Python
金融量化分析

肖建军 高拴平◎编著

清華大學出版社
北 京

内 容 简 介

金融量化分析不仅需要掌握金融领域的知识，还需要掌握相关的计算机编程技术。本书全面、系统地介绍金融量化分析所需要掌握的技能。无论是具有丰富的编程经验的读者，还是普通的投资爱好者，均可参照本书内容开发自己的量化交易策略回测代码，实现金融量化分析辅助投资的目的。

本书共 9 章，涵盖的主要内容有金融量化交易策略分析概述，Python 的基础语法，Pandas 模块基础，NumPy 基础，数据获取与清洗，金融量化交易策略实战，TA-Lib、Empyrical 与 Mplfinance 模块的使用方法，金融数据回归分析，ARIMA 与 VAR 模型在金融量化领域的应用，开源金融量化交易策略回测框架 Backtrader 的使用方法等。掌握这些内容，可以解决金融量化分析涉及的编程语言基础、数据获取、量化交易策略构建、统计学与金融学理论在金融量化领域的高级应用，以及现有的量化回测框架的使用方法等实际问题。

本书内容丰富，体系完整，讲解细致入微，既适合 Python 金融量化分析入门人员阅读，也适合有志从事量化投资工作的各类研究人员和从业人员阅读与参考，还适合作为高等院校金融和投资类相关专业的教材。

图书在版编目（CIP）数据

Python 金融量化分析 / 肖建军，高拴平编著. —北京：清华大学出版社，2024.4
ISBN 978-7-302-65998-3

Ⅰ. ①P… Ⅱ. ①肖… ②高… Ⅲ. ①软件工具－程序设计－应用－金融－量化分析 Ⅳ. ①F830.9-39

中国国家版本馆 CIP 数据核字（2024）第 068322 号

责任编辑：王中英
封面设计：欧振旭
责任校对：徐俊伟
责任印制：刘海龙

出版发行：清华大学出版社
网　　址：https://www.tup.com.cn，https://www.wqxuetang.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-83470000　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：小森印刷霸州有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：17.75　　**字　　数**：424 千字
版　　次：2024 年 4 月第 1 版　　**印　　次**：2024 年 4 月第 1 次印刷
定　　价：79.80 元

产品编号：100835-01

前言

写作背景

随着计算机技术的普及，越来越多的金融投资者希望借助计算机程序辅助投资。这样投资者不仅需要理解投资理论，还需要熟悉计算机编程技术。为此，本书希望能带领读者学习以下内容：

- 金融量化分析需要掌握的 Python 基础语法及相关模块；
- 免费获取金融数据的方法与途径；
- 金融量化交易策略的基础回测方法；
- 经典经济模型与金融理论在金融量化分析中的实际应用方法；
- 利用金融量化交易策略开源回测框架进行专业回测的方法。

通过认真学习本书内容，读者完全可以独立编写金融量化投资策略回测代码，还可以通过 Python 代码验证量化投资策略的历史表现，为当前和未来的投资决策提供参考依据。

本书特色

1．内容全面，涵盖广泛

本书不仅涵盖金融数据获取、数据清洗、量化交易策略回测代码的编写，而且也涵盖统计学与金融学的基础理论在量化投资策略构建中的高级应用，以及免费开源的专业金融量化回测框架 Backtrader 的使用方法。

2．代码实现与专业知识并重

金融量化分析不仅需要编程技术，而且需要对金融投资理论具有一定的理解，二者缺一不可。本书在重点介绍 Python 代码实现的同时，对必要的金融和统计学理论等专业知识进行简要描述，以便读者更深入地理解代码的含义与用途。

3．示例丰富，注重实战效果

本书穿插大量的示例进行讲解，每个示例均力求贴近实战需求。特别是在第 6 章金融量化交易策略开发实战会详细介绍 5 个常用的量化交易策略示例，通过这些示例的历史数据回测结果，显示策略的历史表现，将其作为开发实战策略的依据。

4．提供完整的源代码

笔者对书中涉及的所有源代码都进行了整理，以方便读者使用。读者对这些代码稍加修改，即可用于自己的项目实践中。

本书内容

第 1 章金融量化交易策略分析概述，详细介绍金融量化分析的基本流程、分析方法与工具、金融量化分析的优势、金融量化分析面临的困局与金融分析的注意事项等。

第 2 章金融量化分析工具的准备——基础语法，主要介绍 Python 的基础语法、变量、流程控制、函数、类与对象，以及模块应用等知识。

第 3 章金融量化分析工具的准备——Pandas 基础，围绕 Pandas 内部的 Series 和 DataFrame 对象，详细介绍如何利用 Pandas 处理二维数据表信息，如二维数据表文件的读取，行与列的定位、添加等。

第 4 章金融量化分析工具的准备——NumPy 基础，重点介绍 ndarray 对象的创建和 ndarray 数组数据的访问，以及 NumPy 数组操作、NumPy 模块的主要函数和 NumPy 随机数处理等。

第 5 章金融量化分析数据的准备，重点介绍如何通过 Tushare、Akshare、qstock 和 Alpha Vantage API 等方式获取与分析数据，以及如何通过 Pandas 进行数据清洗，并通过 CSV 文件或 SQLite 数据库存储数据等。

第 6 章金融量化交易策略开发实战，详细介绍趋势追踪交易策略、顶底背离交易策略、小市值交易策略、海龟交易策略与网格交易策略的历史回测代码和回测结果等。

第 7 章金融量化分析常用的工具模块，详细介绍金融量化分析中常用的技术分析指标，如 TA-Lib 模块、金融统计结果计算模块 Empyrical 与金融可视化专业模块 Mplfinance 等。

第 8 章金融量化分析高级应用，详细介绍统计学模型与现代金融理论在金融量化交易中的应用，包括基本的回归模型在金融量化分析中的应用、ARIMA 与 VAR 模型在金融量化分析中的应用，以及金融资产组合优化理论在金融量化分析中的具体应用等。

第 9 章金融量化回测框架 Backtrader 实战应用，详细介绍开源量化交易策略回测框架 Backtrader 的数据读取、自定义指标、指标调用、自定义策略类，以及观察器与分析器的应用等。

读者对象

阅读本书需要读者具备一定的金融投资基础知识，以及对程序设计有基本的理解能力，建议读者最好对 Python 编程语言有基本的了解。具体而言，本书主要适合以下读者阅读：

- 有一定 Python 编程基础的量化分析初学者；
- 了解量化投资的基础理论而希望开发量化投资策略的投资者；

- 已掌握 Python 基础语法而希望寻求 Python 应用场景的程序员；
- 高校金融、大数据和投资等专业的学生与老师。

阅读建议

- 基础相对薄弱的读者，建议从第 1 章开始顺次阅读本书；
- 具备 Python 语言基础的读者，可以从第 2 章开始阅读本书；
- 对金融数据获取感兴趣的读者，可以直接阅读第 5 章；
- 对金融数据可视化感兴趣的读者，可以直接阅读 7.3 节；
- 对交易策略实战感兴趣的读者，可以直接阅读第 6 章；
- 对 Backtrader 框架感兴趣的读者，可以直接阅读第 9 章。

本书配套资源

本书涉及的数据文件、源文件和教学 PPT 需要读者自行下载。请在清华大学出版社网站（www.tup.com.cn）上搜索本书，然后在本书页面上的“资源下载”模块中单击“网络资源”或“课件下载”按钮即可下载；读者也可以关注微信公众号“方大卓越”，然后回复数字“3”获取下载网址。

阅读反馈

限于笔者水平，书中可能还存在一些疏漏，敬请广大读者指正，笔者会及时进行勘误，并将勘误内容传至出版社网站上供读者下载。读者在阅读本书的过程中如果有疑问，可以发电子邮件到 bookservice2008@163.com 获得帮助。

致谢

感谢恩师连平先生、陈伟利先生和郑乡明老师多年来对我的培养与关心！
感谢欧振旭在本书出版过程中给予笔者的大力支持与帮助！
感谢笔者的家人给予笔者的理解与支持！

肖建军
2024 年 3 月

目录

第 1 章　金融量化交易策略分析概述

金融市场资产价格瞬息万变，无论专业投资机构还是个人投资者，都希望借助现代计算工具与手段辅助投资行为，以期获得高于市场平均收益的投资效果。这些计算工具与手段包括易懂、易用的 Python 编程语言，功能强大、使用便捷的 Python 计算工具模块（如 Pandas、NumPy、Backtrader 等）。通过学习与使用这些计算工具，完全可以验证各类投资理论，探索更为有效的交易策略，从而提升自我对金融市场的认知水平，最终提升交易胜算概率。

本书由浅入深，引领零基础读者快速自主开发量化交易策略，也对有计量经济学基础的读者提供开发基于一般回归模型、ARIMA 模型、VAR 模型的交易策略方案。为了便于读者开发完全自主独立的量化交易策略，本书还详细介绍开源的专业量化策略回测框架平台 Backtrader 的使用。通过对这些内容的逐步深入学习，读者完全可以独立开发各类量化交易策略。

本章的学习目标：

- ❑ 了解金融量化分析的基本流程；
- ❑ 了解金融量化分析的优势与困局。

1.1　金融量化分析简介

金融量化分析是指利用数学、统计学和计算机科学等工具，对金融市场进行定量分析和预测的方法。它的基本思想是通过收集大量历史数据，建立数学模型来描述金融市场的运作规律，并利用这些模型来进行交易决策和风险管理。

1.1.1　金融量化分析的应用范畴

金融量化分析应用广泛，包括股票、债券、期货、外汇等市场的分析和交易，以及投资组合管理、风险控制等方面。

- ❑ 资产定价：量化分析可以通过构建数学模型和算法来估计资产的价值和风险，如股票、债券、期货、期权等金融产品的定价。
- ❑ 投资组合管理：通过量化分析可以帮助投资者构建和优化的投资组合，以达到最小化风险、最大化收益或达到某个特定的目标。
- ❑ 风险管理：量化分析可以帮助金融机构评估和管理风险，包括市场风险、信用风

险和操作风险等。

- 交易策略和算法交易：量化分析可以帮助交易员设计和实现有效的交易策略和算法交易，以取得最大化交易效益。
- 高频交易：通过应用量化分析可以帮助交易员捕捉瞬时的市场机会，并在毫秒级别的时间内进行高速交易。

