异常执行不同的处理操作，可以为每种异常类型定义独立的 `except` 块。

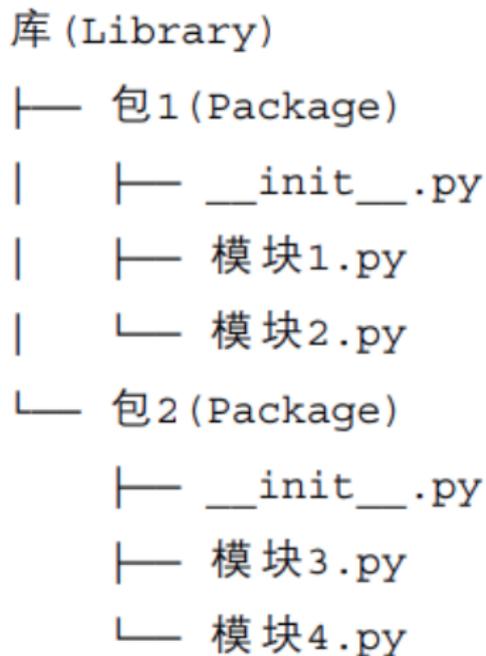
```
try:
    # 可能引发异常的代码
except TypeError as e:
    print(f" 捕获到 TypeError: {e}")
except ValueError as e:
    print(f" 捕获到 ValueError: {e}")
```

以下示例演示了如何为 `TypeError` 和 `ValueError` 定义独立的 `except` 块：

```
def add_numbers(a, b):  
    try:  
        result = a + b  
    except TypeError as e:  
        print(f" 类型错误: {e}")  
    else:  
        print(f" 结果是: {result}")  
  
# 测试代码  
add_numbers(5, "10") # 将引发 TypeError  
add_numbers(5, 10)  # 正常处理
```

在上述代码中，`add_numbers` 函数尝试将两个参数相加。如果参数类型不匹配，将引发 `TypeError`。通过为 `TypeError` 定义独立的 `except` 块，可以针对该异常类型执行特定的处理操作。

在 Python 中，代码组织的基本概念包括模块（Module）、包（Package）和库（Library）。这三个层次的代码组织方式形成了一个层次化的结构：



**模块 (Module):** 模块是一个包含 Python 定义和语句的文件，通常以 `.py` 作为扩展名。模块用于将相关的代码组织在一起，便于重用和维护。例如，创建一个名为 `math_operations.py` 的模块，内容如下：

```
# math_operations.py

def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

在其他脚本中，可以通过 `import` 语句导入该模块并使用其中的函数：

```
import math_operations

result = math_operations.add(5, 3)
print(result) # 输出: 8
```

Python 提供了大量的内置模块，可以通过 `import` 语句直接导入。例如，使用 `math` 模块计算平方根：

```
import math

result = math.sqrt(16)
print(result) # 输出: 4.0
```

**包 (Package):** 包是一个包含多个模块的目录，用于组织相关的模块。在 Python 3.3 之前，包目录中必须包含一个 `__init__.py` 文件，以表示该目录是一个包。在 Python 3.3 及之后的版本中，`__init__.py` 文件变为可选，但通常仍会包含该文件以明确表示包的存在。例如，创建一个名为 `mypackage` 的包，结构如下：

```
mypackage/  
  __init__.py  
  module1.py  
  module2.py
```

在其他脚本中，可以导入包中的模块：

```
from mypackage import module1
```

Python 自带了一些包，例如 `os` 包，其包含与操作系统交互的多个模块。可以直接导入并使用其功能，例如列出当前目录下的文件：

```
import os

files = os.listdir('.')
print(files)
```

**库 (Library)**: 库是一个包含多个包和模块的集合, 提供特定的功能或解决方案。例如, **NumPy** 是一个用于科学计算的库, 包含多个子包和模块, 提供对大型多维数组和矩阵的支持, 以及大量的数学函数。以下是一个典型 Python 库的目录结构示例:

```
numpy/  
  __init__.py  
  core/  
    __init__.py  
    multiarray.py  
    umath.py  
  linalg/  
    __init__.py  
    lapack_lite.py  
  fft/  
    __init__.py  
    fftpack.py
```

该结构展示了库的分层设计, 每个子包提供了特定功能, 例如, **core** 提供核心数组操作, **linalg** 提供线性代数功能, 而 **fft** 处理快速傅里叶变换。

Python 标准库是随 Python 解释器一同发布的模块集合，提供了丰富的功能，涵盖文件操作、系统交互、网络通信、数据处理等多个方面。利用标准库，开发者无需编写大量基础代码，即可实现复杂的功能，从而提高开发效率和代码质量。

讲义表 12.1 列出了常用的 Python 标准库模块及其基本功能和应用场景。

**random 模块：** `random` 模块用于生成随机数，支持各种分布的随机数生成，以及从序列中随机选择元素等功能。

(1) 生成 0 到 1 之间的随机浮点数：

```
import random

# 生成随机浮点数
random_number = random.random()
print(random_number)
```

## (2) 生成指定范围内的随机整数:

```
import random

# 生成 1 到 10 之间的随机整数
random_integer = random.randint(1, 10)
print(random_integer)
```

## (3) 从列表中随机选择元素:

```
import random

# 定义列表
items = ['apple', 'banana', 'cherry']

# 随机选择一个元素
random_item = random.choice(items)
print(random_item)
```

## 1. 编写自定义模块:

自定义模块本质上是一个包含 Python 代码的文件，通常以 `.py` 作为扩展名。在该文件中，可以定义函数、类和变量等。例如，创建一个名为 `string_utils.py` 的文件，内容如下：