总之，金融量化分析的应用范畴非常广泛，它已经成为现代金融领域不可或缺的一部分。

1.1.2 金融量化分析的基本流程

进行金融量化分析，需要遵循一定的基本流程，这些流程为金融量化分析提供了程序保障。一般情况下，金融量化分析的基本流程包括以下几个步骤。

（1）问题定义，即明确研究的问题和目标，如资产定价、投资组合管理、风险管理等。

（2）数据获取，即收集和整理相关数据，包括金融市场的历史价格数据、经济指标数据、公司财务数据等。

（3）数据预处理，即对数据进行清洗、转换和缺失值处理等预处理操作，以保证数据的质量和完整性。

（4）特征工程，即从原始数据中提取有用的特征，并进行数据降维和特征选择等操作，以提高模型的精度和泛化能力。

（5）模型建立，即选择适当的算法和模型结构，并进行参数调整和优化等操作，以构建准确可靠的量化模型。

（6）模型验证，即对构建的模型进行验证和测试，以评估模型的性能和泛化能力。

（7）模型应用，即将构建好的模型应用到实际问题中，如资产定价、投资组合管理、风险管理等。

（8）模型监测，即对模型进行定期监测和维护，以保证模型的有效性和稳定性。

以上是金融量化分析的基本流程。在实际应用中，可根据具体问题和数据进行适当调整和修改。

1.1.3 金融量化分析的方法与工具

在金融量化分析的过程中，不仅需要具备一定的数学基础，还需要具备一定的计算工具与计算环境，二者缺一不可。其中，构建金融量化分析模型需要一定的数学基础，完成历史数据回测需要构建计算环境。目前，金融量化分析常用的方法与工具包括以下几种。

- 数学模型：包括时间序列模型、回归模型、机器学习模型等，用于对金融市场和金融产品进行定价、预测和分析。
- 统计分析：包括统计描述、假设检验、方差分析等，用于对金融数据进行分析和推断。

- 数据挖掘：包括分类、聚类、关联规则挖掘等，用于发现数据中的模式和规律，以提高模型的预测能力。
- 优化算法：包括线性规划、整数规划、动态规划等，用于求解投资组合和风险控制等问题。
- 人工智能：包括神经网络、深度学习、自然语言处理等，用于自动处理金融数据的分析和预测。
- 编程语言和软件工具：包括 Python、R、MATLAB、SAS 等编程语言，以及 Quantopian、QuantConnect、Alpha Vantage 等金融量化分析平台和软件工具。
- 数据库和云计算：包括 MySQL、MongoDB、AWS、Azure 等数据库和云计算平台，以便存储和处理大量的金融数据。

以上是一些常用的金融量化分析方法和工具。在实际应用中，可根据具体问题和数据进行选择和应用。

1.1.4　金融量化分析的优势

与传统的金融分析方法和金融交易方式相比，金融量化分析具有明显的优势，具体如下。

- 客观性：金融量化分析基于数据进行分析，能够有效地消除人为情绪和主观判断的干扰，从而使投资决策更加客观、理性和科学。
- 高效性：金融分析工作需要处理大量的数据，单纯依靠人工分析这些数据，耗时费力。通过基于计算机工具的量化分析可以自动处理海量数据，不仅能大大缩短分析时间，而且同时也能够降低人为误差。
- 可重复性：金融量化分析是基于数学模型和数据进行分析和预测的，可以在相同的条件下重复分析和验证结果，以提高分析结果的可信度。
- 量化风险控制：金融量化分析可以对市场风险进行有效的量化和控制，通过优化投资组合和风险管理策略，以提高投资收益并降低风险。
- 自动化交易：金融量化分析可以实现自动化交易，即根据分析结果自动调整投资组合和交易策略，以降低交易成本，提高投资效率。

综上所述，金融量化分析具有客观性、高效性、可重复性、量化风险控制和自动化交易等优势，能够为各类投资者、基金管理人和交易员等金融相关人员提供有力的决策依据。

1.2　金融量化分析的困局

尽管金融量化应用范围广、工具丰富、优势明显，但是在现实操作中金融量化分析也会面临一些困局。

1.2.1 金融量化分析策略的同质化

量化分析策略的同质化是指在市场上出现了大量相似或重复的量化策略，这些策略可能使用了相似的数据、模型、算法和交易规则，导致市场上的量化交易员和基金管理人员之间出现了激烈的竞争。

造成量化策略同质化的主要原因包括以下几个方面。

- 数据来源的同质化：随着金融数据的公开和普及，越来越多的交易员和基金管理人员都可以使用相同的数据源进行分析和预测，导致分析结果的相似性。
- 量化分析模型和算法的同质化：某些常见的模型和算法已经被广泛应用于量化分析中，如时间序列模型、回归模型、机器学习算法等，这些模型和算法可能会被多个人或机构使用。
- 计算工具的同质化：越来越多的投资者与投资机构在投资决策过程中开始使用诸如 Python、R 等编程语言进行数据分析和建模，计算工具的同质化导致计算过程与计算结果的同质化。

对于量化策略同质化问题，可以从以下几个维度克服。

- 选择多样化的数据来源：量化交易员和基金管理人员应该尽可能选择多样化的数据来源，以避免使用相同的数据进行分析和预测。
- 选择独特的模型和算法：量化交易员和基金管理人员可以尝试开发独特的模型和算法，以提高策略的独特性和差异性。
- 优化交易规则：在同质化的情况下，交易规则的优化和调整可以帮助交易员和基金管理人员获得更好的交易效果。

1.2.2 量化分析工具的局限性

尽管量化分析工具可以提高分析效率和准确度，但也存在一些局限性。以下是一些常见的局限性。

- 数据缺失或错误：量化分析依赖于大量的数据，但是数据缺失或错误可能会导致分析结果不准确。
- 过度拟合：量化分析使用的模型和算法可能会过度拟合历史数据，导致对未来数据的预测不准确。
- 黑天鹅事件：量化分析可能无法预测突发事件，如自然灾害和金融危机等，这些事件可能会对市场产生不可预测的影响。
- 模型偏差：量化分析使用的模型和算法可能会存在一定的偏差，从而导致分析结果的误差。
- 交易成本：量化分析通常需要频繁交易，但频繁交易可能会导致高交易成本，降低投资收益。
- 竞争激烈：随着越来越多的人开始使用量化分析工具，市场上的竞争也变得越来

越激烈，导致量化分析策略的同质化和交易效果的下降。

综上所述，量化分析工具的局限性包括数据缺失或错误、过度拟合、黑天鹅事件、模型偏差、交易成本高和竞争激烈等方面，需要量化分析从业人员谨慎使用并及时调整策略。

1.2.3　量化分析结果的随机性

量化分析结果的随机性是指，尽管使用相同的数据和模型进行分析，但在不同的时间和情况下结果可能会有所不同，这是由于金融市场的复杂性和随机性导致的。以下是一些常见的随机因素。

- 市场风险：金融市场的风险是无法完全预测和控制的，市场波动可能会导致分析结果出现偏差。
- 噪声因素：金融数据中存在大量的噪声，如数据的错误、不准确和缺失等，这些因素可能会影响分析结果的准确性。
- 模型偏差：量化分析所使用的模型和算法可能会存在一定的偏差，这些偏差可能会影响分析结果的稳定性和准确性。
- 竞争因素：量化分析策略在市场中的表现会受到其他投资者的影响，例如，大量资金涌入某个投资领域可能会导致市场行情的不稳定和结果的随机。

综上所述，量化分析结果的随机性是由市场风险、噪声因素、模型偏差和竞争因素等多种因素共同作用带来的。在进行量化分析时，需要充分考虑这些随机因素，并采取相应的措施来降低分析结果的随机性。例如，可以加入多种数据和模型来提高分析结果的准确性和稳定性。

1.3　金融量化分析注意事项

鉴于金融量化分析优势与困局并存，为了更有效地开发量化交易策略，需要注意以下事项。

- 确保数据质量：开发量化投资策略的基石是数据，没有数据就没有策略，数据质量对策略的表现有着决定性的影响。因此，需要收集尽可能多的数据，并确保数据的准确性、完整性和及时性。
- 梳理策略逻辑：量化投资策略的逻辑需要清晰、可靠、可复制。在开发策略时，需要考虑不同的市场环境、行情波动和投资风险，设计出不同情景下的交易策略。
- 优化策略算法：量化投资策略的核心是算法，需要根据策略的逻辑选择合适的算法。常用的算法包括均值回归、趋势跟踪、机器学习等。
- 完善风险控制机制：量化投资策略的风险控制非常重要。在策略中设置风险控制指标，如止损、止盈、仓位控制等；同时，需要定期评估策略的风险水平，及时调整风险控制指标，通过这些指标及时发现风险并采取应对措施，从而避免产生过度承担风险的行为操作。

- 系统分析回测结果：在开发完策略后，需要进行回测测试，模拟交易策略在历史数据上的表现。回测测试可以评估策略的有效性和稳定性，并优化策略的参数。在分析策略回测结果时，必须从收益性和风险性两个大的维度评估，判别策略的优劣，最终筛选出有一定水平正收益、交易频率适当（交易频率不能过低，以避免赌博行为）、胜算概率较高的策略进行实盘交易。

1.4 本 章 小 结

本章介绍了金融量化分析的基本概况、流程与主要的分析方法和工具。金融量化分析虽然具备客观性、全面性、有效性等优势，但也存在诸如量化分析策略同质化、量化分析工具有局限性、量化分析结果有随机性等劣势。但总体而言，金融量化分析依然是进行金融投资决策的基石，是金融投资技术的主流。

1.5 思 考 题

1．金融量化分析的基本流程是什么？
2．如何应对金融量化分析的困局？

第 2 章　金融量化分析工具的准备——基础语法

由于本书使用 Python 作为金融量化分析工具，因此在进行金融量化分析之前，本章先结合金融量化分析的相关知识，对 Python 语法和基本用法进行介绍，如变量、函数、逻辑判断、流程控制、类与对象等，从而为后续金融量化分析的学习打好基础。

说明：本章介绍的为 Python 的基础知识，适合零基础的读者学习。

本章的学习目标：

- 掌握 Python 变量的使用；
- 掌握 Python 判断、循环语言的使用方法；
- 掌握 Python 函数的使用方法；
- 掌握 Python 常用模块的安装与使用方法；
- 了解 Python 类的使用方法。

2.1　Python 简介

Python 是一种通用的解释型高级编程语言，它简单易学，语法结构可读性强，拥有大量的第三方库和框架，可以用于各种不同的应用场景，如 Web 开发、数据科学和人工智能等。在金融量化分析中，数据获取、数据处理和数据预测等领域均属于 Python 擅长的领域。

2.1.1　Python 数据处理的优势

进行金融量化分析的工具很多，如 MATLAB、R、SPSS、EViews 等，甚至使用 Excel 都可以作为金融量化分析的工具。本书之所以选择 Python 作为编程语言工具，主要是因为以下几个原因。

- Python 具有简单易学和可读性强的语法结构，这使得金融从业者能够快速上手。Python 代码通常比其他语言的代码更简洁，更易于理解和维护。
- Python 拥有丰富的第三方模块和工具，可以快速地进行数据分析、统计计算和机器学习等领域的开发。例如，NumPy、Pandas、Matplotlib 等模块能够帮助量化分

析人员进行数据处理和可视化，scikit-learn、TensorFlow 等框架能够支持机器学习和深度学习的相关应用。

- Python 是一种解释型语言，它能够快速运行和调试代码，可以节省编译的时间，避免烦琐的编译过程，这对迭代开发和快速的原型开发非常有帮助。
- Python 是开源语言，可以免费使用，并且其社区非常活跃，有很多开发者为其贡献代码和文档，支持更新和维护，因此，在开发过程中可以得到更多的帮助和支持。

基于以上优势，金融量化分析通常采用 Python 作为编程语言工具，大多数金融投资量化分析框架也是基于 Python 实现的。

2.1.2 Python 的基本语法

作为最流行的计算机语言之一，Python 具有独立风格的语法结构，在程序设计过程中必须遵循这些基本语法。

1．函数、变量、对象名称必须区分大小写

在定义变量和调用函数等过程中，Python 要求区分大小写，如 Buy 与 buy 是两个不同的独立函数。

2．标点符号必须使用英文字符输入

在 Python 编程中，冒号、逗号、分号、括号、引号等各种标点符号必须使用英文符号，如果使用中文输入这些标点符号会报错。

3．注释方法

为了增强代码的可读性，Python 允许使用以下两种方式进行注释。

- 在代码后添加符号“#”，例如，代码“import time # 引入 time 模块”只执行 import time，而“#”后面的内容不被 Python 执行。
- 使用一对三个单引号（'''）或一对三个双引号（""")，将引号之间的内容设定为注释。例如，在代码中输入“'''这是一个注释'''”，则三个单引号之间的“这是一个注释”被视为注释，不被执行。

4．行对齐与缩进

Python 要求代码靠左对齐，如果使用条件、循环、定义对象（函数、方法）等，则需要进行缩进。代码缩进方法是输入四个空格或一个 Tab 键（一般使用 Tab 键）。

5．允许在一行中写多条语句

Python 代码一般是一行只写一条语句。在特殊情况下，也允许一行写多条语句，这些语句之间需要用分号隔开。

6．允许一条语句写在多行

如果一条语句过长，为了提高代码的可读性，Python 允许在每行末尾添加斜杠“\”，表示下一行代码也是本行代码的一部分，即下一行代码与本行代码一起被执行。

2.2　Python 变量

在进行金融量化分析时，会涉及各类金融数据，读取、处理、分析和存储这些金融数据是金融量化分析的基本操作。这些金融数据一般分为浮点数、整数、字符串、列表、元组、字典等类型，本节重点介绍如何使用 Python 变量来存储和操作这些不同类型的金融数据。

2.2.1　变量的命名规则

变量的命名规则有利于提升 Python 代码的可读性，便于后期理解与维护；另外，Python 语法对变量定义也设置了一些规则，不符合语法的变量定义会出现错误提示。因此，在定义 Python 变量时，既要遵循 Python 的语法规定，也要体现 Python 的语言风格，以便增加代码的可读性。Python 变量的命名规则主要包括以下几种。

- 变量名称可以由字母、数字及下画线组成，但数字不能作为变量名的开头。例如，“价格_1”“price_1”“Price_1”“_Price_1”均符合 Python 变量的命名规则，而“1_Price”则不符合 Python 变量的命名规则。
- Python 自带的关键字不能作为变量名。Python 3 中有 35 个关键字，如表 2.1 所示。

表 2.1　Python 3 中的关键字

and	continue	FALSE	Is	raise
as	def	for	lambda	return
assert	del	from	nonlocal	try
async	elif	global	None	TRUE
await	else	if	not	while
break	except	import	or	with
class	finally	in	pass	yield

- Python 自带的内置函数名不能作为变量名。Python 3 自带 154 个内置函数（可以使用命令“dir (_ _ builtins _ _)”查询）。常见的内置函数如表 2.2 所示。

表 2.2　Python 3 常见的内置函数

abs	ascii	bytes	compile	credits	dir
all	bin	callable	complex	delattr	display
any	bool	chr	copyright	dict	divmod

续表

enumerate	getattr	iter	min	print	slice
eval	globals	len	next	property	sorted
exec	help	license	object	range	str
filter	hex	list	oct	repr	sum
float	id	locals	open	reversed	super
format	input	map	ord	round	tuple
frozenset	int	max	pow	setattr	zip

2.2.2 数值型变量

金融量化分析需要处理大量的数据，一般通过数值型变量来存储各类金融数据，这些变量主要包括以下两种。

- 整数型变量（int）：用来表示整数类型的数据。例如，持仓股票个数、股票交易次数等均可用整数型变量表示。
- 浮点型变量（float）：用来表示带有小数的数据。例如，股票收盘价价格、投资收益率等可以使用浮点型变量表示。

Python 在创建变量时即对该变量进行赋值，并根据赋值数据的类型定义该变量的数据类型。可以通过 Python 内置函数“type(变量名)”来查看该变量的数据类型（见示例 2.1）。

【示例 2.1】数值型变量的应用。

代码如下：

```
# 定义交易次数
trade_times = 20
# 定义交易胜率
win_rate = 0.72
# 打印变量数据类型
print("trade_times 的数据类型为：", type(trade_times))
print("win_rate 的数据类型为：", type(win_rate))
```

程序输出结果如下：

```
trade_times 的数据类型为： <class 'int'>
win_rate 的数据类型为： <class 'float'>
```

说明：

- 变量“trade_times”为整数 int 类型变量。
- 变量“win_rate”为浮点 float 类型变量。

2.2.3 布尔类型变量

布尔类型变量（bool）只有两个值，即 True 与 False，分别表示“真”与“假”。由于布尔对象（bool）是整数对象（int）的子类，布尔类型变量等价于整数类型中的 1 与 0，即 True 与 1 等价，False 与 0 等价。

【示例 2.2】布尔类型变量的应用。

需求说明：

- 首先设定止盈价格 price_win 的变量值为 10.00。
- 设置现价 price_now 的变量值为 9.85。
- 如果现价 price_now 的变量值大于止盈价 price_win 的变量值，则变量 win_value 的值设为 True；否则变量 win _ value 的值设为 False。

代码如下：

```
# 设置止盈价格变量名为 price_win
price_win = 10.00
# 设置现价变量名为 price_now
price_now = 9.85
# 如果现价 price_now 大于止盈价格 price_win
if  price_now > price_win:
    # 变量 win_value 被赋值为 True
    win_value = True
# 否则
else:
    # 变量 win_value 被赋值为 False
    win_value= False
    # 打印变量 win_value
    print("win_value 的值为：", win_value)
```

程序输出结果如下：

```
win_value 的值为： False 字符串类型变量
```

2.2.4　字符串类型变量

Python 使用字符串类型变量来存储文本字符和字符串内容，如股票名称和公司代码等信息。在 Python 编程中会大量出现字符串类型变量，字符串对象相关方法与函数也是 Python 基础的重要组成部分。

1．字符串类型变量

Python 一般使用英文输入法下的双引号" "和单引号' '来包裹赋值的字符串类型变量的内容，有时为了存储字符串的内部格式，会使用三个单引号来包裹带有排版格式的字符串内容。具体应用方式见示例 2.3。

【示例 2.3】字符串类型变量的应用。

代码如下：

```
# 定义股票名称
stock_name = '贵州茅台'
# 定义股票代码
stock_code = '600519.SH'
# 定义带格式的字符串变量
a = '''
    公司股票：浦发银行　贵州茅台
```

```
    行    业：银行业    白酒行业
'''
```

说明：

- 字符串是由单个字符组合的数组组成的，Python 可以通过索引号读取字符串中的部分字符。字符串的索引号从 0 开始。例如，读取字符串类型变量的第 2 个字符，可以通过“字符串变量名[1]”来实现。
- 字符串变量支持切片操作。例如，str_name[2:5]表示读取字符串 str_name 变量中第 3～6 个字符的内容。
- 字符串变量支持逆序切片。例如，str_name[-2:]表示读取字符串 str_name 变量从倒数第 2 个到最后一个字符之间的内容。

2．字符串类型变量常用函数与方法

（1）find 函数

功能：用于在一个字符串中查找另一个子字符串，并返回子字符串第一次出现的索引位置，如果未找到则返回-1。

语法：str.find(sub[, start[, end]])。

参数：

- sub：查找的子字符串。
- start：可选参数，指定查找的起始位置。默认值为 0，表示从字符串的开头开始查找。
- end：可选参数，指定查找的结束位置。默认为字符串的长度，即查找到字符串的末尾。

【示例 2.4】str.find 函数的应用。

代码如下：

```
# 在股票代码变量 stock_code 中查找是否包含字符串 SH
print(stock_code.find('SH'))
# 在股票代码变量 stock_code 中查找是否包含 SZ
print(stock_code.find('SZ'))
```