```
# string_utils.py

def to_uppercase(s):
    """ 将字符串转换为大写 """
    return s.upper()

def count_vowels(s):
    """ 统计字符串中元音字母的数量 """
    vowels = 'aeiouAEIOU'
    return sum(1 for char in s if char in vowels)
```

在其他 Python 脚本中，可以通过 `import` 语句导入该模块并使用其中的函数：

```
import string_utils

text = "Hello World"
uppercase_text = string_utils.to_uppercase(text)
vowel_count = string_utils.count_vowels(text)

print(uppercase_text) # 输出: HELLO WORLD
print(vowel_count) # 输出: 3
```

## 2. 模块文件的命名和存储位置对模块导入的影响:

**命名规范:** 根据 PEP 8 的建议, 模块名称应简短且全部使用小写字母, 必要时可使用下划线以提高可读性。

**存储位置:** Python 解释器通过 `sys.path` 列表中的目录来搜索模块。因此, 模块文件应存放在这些目录中, 或者与导入它的脚本位于同一目录。

## 3. 模块功能探索

在 Python 编程中, 了解模块的内容和功能对于高效开发至关重要。可以使用内置的 `dir()` 和 `help()` 函数, 以及 `__doc__` 属性, 快速获取模块、函数或类的相关信息。

- 使用 `dir()` 函数获取模块的属性和方法列表;
- 使用 `help()` 函数获取模块、函数或类的详细信息;
- 使用 `__doc__` 属性查看文档字符串;

#### 4. 导入包与模块中的内容

在 Python 中，导入包和模块中的函数等内容是组织代码、提高可读性和重用性的关键。

**(1) 导入整个模块：**使用 `import` 语句导入模块后，需通过 `模块名.对象名` 的方式访问其中的函数或变量。例如：

```
import math
print(math.pi) # 输出: 3.141592653589793
```

此方式将整个 `math` 模块导入，访问其内容时需使用 `math` 作为前缀。

**(2) 从模块中导入特定函数或变量：**使用 `from...import...` 语句直接导入模块中的特定对象，可直接使用其名称，无需模块前缀。例如：

```
from math import pi
print(pi) # 输出: 3.141592653589793
```

**(3) 为导入的模块或函数指定别名：**使用 `as` 关键字为导入的模块或函数指定别名，简化代码书写或避免命名冲突。例如：

```
import numpy as np
from pandas import DataFrame as DF
```

此方式将 `numpy` 模块重命名为 `np`，`DataFrame` 类重命名为 `DF`，提高代码可读性。

**(4) 从包的模块中导入特定函数或变量：**使用 `from...import...` 语句从包的特定模块中导入所需对象。例如：

```
from my_package.sub_module import some_function

# 直接调用导入的函数
result = some_function()
```

此方式仅导入 `some_function`，调用时无需指定模块前缀。

在 Python 编程中，第三方库是由外部开发者创建的代码集合，旨在提供特定的功能或解决方案，供其他项目复用。这些库通常被打包并发布在 Python 包索引（PyPI）上，开发者可以使用包管理工具如 `pip` 进行安装和管理。通过利用第三方库，开发者能够避免重复编写常见或复杂的代码，从而节省时间和精力。

`pip` 是官方推荐的包管理工具，用于安装和管理第三方库。通过 `pip`，可以从 Python 包索引（PyPI）以及其他索引安装软件包。

## 1. 安装和升级 pip :

在大多数 Python 发行版中，`pip` 已经预装。可以通过以下命令检查 `pip` 是否已安装：

```
pip --version
```

如果未安装 `pip`，可以使用以下命令进行安装：

```
python -m ensurepip --default-pip
```

为了确保使用最新版本的 `pip`，建议运行以下命令进行升级：

```
python -m pip install --upgrade pip
```

## 2. 安装第三方库：

使用 `pip` 安装第三方库非常简便。例如，安装名为 `requests` 的库，可以执行以下命令：

```
pip install requests
```

此命令会从 PyPI 下载并安装 `requests` 及其所有依赖项。

### 3. 卸载库：

如果需要卸载已安装的库，可以使用以下命令：

```
pip uninstall requests
```

此命令会从环境中移除 `requests` 库。

### 4. 列出已安装的库：

要查看当前环境中已安装的所有库，可以运行：

```
pip list
```

此命令会显示已安装库的列表及其版本号。

## 5. 使用 `requirements.txt` 管理依赖:

在项目开发中, 通常会使用 `requirements.txt` 文件来记录项目的所有依赖库。可以通过以下命令生成该文件:

```
pip freeze > requirements.txt
```

此命令会将当前环境中的所有库及其版本信息写入 `requirements.txt`。在新的环境中, 可以使用以下命令根据该文件安装所有依赖:

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

## 6. 使用虚拟环境:

为了避免不同项目之间的依赖冲突, 建议为每个项目创建独立的虚拟环境。可以使用 `venv` 模块创建虚拟环境:

```
python -m venv myenv
```

激活虚拟环境后, 使用 `pip` 安装的库将仅作用于该环境, 确保项目的依赖独立性。

寻找第三方库的主要途径包括：

- **Python 包索引 (PyPI)**: PyPI 是官方的第三方软件仓库，提供了超过 50 万个 Python 包，涵盖从数据分析到网络开发等各个领域。
- **GitHub**: 作为全球最大的代码托管平台，GitHub 上托管了大量的 Python 项目和库。开发者可以搜索特定功能的库，并查看其源代码、文档和更新情况。
- **Awesome Python 列表**: 这是一个由社区维护的精选 Python 库和工具的列表，涵盖了不同的应用领域，帮助开发者快速找到高质量的第三方库。
- **官方文档和社区论坛**: Python 的官方文档和社区论坛（如，Stack Overflow）也是获取第三方库信息的重要来源。在这些平台上，开发者可以找到推荐的库、使用示例以及其他开发者的经验分享。

THE END