程序输出结果如下：

```
0
-1
```

（2）replace 函数

功能：用于替换字符串变量中指定的字符串，输出结果为替换后的字符串变量。

语法：str.replace(old, new[, count])。

参数：

- old：字符串类型，要被替换的子字符串。
- new：字符串类型，用来替换旧字符串的新字符串。
- count：可选参数，指定替换的次数，默认为所有出现的旧字符串都被替换。如果指定了 count 参数，则只有前 count 个出现的旧字符串会被替换为新字符串，其他

出现的旧字符串将保持不变。

【示例 2.5】str.replace 函数的应用。

代码如下：

```
# 将股票代码变量 stock_code 中的 SH 替换为“沪市”
print(stock_code.replace('SH', '沪市'))
```

程序输出结果如下：

```
沪市.600519
```

（3）strip 函数

功能：用于去除字符串开头和结尾的指定字符（默认为空格和换行符）。

语法：str.strip([chars])。

参数：chars 为可选参数，指定要去除的字符。默认值为空格和换行符，表示去除空格和换行符。

【示例 2.6】str.strip 函数的应用。

代码如下：

```
# 将股票代码变量 stock_code 中的 SH.删除
print(stock_code.strip('SH.'))
```

程序输出结果如下：

```
600519
```

（4）join 函数

功能：用于将序列中的元素连接成一个字符串。

语法：str.join(iterable)。

参数：iterable 表示一个可迭代对象，如列表、元组和字符串等。

【示例 2.7】str.join 函数的应用。

代码如下：

```
# 定义列表变量，元素为字符串类型数据
date_list = ['2023', '1', '15']
# 设置连接符号变量为字符串类型数据
string1 = '-'
# 将 date_list 内部元素使用 string1 进行连接，生成一个新的字符串类型数据
string1.join(date_list)
```

程序输出结果如下：

```
'2023-1-15'
```

（5）count 函数

功能：用于计算某个子字符串在当前字符串中出现的次数。

语法：str.count(sub, start=0, end=len(string))。

参数：

- str：要进行计数操作的字符串。
- sub：要查找的子字符串。

- start：开始查找的索引位置（默认为 0）。
- end：结束查找的索引位置（默认为 len(string)）。

【示例 2.8】str.count 函数的应用。

代码如下：

```
# 统计字符串类型变量 stock_list 中含有 SH 的个数
stock_list = '600000.SH 600519.SH 600188.SH'
# 在字符串变量 stock_list 中出现 SH 的次数
stock_list . count ('SH')
```

程序输出结果如下：

```
3
```

2.2.5 列表类型变量

列表（list）是在金融量化分析过程中常用的一种数据类型，在进行金融量化分析时，需要各类序列类型的原始数据，它们均可保存在列表变量中。例如，股票每日收盘价即可以列表类型进行存储，并可通过列表对象方法对列表内部元素进行各类操作。

1．列表变量

在 Python 中，列表会将所有各类数据元素都放在一对中括号“[ ]”里面，相邻数据元素之间用逗号分隔。在金融量化分析中，常用列表变量来保存各类序列数据，如一段期间的日线序列数据、策略收益率变动序列数据等。

【示例 2.9】创建列表。

代码如下：

```
# 定义一个存储开盘价序列的列表变量 open_list
open_list = [10.12, 10.58, 11.00, 13.25]
# 定义一个存储股票代码的列表变量 stock_list
stock_list = ['600000.SH', '000001.SZ', '600188.SH']
```

2．列表变量的常用方法（函数）

（1）append 方法

功能：在列表末尾添加新的元素。

语法：list.append(item)。

参数：item 表示要添加到列表末尾的元素。

说明：list.append 函数将在原地修改列表，即它会直接修改调用它的列表，而不是创建一个新列表。

【示例 2.10】list.append 方法的应用。

代码如下：

```
# 在列表变量 stock_list 末尾添加元素 000007.SZ
stock_list.append('000007.SZ')
```

```
# 打印输出添加元素 000007.SZ 后的列表 stock_list
print(stock_list)
```

程序输出结果如下：

```
['600000.SH', '000001.SZ', '600188.SH', '000007.SZ', '000007.SZ']
```

（2）insert 方法

功能：将指定对象插入列表的指定位置。

语法：list.insert(index, item)。

参数：

- index：设置要插入元素的位置。
- item：设置要插入列表的元素。

【示例 2.11】list.insert 方法的应用。

代码如下：

```
# 在列表变量 stock_list 的第 2 个位置（列表的索引号也是从 0 开始的）插入元素 600188.SH
stock_list.insert(1, '600188.SH')
# 打印输出插入新元素后的列表 stock_list
print(stock_list)
```

程序输出结果如下：

```
['600000.SH', '600188.SH', '000001.SZ', '600188.SH']
```

（3）remove 方法

功能：删除列表中某个值的第一个匹配项。

语法：list.remove(item)。

参数：item 为要从列表中删除的元素。

【示例 2.12】list.remove 方法的应用。

代码如下：

```
# 从列表变量 stock_list 中删除元素 000007.SZ
stock_list.remove('000007.SZ')
# 打印输出删除元素 000007.SZ 后的列表 stock_list
print(stock_list)
```

程序输出结果如下：

```
['600000.SH', '600188.SH', '000001.SZ', '600188.SH']
```

（4）pop 方法

功能：删除列中的一个元素（默认为最后一个元素），并返回该元素的值。

语法：list.pop(index)。

参数：index 用于设置要删除的元素的索引位置。如果不传入任何参数，则默认删除列表的最后一个元素。

【示例 2.13】list.pop 方法的应用。

代码如下：

```
stock_list.pop(1)
# 从列表变量 stock_list 中删除第 2 个元素
```

程序输出结果如下：

```
'600188.SH'
```

（5）count 方法

功能：统计某个元素在列表中出现的次数。

语法：list.count(item)。

参数：item 用于设置需要统计出现次数的元素。

【示例 2.14】list.count 方法的应用。

代码如下：

```
# 统计列表变量 stock_list 中 600000.SH 出现的次数
times = stock_list.count('600000.SH')
# 打印输出列表 stock_list 中包含 600000.SH 的次数
print('600000.SH 在列表中出现的次数：', times)
```

程序输出结果如下：

```
1
```

（6）sort 方法

功能：对原列表进行排序，如果指定参数，则使用比较函数来指定要比较的函数。

语法：list.sort(key=None, reverse=False)。

参数：

- key：可选参数，用于指定排序时要使用的键（函数）。
- reverse：可选参数，用于指定排序顺序，如果为 True 则按照降序排列，否则按照升序排列。
- 如果不指定这两个参数，则按照默认顺序（即升序）排序。

【示例 2.15】按 stock_list 元素的数字部分升序排序。

代码如下：

```
# 定义函数
def get_stock_num(stock_code):
    # 返回从第 4 个字符开始到最后一个字符即股票代码
    return stock_code[3:]

# 列表对象按逆序，将 get_stock_num 函数返回值进行排序
stock_list.sort(key=get_stock_num, reverse=False)
# 打印排序后的列表变量
print(stock_list)
```

程序输出结果如下：

```
['000001.SZ', '600000.SH', '600188.SH']
```

2.2.6 元组类型变量

元组（tuple）与列表（list）的类型一样，可以存储包含多个元素的变量。与列表不同的是，元组一旦创建就不可以对内部元素进行追加和删除操作。

元组变量可以通过圆括号“( )”进行创建。为了区分元组的圆括号“( )”与运算符的圆括号“( )”，当元组内部只有一个元素时，需要在该元素后面加上一个逗号“,”。

【示例 2.16】元组变量赋值与内部元素读取。

代码如下：

```
# 定义并赋值元组类型的变量 stock_tuple_1 内部有多个元素
stock_tuple_1 = ('贵州茅台', '中国石化', '工商银行')
# 定义并赋值元组类型的变量 stock_tuple_2 内部只有一个元素
stock_tuple_2 = ('赢时胜',)
# 元组元素的读取
print('stock_tuple_1 第 3 个元素为：', stock_tuple_1[2])
# 两个元组变量的数学运算
stock_tuple_3 = stock_tuple_1 + stock_tuple_2
print('stock_tuple_1 + stock_tuple_2 生成的元组变量 stock_tuple_3 为：',
stock_tuple_3)
```

程序输出结果如下：

```
stock_tuple_1 元组的第 3 个元素为： 工商银行
stock_tuple_1+stock_tuple_2 生成的元组变量 stock_tuple_3 为： ('贵州茅台',
'中国石化', '工商银行', '赢时胜')
```

说明：

- 元组的相关函数与列表函数的使用方式基本一致。
- 两个元组可以相加，相加结果是两个元组元素合并在一起，如果元素相同，则相同元素会重复出现在新的元组内。
- 虽然两个元组可以相加，但是两个元组不可以进行相减运算。

2.2.7　集合类型变量

列表和元组内部元素按顺序存储，并且可以包含值相同的元素。集合类型变量（set）则与列表、元组不同，它是一种存放不重复元素的数据类型，并且集合中的元素都是无序的。

1．集合变量

在 Python 中，集合类型变量既可以通过大括号{ }来创建，也可以通过 Python 内置函数 set 传入一个列表或元组进行创建（见示例 2.17）。

【示例 2.17】创建集合类型变量。

代码如下：

```
# 定义集合类型变量
stock_set1 = {'贵州茅台', '中国石化', '工商银行'}
# 打印集合类型变量 stock_set1
print(stock_set1)

# 定义集合类型变量
stock_set2 = set(stock_list)
# 打印集合类型变量 stock_set2
```

```
print(stock_set2)
```

程序输出结果如下：

```
{'工商银行', '中国石化', '贵州茅台'}
{'000001.SZ', '600188.SH', '600000.SH'}
```

说明：

- 由于集合内部元素是无重复、无顺序保存的，因此，集合内部元素无法通过索引或切片形式读取。判断元素与集合的关系只可以通过 in 关键词来判断。例如 set_1={1,2,3}，1 in set_1 返回结果为 True，4 in set_1 则返回结果为 False。
- 有时，为了清除列表或元组内部重复的元素，可以将列表或元组强制转成集合类型，即要保证内部元素的唯一性。
- 集合的相关方法与列表方法的使用方式基本一致。

2．集合变量的常用方法（函数）

Python 中的集合变量可以通过集合类成员方法（函数）完成多种操作，如添加元素、删除元素和插入元素等，具体方法（函数）包括以下几种。

- set.add(item)：向集合中添加一个元素，如果元素已经存在，则不会进行任何操作。
- set.update(iterable)：向集合中添加一个可迭代对象中的所有元素。
- set.remove(item)：从集合中删除一个元素，如果元素不存在，则引发 KeyError 异常。
- set.discard(item)：从集合中删除一个元素，如果元素不存在，则不会进行任何操作。
- set.pop：删除并返回集合中的任意一个元素，如果集合为空，则引发 KeyError 异常。
- set.clear：从集合中删除所有元素。
- set.copy：返回集合的一个副本。
- set.union(other_set)：返回一个包含两个集合中所有元素的新集合。
- set.intersection(other_set)：返回一个包含两个集合中共同元素的新集合。
- set.difference(other_set)：返回一个包含所有存在于第一个集合但不存在于第二个集合中的元素的新集合。
- set.symmetric_difference(other_set)：返回一个包含所有存在于两个集合中但不重复的元素的新集合。
- set.issubset(other_set)：检查第一个集合是否为第二个集合的子集，返回布尔值。
- set.issuperset(other_set)：检查第一个集合是否为第二个集合的超集，返回布尔值。

需要注意的是，集合是可变对象，上述方法都会修改原始集合，而不是返回新的集合。如果需要一个新的集合而不影响原始集合，则可以使用 set.copy 函数创建一个副本。

2.2.8 字典类型变量

字典类型（dict）变量也是金融量化分析中经常用到的一种 Python 数据类型。与列表、

元组、集合类型数据不同，字典是通过键值对的形式存储数据。

1．字典变量

在 Python 中，字典类型变量需要通过大括号{ }进行创建，字典内部每个元素以 key：value 的形式存放。例如，{'stock_code':'600000.SH'}表示字典中有一个键为 stock_code，对应的值为 600000.SH。

【示例 2.18】创建字典。

代码如下：

```
# 定义字典类型变量 stock_info_dict
stock_dict = {'stock_name':'贵州茅台', 'date':'2023-01-16', 'price':1912.90}
# 打印字典类型变量 stock_info_dict
print(stock_dict)
```

程序输出结果如下：

```
{'stock_name': '贵州茅台','date': '2023-01-16', 'price': 1912.9 }
```

2．字典变量的常用函数

- dict.keys：返回一个包含字典所有键的列表。
- dict.values：返回一个包含字典所有值的列表。
- dict.items：返回一个包含字典所有键值对的列表，每个键值对表示一个元组。
- dict.get(key, default=None)：返回给定键的值，如果键不存在，则返回默认值。
- dict.pop(key, default=None)：删除并返回给定键的值，如果键不存在，则返回默认值。
- dict.popitem：删除并返回字典中的任意一个键值对。
- dict.clear：删除字典中的所有键值对。
- dict.update(other_dict)：将另一个字典中的键值对添加到当前字典中。
- dict.fromkeys(iterable, value=None)：返回一个新字典，其键为可迭代对象中的元素，值为指定的值。
- dict.setdefault(key, default=None)：返回给定键的值，如果键不存在，则将给定键和默认值插入字典。

需要注意的是，字典是可变对象，上述函数都会修改原始字典，而不是返回新的字典。如果需要一个新的字典而不影响原始字典，则可以使用 copy 函数创建一个副本。

2.3　流 程 控 制

Python 的流程控制包括条件语句和循环语句，其中，条件语句包括 if 语句、if…else 等语句，循环语句包括 for 语句与 while 语句。逻辑判断在流程控制中发挥基础作用。

2.3.1 逻辑判断

在 Python 中，逻辑判断涉及布尔类型变量与逻辑运算符两个要素。其中，逻辑运算符包括 and（与）、or（或）和 not（非），这 3 种逻辑运算符使用的函数如下：

- and：条件 1 and 条件 2，表示如果条件 1 和条件 2 都是 True 则返回 True，否则返回 False。
- or：条件 1 or 条件 2，表示如果条件 1 和条件 2 都是 False 则返回 False，否则返回 True。
- not：not 条件 1，表示如果条件 1 是 True 则返回 True，否则返回 False。

and（与）、or（或）和 not（非）这 3 种逻辑运算符可以一起使用，一般使用括号来标记运算优先级。

【示例 2.19】逻辑运算符的应用。

代码如下：

```
# 设置两个条件，分别为 con1 与 con2
# 条件 con1 表示字典 stock_dict 的键 name 对应的值为“贵州茅台”
con1 = stock_dict['stock_name'] == '贵州茅台'
# 条件 con2 表示字典 stock_dict 的键 trade_date 对应的值为“20230120”
con2 = stock_dict['date']=='20230116'
# 打印结果：con1 与 con2 同时为真
print(con1 and con2 )
# 打印结果：stock_dict['trade_date']=='20230121'为假
print(stock_dict['stock_name']=='贵州茅台' and stock_dict['date']==
'2023-01-16')
```

程序输出结果如下：

```
False
True
```

2.3.2 if 判断

if 语句既可以进行单一条件判断，又可以进行多条件判断。单一条件判断通过 if…else 语句实现，多条件判断可通过 if…elif…else 语句实现。

1．if…else单一条件判断

if…else 语句是 Python 编程中最常用的判断语法结构。通过 if…else 条件判断可以设置触发条件，并执行对应条件的代码与功能。例如，金融量化策略在判断是否满足入场交易条件时，可以通过 if…else 语句进行判断。if…else 条件判断语句的语法如下：

```
if 条件判断：
      执行语句 1
else：
      执行语句 2
```

【示例 2.20】单一条件判断流程的应用。

代码如下：

```
# 买入价为 10 元
buy = 10
# 收盘价为 9 元
close = 9
# 如果收盘价大于买入价的 1.1 倍
if  close > buy * (1+0.1):
    print('卖出')
# 否则
else:
    print('继续持有')
```

程序输出结果如下：

```
继续持有
```

2．if…elif…else多条件判断

if…elif…else 可以完成判断条件多于两个的判断任务，使用语法如下：

```
if 条件判断 1：
    执行语句 1
elif 条件判断 2：
    执行语句 2
else：
    执行语句 3
```

【示例 2.21】多条件判断流程的应用。

说明：如果收盘价大于买入价的 20%，则打印“上涨超过 20%”；如果收盘价介于买入价的 1～1.2 倍之间，则打印“上涨幅度小于 20%”；否则打印“无上涨”。

代码如下：

```
# 条件 1：如果收盘价大于买入价的 1.2 倍
if  buy * (1+0.2) < close:
    print("太棒了！！上涨幅度超过 20% ！")
# 条件 2：如果收盘价大于买入价但小于买入价的 1.2 倍
elif  buy * (1+0.2) >= close>buy:
    print("还可以！！上涨幅度小于 20% ！")
# 条件 3：其余情形（否则）
else:
    print("没有上涨！")
```

程序输出结果如下：

```
没有上涨！
```

2.3.3　循环语句

Python 中的循环语句包括 for 循环和 while 循环。for 循环与 while 循环都可以执行循环操作，二者的主要区别在于：for 循环先循环，后判断是否符合继续循环的条件；而 while 循环是先判断是否执行循环的条件，当满足循环的条件时再执行循环体操作。

1．for循环

金融量化分析常常需要多次执行相同的操作，例如，需要针对多个数据文件进行相同的读取和分析操作等。for 循环作用于一个可迭代对象，可迭代对象包括列表、元组、集合、字典和字符串等。for 循环可以实现遍历可迭代对象内部的全部元素并重复执行特定的操作，直至可迭代对象内部的元素全部遍历结束。语法如下：

```
for 变量 in 可迭代变量：
    执行语句（重复执行部分）
```

【示例 2.22】利用 for 循环将列表内的股票名称全部打印出来。

代码如下：

```
# 创建列表变量“stock_buy_list”，用于存储需要买入的股票名称
stock_buy_list = ['贵州茅台', '工商银行', '老白干', '复星医药']
# 使用 for 循环遍历列表变量“stock_buy_list”内部的每个元素
for stock_name in stock_buy_list:
    # 打印列表变量“stock_buy_list”内部的每个元素
    print(stock_name)
```

程序输出结果如下：

```
贵州茅台
工商银行
老白干
复星医药
```

2．while循环

与 for 循环相似，while 循环也可以根据设置的条件执行重复的操作，语法格式如下：

```
while  条件判断：
    执行语句（重复执行部分）
```

while 循环需要事先指定一个循环结束的条件，每循环一次，在开始之前都需要进行一次条件判断，如果满足这个 while 条件就可以继续进行循环，否则就终止循环，跳转至 while 循环之外的代码语句。如果 while 未设置终止循环条件，则需要在 while 循环内部的执行语句中设置循环终止条件；否则，该 while 循环将一直重复运行循环内部的代码，从而造成无限循环或死循环的情形。

【示例 2.23】利用 while 循环将列表内的股票名称逐个打印出来。

代码如下：

```
# 设置计数器，初始值为 0
i = 0
# 当计数器小于“stock _ buy _ list”列表长度时，则执行 while 循环内部语句
while i < len(stock_buy_list):
    # 打印 stock_name
    print(stock_buy_list[i])
    # 计算器加 1
    i += 1
```

程序输出结果如下：

```
贵州茅台
工商银行
老白干
复星医药
```

3．break关键字

在 for 与 while 循环结构中，如果需要在循环体内部中断循环，则可以通过 break 关键字来实现。在利用 break 终止循环结构继续运行之前，一般需要进行逻辑判断是否执行 break 语句。

4．continue关键字

continue 关键字的作用是在 for 循环体内结束本次循环，继续执行下一次循环。与 break 关键字一样，使用 continue 关键字时也需要与逻辑判断语句相结合。

【示例 2.24】打印列表内的股票名称，“工商银行”除外。

代码如下：

```
# 使用 for 循环遍历列表变量“stock_buy_list”内部的每个元素
for stock_name in stock_buy_list:
    # 设置执行“continue”语句的逻辑判断条件
    # 当遇到循环至“工商银行”时，则跳出本次循序，执行下一次循环
    if stock_name == '工商银行':
        continue
    print(stock_name)
```

程序输出结果如下：

```
贵州茅台
老白干
复星医药
```

2.4　函　　数

在 Python 中，函数是一段可重用的代码块，可用于完成特定的功能。Python 函数既可以接受参数，也可以产生返回值。Python 提供了一些内置函数，如 print 和 len 函数等，同时其也支持用户自定义函数。

2.4.1　函数的定义与调用

在调用函数之前，必须事先对函数进行定义。由于每个函数会执行特定的功能，所以在定义函数之前需要确定函数的功能，以便进行代码封装。如果函数内部计算需要从函数外部调取数据，则需要为函数设置形式参数。

在函数体内部完成计算后，如果需要返回计算结果，则可以通过 return 语句返回函数计算结果。如果无须返回计算结果，则可以省略 return 语句或 return None（返回空值）。定

义函数的语法如下：

```
def 函数名称（形式参数列表）：
    函数体代码
    return 函数结果
```

说明：

- 创建与定义函数必须通过 def 关键字来声明，函数名称通常与函数功能相关，以提高函数的可读性。def 关键字与函数名称之间用一个空格分隔。
- 函数的形式参数列表为可选项，可以根据函数功能的实际需要确定是否需要设置形式参数。形式参数可以直接设置默认值，如果函数调用时不传递实际参数，则按默认值执行函数。
- 可以通过“函数名称（参数列表）”方式调用函数。

【示例 2.25】定义计算股票的收益率函数。

代码如下：

```
# 定义函数 calculate_return_ratio
# 参数：buy_price 与 sell_price
def calculate_return_ratio(buy_price, sell_price):
    return_ratio = sell_price / buy_price - 1
    return return_ratio

# 定义 buyprice 变量，用于存储买入价格
buyprice = 10
# 定义 sellprice 变量，用于存储卖出价格
sellprice = 11
# 设置变量 investratio，用于保存 caculate_return_ratio 函数返回值
investratio = calculate_return_ratio(buyprice, sellprice)
# 打印函数返回值
print(investratio)
```

程序输出结果如下：

```
0.100000
```

说明：

- 函数名称为 calculate_return_ratio。
- 函数的形式参数有两个，即 buy_price 与 sell_price。
- 函数功能为计算参数 sell_price 与参数 buy_price 之间的收益率。
- 函数返回值为 return_ratio。
- investratio = calculate_return_ratio(buyprice, sellprice)中的 buyprice 与 sellprice 是实际参数，分别代表 10 与 11。

2.4.2 函数的参数

Python 函数在定义时需要设置形式参数。形式参数的数目可以是 0 个，也可以是多个，甚至是不确定数量。在定义函数的过程中，必须注意形式参数的不同设置方法，防止发生各类报错的情形。

1．形式参数的顺序

当形式参数为多个时，调用函数所传递的实际参数顺序应该与形式参数的顺序保持一致。但是，如果直接将实际参数赋值给形式参数关键字，则形式参数的顺序可以与函数调用时的实际参数的顺序不一致。例如，calculate_return_ratio(buy_price, sell_price)函数包括两个形式参数，第一个形式参数为 buy_price，第二个形式参数为 sell_price。

- 如果以 calculate_return_ratio(10, 20)调用该函数，则 10 代表第一个形式参数 buy_price，20 代表第二个形式参数 sell_price。
- 如果以 calculate_return_ratio(sell_price=20, buy_price=10)调用该函数，10 仍然代表第一个形式参数 buy_price，20 仍然代表第二个形式参数 sell_price。

2．形式参数默认值的设置

形式参数既可以全部设置为默认值，也可以部分设置为默认值。但是，如果只为部分形式参数设置了默认值，则设置了默认值的形式参数应该集中放置在形式参数末尾，即在没有设置默认值的形式参数前，不允许出现设置默认值的形式参数，否则会报错 SyntaxError: non-default argument follows default argument。例如：

- def calculate_return_ratio(buy_price=10, sell_price)就会报错。
- def calculate_return_ratio(buy_price, sell_price=20)则是正确写法，此时以 calculate_return_ratio(10)调用函数 calculate_return 会返回 1.0 的正确结果，即 calculate_return_ratio 的实际参数 10 代表形式参数 buy_price，而 sell_price 以默认值参与函数体内的计算。

3．可变形式参数

形式参数的长度可以是事先不确定的，即函数可以接受不确定数量的实际参数。在 Python 中，可变参数有两种表示方式，一种是基于元组的可变参数，通常以*args 表示；另一种是基于字典的可变参数，通常以**kwargs 表示。

【示例 2.26】元组类型可变参数的应用。

代码如下：

```
# 函数 my_function 的形式参数为元组类型的可变参数
def my_function(*args):
    print(*args)

# 调用函数 my_function，直接传入元组类型的实际参数
my_function(1, 2)
```

程序输出结果如下：

```
1 2
```

说明：

- 自定义函数 my_function 的可变参数*args，其中 args 表示元组类型变量，在调用 my_function 函数时传入的 1 和 2 即为元组 args 的值。

- *表示打开元组变量 args。在本例中，print(*args)为打印元组变量 args 内部的元素，即输出结果为 1 2。如果改为 print(args)则为打印元组变量 args，输出结果为(1,2)。

【示例 2.27】字典类型可变参数的应用。

代码如下：

```
# 函数 my_function 的形式参数为字典类型的可变参数
def my_function(**kwargs):
    print(kwargs)

# 调用函数 my_function，传入字典类型的实际参数
my_function(price=10, volume=300)
```

程序输出结果如下：

```
{'price': 10, 'volume': 300}
```

说明：

- 定义 my_function 时可变参数为**kwargs，即说明在该函数调用时应该以字典形式传递实际参数。
- 在本例中，当调用 my_function 函数时，实际参数为 price=10, volume=300。如果直接以数值 10 与 300 作为实际参数调用该函数，即 my_function(10, 300)，则会报错 TypeError: my_kwargs_function() takes 0 positional arguments but 2 were given。

2.4.3 lambda 匿名函数

lambda 匿名函数是一种可以在代码中临时定义并调用的小型函数。lambda 匿名函数通常是一行代码，并且只被使用一次。

语法：lambda 参数列表: 函数体。

说明：

- lambda 匿名函数的参数列表可以包含 0 个或多个参数。如果是多个参数，则可以用逗号分隔。
- lambda 匿名函数体通常是一个表达式，用于计算并返回的函数结果。

【示例 2.28】lambda 匿名函数的应用。

代码如下：

```
# 创建匿名函数，并将返回值赋给 add 变量
add = lambda x, y: x + y

# 调用 lambda 匿名函数，并传入实际参数 3、5 给匿名函数中的 x 与 y 形式参数
result = add(3, 5)
print(result)
```

程序输出结果如下：

```
8
```

说明：

- 本例使用 lambda 关键字定义了一个匿名函数并赋值给变量 add，lambda 匿名函数

需要接受两个参数，即 x 和 y，lambda 匿名函数的功能是计算 x+y 的值，并将计算结果返回。

- 由于 lambda 匿名函数通常只被使用一次，因此它的主要优势在于简洁性和可读性。如果需要定义一个功能复杂的函数，通常应该使用常规函数定义，使其代码更易于理解和维护。

2.4.4　Python 高阶函数

高阶函数是指可以接受函数作为参数或者以返回函数作为结果的函数。Python 中的许多内置函数都是高阶函数，如 map、filter 和 reduce 等。

1．map函数

功能：将一个可迭代对象中的元素作为参数传递给某个函数，该函数将返回一个新的可迭代对象，返回的可迭代对象包含函数的全部返回值。

语法：map(function, iterable)。

参数：

- function：需要多次执行的函数。
- iterable：可迭代对象。

【示例 2.29】map 函数的应用：根据 price 列表元素计算收益率。

代码如下：

```
# 创建浮点类型数据的列表 price
prices = [100.0, 105.5, 110.2, 108.0, 112.3, 115.5]
# 计算 price 列表内部元素之间的涨跌幅
returns = map(lambda p:round((p-prices[0])/prices[0], 2), prices)
print(list(returns))
```

程序输出结果如下：

```
[0.0, 0.055, 0.10, 0.08, 0.12, 0.15]
```

说明：

- 在本例中，map 函数中的 function 是一个 lambda 匿名函数，该 lambda 匿名函数的功能是计算收益率。
- 高阶函数 map 将 lambda 匿名函数应用于列表 price 的每个元素中，即计算列表 price 中的每个元素与第 1 个元素的涨幅作为返回值并逐个保存在 returns 变量中。
- 在本例中，map 函数的返回值 returns 是 map 类型数据，map 类型数据的内容不能直接打印出来，可以将该变量强制转换为列表类型，即进行 list(returns)操作后方可打印。

2．filter函数

功能：filter 函数用于过滤一个可迭代对象中的元素，并返回一个新的可迭代对象，其中只包含满足指定条件的元素。在金融领域内，filter 函数常被用于条件筛选，如筛选出符

合要求的股票代码、剔除数据序列中的空值与异常值等。

语法：filter(function, iterable)。

参数：

- function：需要执行的函数，该函数用于指定具体的过滤条件。
- iterable：可迭代对象，用于指定需要进行过滤的元素集。

【示例 2.30】filter 函数的应用：从股票列表中筛选出深市股票代码。

代码如下：

```
# 创建股票代码列表 stocks_list
stocks_list= ['60000.SS','000001.SZ','600004.SS','600519.ss', '000032.sz']
# 在股票代码列表 stocks_list 中筛选出结尾是“SS”的股票代码，并保存至 stocks_SS 列表
stocks_SS = filter(lambda x: x[-2:].upper() == 'SS', stocks_list)
print(list(stocks_SS))
```

程序输出结果如下：

```
['60000.SS', '600004.SS', '600519.ss']
```

说明：

- 本例中的筛选函数 function 为 lambda 匿名函数，该 lambda 匿名函数将参数最后两位转换为大写格式与“SS”进行比较。
- filter 函数以 lambda 匿名函数比较结果来确定留下的元素，如果 lambda 匿名函数比较结果为 True，则该元素被保留在 filter 函数的返回结果内；如果比较结果为 False，则该元素不被保留在 filter 函数的返回结果内。
- filter 函数的返回值内容不能直接被打印出来，可以将其转换成为列表类型数据后方可被打印。

【示例 2.31】filter 函数的应用：筛选符合条件的股票。

代码如下：

```
# 定义 stock 变量，即元素为字典的列表类型数据
stocks = [
    {'symbol': 'AAPL', 'pe_ratio': 31.24, 'dividend_yield': 1.24},
    {'symbol': 'GOOG', 'pe_ratio': 27.35, 'dividend_yield': 0.78},
    {'symbol': 'MSFT', 'pe_ratio': 35.12, 'dividend_yield': 0.87},
    {'symbol': 'AMZN', 'pe_ratio': 76.43, 'dividend_yield': 0},
]

# 筛选出市盈率低于 35 的股票
low_pe_stocks = filter(lambda stock: stock['pe_ratio'] > 35, stocks)
print(r'市盈率低于 30 的股票：{}'.format(list(low_pe_stocks)))

# 筛选出股息率高于 1%的股票
high_dividend_stocks = filter(lambda stock: stock['dividend_yield'] > 1,
stocks)
print(r'股息率高于 1%的股票：{}'.format(list(high_dividend_stocks)))
```

程序输出结果如下：

```
市盈率低于 30 的股票：[{'symbol': 'MSFT', 'pe_ratio': 35.12, 'dividend_yield':
0.87}, {'symbol': 'AMZN', 'pe_ratio': 76.43, 'dividend_yield': 0}]
股息率高于 1%的股票：[{'symbol': 'AAPL', 'pe_ratio': 31.24, 'dividend_yield':
```

```
1.24}]
```

【示例 2.32】filter 函数的应用：数据清洗之剔除空值与异常值。

代码如下：

```
# 创建含有空值 None 的列表 data
data = [0.5, 0.3, None, 0.7, 0.2, 21.2, 0.6, None, 0.9, 1.0]
# 剔除 data 列表中的空值
clean_data = list(filter(lambda x: x is not None, data))
print(r'去除空值 None 之后的序列：{}'.format(clean_data))

# 剔除异常值（偏离均值超过 2 倍标准差的值）
# 计算均值
mean = sum(clean_data) / len(clean_data)
# 计算标准差
std_dev = (sum((x - mean) ** 2 for x in clean_data) / len(clean_data)) ** 0.5

# 剔除偏离均值超过 2 倍标准差的值
filtered_data = filter(lambda x: abs(x - mean) <= 2 * std_dev, clean_data)
print(r'去除异常值后的序列：{}'.format(list(filtered_data)))
```

程序输出结果如下：

```
去除空值 None 之后的序列：[0.5, 0.3, 0.7, 0.2, 21.2, 0.6, 0.9, 1.0]
去除异常值后的序列：[0.5, 0.3, 0.7, 0.2, 0.6, 0.9, 1.0]
```

说明：

- lambda x: x is not None 返回布尔类型数据，判断 x 不是空值。如果 x 是空值 None，则返回 False；如果 x 不是空值 None，则返回 True。
- 在本例中，异常值的标准定义为偏离均值附近 2 倍标准差，其中，21.2 属于异常值，因此被剔除。

3．reduce函数

功能：用于对一个可迭代对象中的元素逐个应用一个函数，从而将所有元素缩减为单个值。在金融领域计算中，reduce 函数可以用于累计收益率计算、资产组合收益率计算等场景。

语法：

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

参数：

- function：用于指定要应用的函数。
- iterable：可迭代对象，用于指定需要逐一处理的元素集。
- initializer：可选的初始值，用于在可迭代对象为空时作为结果返回。

【示例 2.33】reduce 函数的应用：计算列表元素加总求和。

代码如下：

```
# 导入 functools 模块中的 reduce 函数
from functools import reduce
```

```
# 创建整数型列表 numbers
numbers = [1, 2, 3, 4, 5]
# 计算列表 numbers 内部元素的总行
product = reduce(lambda x, y: x + y, numbers, 20)
print(product)
```

程序输出结果如下：

```
15
```

在金融领域中，reduce 函数可以用于计算累计收益率和加权平均收益率等。

【示例 2.34】reduce 函数的应用：基于每日收益率计算累计收益率。

代码如下：

```
from functools import reduce

# 创建收益率列表 returns
returns = [0.1, 0.2, 0.3]
# 以 returns 列表内部元素为每期收益率，计算 returns 列表的累计收益率
cumulative_return = reduce(lambda x, y: x * (1 + y), returns, 1) - 1
print(cumulative_return)
```

程序输出结果如下：

```
0.716
```

说明：

- reduce 函数的 function 参数为一个 lambda 匿名函数，可迭代对象 iterable 为列表 returns，初始值 initializer 参数为 1，表明当不满足匿名函数计算要求时，reduce 函数返回结果为 1。
- 当 reduce 函数第 1 次调用这个 lambda 函数时，x 的初始值为 1，y 的值为 0.1，即将 returns 列表的第 1 个元素 0.1 作为 y 的初始值。因此，第 1 次调用时，x 的值为 1，y 的值为 0.1。
- 以后每次调用 lambda 函数时，x 的值都是上一次调用的结果，而 y 的值则是 returns 列表中的下一个元素。因此，当第 2 次调用时，x 的值为 1.1，y 的值为 0.2；当第 3 次调用时，x 的值为 1.32，y 的值为 0.3。
- reduce 函数返回的值为(1.32× (1 + 0.3)) − 1，即累计收益率减去初始值 1，结果为 0.716。

【示例 2.35】reduce 函数的应用：计算投资组合加权平均收益率。

代码如下：

```
"""
这是一个使用 reduce 函数计算投资组合加权平均收益率的示例：
假设持有三个资产 A、B、C：
收益率：10%、 20%、 30%；
权 重： 0.3、 0.4、 0.3。
现计算这个投资组合的加权平均收益率：
"""
from functools import reduce

# 三个资产 A、B、C 的收益率存放于 returns 列表
returns = [0.1, 0.2, 0.3]
```

```
# 三个资产 A、B、C 的权重存放于 weights
weights = [0.3, 0.4, 0.3]
# 计算三个资产 A、B、C 的实际收益率，存放于 weighted_returns 列表
weighted_returns = [r * w for r, w in zip(returns, weights)]
# 通过 reduce 函数计算三个资产 A、B、C 组合的实际收益率
weighted_average_return = reduce(lambda x, y: x + y, weighted_returns) /
sum(weights)
print(weighted_average_return)
```

程序输出结果如下：

```
0.2
```

说明：

- 收益率和权重分别存储在列表 returns 和 weights 中。
- 使用 zip 函数将这两个列表打包成一个元组列表，将每个元组中的收益率和权重相乘得到加权收益率列表 weighted_returns，即[0.03, 0.08, 0.09]。
- reduce 函数应用于 weighted_returns 列表，计算出加权平均收益率。当第 1 次调用 lambda 匿名函数时，x 的初始值为 weighted_returns 列表的第 1 个元素，即 0.03，y 的值为 weighted_returns 列表的第 2 个元素，即 0.08。因此，当第 1 次调用时，x 的值为 0.03，y 的值为 0.08。此后每次调用这个 lambda 匿名函数时，x 的值都是上一次调用的结果，而 y 的值则是 weighted_returns 列表中的下一个元素。因此，当第 2 次调用时，x 的值为 0.11，y 的值为 0.09，返回结果为 0.2。

2.5　类 与 对 象

作为面向对象的程序语言，类 class 与对象 object（或 instance）是 Python 的两个重要概念。其中：类是一种定义对象类型的结构，用于封装数据和函数；对象则是类的实例，可以访问类中定义的属性和方法。

2.5.1　创建类与实例对象

在创建类时，需要完成对类内部成员的定义。类内部成员主要包括类变量、初始化构造函数、类成员函数及 property 属性等，具体内容如下。

1．类变量

- 类变量又称为类属性，是指定义在类中而不是在类函数中的变量。它是所有实例共享的变量，可以被类的所有实例对象、类内部的所有函数访问和修改。
- 类变量无论在类内部访问还是在类外部访问，其访问的方式均为“类名称.类变量”。类变量可以看作类的全局变量，它在整个类的生命周期中都是存在的。
- 类变量通常用于存储与类有关的数据，如类的成员数量和类的名称等。类变量还可以用于在类的所有实例之间共享数据，如记录所有实例对象的计数器等。

2．初始化构造函数__init__

- 构造函数的名称是__init__()，它是用于初始化类的实例对象的特殊函数，构造函数在创建类的实例对象时自动调用，用于设置实例对象的初始状态。
- 构造函数通常用于初始化实例变量，即定义在实例对象中的变量。实例变量是每个实例对象独有的，它们在实例对象创建时被分配，并在整个实例对象的生命周期中存在。
- 构造函数是可选的。如果没有定义构造函数，Python 会使用默认的构造函数来创建实例对象。默认的构造函数不会做任何事情，只会创建一个空的实例对象。如果需要在创建实例对象时执行一些初始化操作，例如设置实例变量的初始值，则需要定义一个构造函数。

3．类成员函数

- 类成员函数是定义在类中的函数，它通常用于实现类的功能，如计算和操作数据等。
- 类成员函数必须包含一个名为 self 的参数，该参数用于表示调用该函数的实例对象。在调用类成员函数时，Python 会自动将实例对象作为第 1 个参数传递给该函数。因此，在定义类成员函数时，必须将 self 参数作为第 1 个参数。
- 类成员函数可以访问并修改类的属性、类构造函数中初始化的变量及实例对象变量。类成员函数也可以用于修改实例对象的状态，如设置实例变量的值、添加和删除实例变量等操作。
- 类成员函数被类实例对象进行调用的格式为“类实例对象.类成员函数”。

【示例 2.36】创建类与实例对象。

代码如下：

```
"""
这是一个定义类变量、类构造函数的示例：
"""
class StockClass:
    # 定义类变量 count，用于统计类调用次数
    count = 0

# 定义类构造函数__init__，利用类创建对象时需要传递参数 code
    def __init__(self, code):
        # 将外部传递的参数 name 存储于类函数内部变量
        self.code = code
        # 调用类变量 count，并修改类变量 count 的值
        StockClass.count += 1  # 计数器加 1
        print('类 StockClase 构造函数__init__被调用。')

# 定义记录函数 log
    def log(self):
        print(r'类内部调用类变量：本次实例类变量值为{}，本次实例对象参数为{}。'
.format(StockClass.count, self.code))
```

```
a = StockClass("600000.SS")
a.log()
print(r'类外部调用类变量：第 1 次调用类变量 StockClass.count 值为{}'
.format(StockClass.count))
b = StockClass("600519.SS")
b.log()
print(r'类外部调用类变量：第 2 次调用类变量 StockClass.count 值为{}'
.format(StockClass.count))
```

程序输出结果如下：

```
类 StockClase 构造函数__init__被调用。
类内部调用类变量：本次实例类变量值为 1，本次实例对象参数为 600000.SS。
类外部调用类变量：第 1 次调用类变量 StockClass.count 值为 1
类 StockClase 构造函数__init__被调用。
类内部调用类变量：本次实例类变量值为 2，本次实例对象参数为 600519.SS。
类外部调用类变量：第 2 次调用类变量 StockClass.count 值为 2
```

说明：

- 本例中设置了自定义类 StockClass 没有继承父类。
- 类 StockClass 设置了成员变量 count，用于计数类 StockClass 被实例化的次数。无论在类 StockClass 内部，还是在类 StockClass 外部，调用类成员变量 count 的函数均为 StockClass.count。
- 类 StockClass 设置了构造函数__init__，参数为 2 个，即 self 和 code。self 为类 StockClass 实例对象本身，code 为类 StockClass 实例化时必须传入的外部参数变量。构造函数__init__内部通过 self.code=code 将类实例参数传入类内部，以便类实例函数调用。
- 类 StockClass 内部定义了 log 函数，该类的成员函数仅有一个参数 self。
- 类 StockClass 在创建实例对象 a 时，传入了类__init__函数需要的初始化参数 code。
- 类 StockClass 的实例 a 与实例 b 可以执行内部函数 log，即 a.log 和 b.log 函数。

4．property属性

- property 是一种特殊的装饰器，它可以将类的函数转换为属性，从而使得这个属性看起来像一个普通的数据属性，但实际上它是通过方法函数计算得到的。例如，在定义类成员函数前一行加上“@property”，则该类成员函数即被转换为属性。
- 通过添加@property 修饰转换的属性是只读的，不能直接修改。如果需要修改该属性，则可以使用@property 的 setter 函数来实现（具体实现方法见下方代码）。
- 如果需要删除 property 属性，则可以使用@property 的 deleter 函数来实现。删除了该属性后，如果再通过实例对象调用 property 属性则会报错（具体实现方法见下方代码）。

【示例 2.37】设置 property 属性。

代码如下：

```
"""
这是一个@property 属性定义、修改、删除的示例：
"""
```

```
class StockClass:
    # 定义类构造函数__init__
    def __init__(self, price, volume):
        self._price = price
        self._volume = volume

    # 设置property属性cost_value，记录总金额
    @property
    def cost_value(self):
        return self._price*self._volume

    # 设置property属性cost_value为可修改状态
    @cost_value.setter
    def cost_value(self,price_volume):
        self._price,self._volume = price_volume

    # 设置propety属性cost_value为可删除状态
    @cost_value.deleter
    def cost_value(self):
        # 删除内部变量
        del self._price
        del self._volume

a = StockClass(10,200)
# 打印输出实例对象a的property属性cost_value
print(r'实例对象a.cost_value属性为：{}'.format(a.cost_value))
# 修改实例对象a的property属性cost_value
a.cost_value = (20,200)
# 打印输出修改后实例对象a的property属性cost_value
print(r'实例对象a.cost_value属性修改后为：{}'.format(a.cost_value))
# 删除实例对象a的cost_value属性
del a.cost_value
print(a.cost_value)
```

程序输出结果如下：

```
实例对象a.cost_value属性为：2000
实例对象a.cost_value属性修改后为：4000
报错信息AttributeError: 'StockClass' object has no attribute '_price'
```

说明：

- 类 StockClass 内部通过@property 定义了属性 cost_value，即在 def cost_value(self)前一行添加“@property”。此时，类属性 cost_value 仅能被读出，不能进行修改。
- 为了实现修改类属性 cost_value，首先需要重新定义 cost_value 函数的内容，再在重新定义的 cost_value 函数前一行添加“@cost_value.setter”。此时，类属性 cost_value 不仅可以被读出，而且可以被修改。注意，在 setter 函数的 cost_value(self,price_volume)参数中，price_volume 是包括两个数值的元组，分别用于传递给 cost_value 函数内部的 self._price 和 self._volume。
- a.cost_value = (20,200)实现重新设置 a.cost_value 的属性，此时的 a.cost_value 值为 4000。
- 为了实现删除类属性 cost_value，首先需要重新定义 cost_value 函数的内容，再在

重新定义的 cost_value 函数前一行添加“@cost_value.deleter”。此时，类属性 cost_value 不仅可以被读出、修改，而且可以被删除。本例通过 del a.cost_value 删除 a 的 cost_value 属性后，无法再通过 a.cost_value 调用此属性。

2.5.2　面向对象的封装

封装是指将类内部使用的变量设置为私有变量，防止外部进行修改。当外部调用类实例所包含的函数时，不能修改类内部的私有变量及私有函数。

1．私有变量

在 Python 中定义类时，可以通过在变量名前添加两个下画线即“__”（注意中间没有空格）将变量定义为私有变量。私有变量只能在类的内部访问，外部无法直接访问，这样可以确保变量不会被意外修改或访问。

- 如果一定要直接访问类中的私有变量，则可以通过“_类名__变量名”的形式来访问类内部的私有变量。但并不推荐这种做法，因为私有变量被设计为仅在类的内部使用，直接访问可能会导致代码不稳定和不可维护。
- 私有变量也可以通过@property 属性进行修改。

【示例 2.38】读取私有变量的方式。

代码如下：

```
"""
这是一个类私有变量读取的示例：
"""
class StockClass:
    # 类构造方法内有私有变量__price
    def __init__(self, price):
        self.__price = price

    # 通过类普通成员函数读取私有变量__price
    def get_price(self):
        return self.__price

    # 通过 property 属性获取私有变量__price
    @property
    def price(self):
        return self.__price

    # 通过属性 setter 函数修改私有变量__price
    @price.setter
    def price(self, price):
        self.__price = price

a = StockClass(10)
print(r'调用内部函数读取私有变量__price 为：{}'.format(a.get_price()))
print(r'直接读取私有变量__price 为：{}'.format(a._StockClass__price))
# 设置 property 属性 price，实现修改类私有变量__price
a.price = 20
```

```
print(r'修改后的私有变量__price 为：{}'.format(a.get_price()))
```

程序输出结果如下：

```
调用内部函数读取私有变量__price 为：10
直接读取私有变量__price 为：10
修改后的私有变量__price 为：20
```

说明：

- 本例中设置了私有变量__price，通过类内部成员函数 get_price 可以调取私有变量__price 的值。
- a._StockClass__price 直接读取实例 a 中的私有变量__price 的值。
- a.price=20 直接修改了 property 属性，同步修改私有变量__price 的值。

2．私有函数

在 Python 类中，可以通过在函数名前添加两个下画线__将函数定义为私有函数。

- 私有函数只能在类的内部访问，外部无法直接访问，这样可以确保方法不会被意外调用。
- 如果一定要直接访问类中的私有函数，则可以通过“_类名__私有函数”的形式来访问类内部的私有函数。但并不推荐这种做法，因为私有函数被设计为仅在类的内部使用，直接访问可能会导致代码不稳定和不可维护。

【示例 2.39】定义类私有函数。

代码如下：

```
class MyClass:
    def __init__(self):
        pass

    def get_private_method(self):
        # 调用类的私有函数
        self.__private_method()

    def __private_method(self):
        print("现在执行私有函数.")

# 实例化
my_object = MyClass()
print('调用实例的 get_private_method 函数:')
my_object.get_private_method()
print('---------------------------------')

print('实例直接调用类私有函数:')
my_object._MyClass__private_method()
```

程序输出结果如下：

```
调用实例的 get_private_method 函数:
现在执行私有函数.
---------------------------------
实例直接调用类私有函数:
现在执行私有函数.
```

2.5.3　面向对象的继承

在面向对象编程中，继承是一个重要的概念，它允许一个类（称为子类或派生类）继承另一个类（称为父类或基类）的属性和函数。

- 在 Python 中，继承是通过在子类定义时指定父类作为其基类来实现的。
- 子类可以使用父类的属性和函数，也可以重写父类的函数，但是子类的实例不能修改父类的属性和函数。在 Python 中，一个类可以同时继承多个类，这被称为多重继承。
- 子类可以通过调用 super 函数来调用父类的函数，这样可以避免在代码中出现硬编码的父类名称。
- 在继承中，如果子类和父类都定义了同名的属性或函数，则子类的属性或函数会覆盖父类的属性或函数。

【示例 2.40】子类继承父类。

代码如下：

```
"""
这是一个子类继承父类的示例：
"""
# 定义类 Stock
class Stock:
    # 构造函数，传入 code 参数
    def __init__(self, code):
        self.code = code

    def get_code(self):
        print(r"执行父类 get_code 函数，股票代码为：{}".format(self.code))

# 定义类 MyStock，继承父类 Stock
class MyStock(Stock):
    # 构造函数，传入 code、name 参数
    def __init__(self, code, name):
        # 调用 super 函数，引入父类构造函数
        super().__init__(code)
        self.name = name

    def get_code(self):
        super().get_code()
        print("执行子类 get_code 函数，股票代码为：{}，股票名称为：{}"
.format(self.code,self.name))

stock = MyStock("600000.SS", "浦发银行")
stock.get_code()
```

程序输出结果如下：

```
执行父类 get_code 函数，股票代码为：600000.SS
执行子类 get_code 函数，股票代码为：600000.SS，股票名称为：浦发银行
```

说明：

- super 函数代表父类，super().__init__(code)执行父类 Stock 构造函数，即将 code 参数传递给父类 Stock 的内部 self.code；而 MyStock 类继承了 Stock 类，因此 MyStock 内部也有 self.code 变量。
- super().get_code()执行父类 Stoc 的内部 get_code 函数。

2.5.4 面向对象的多态

多态是指类可以具有多种形态，即子类可以与父类有不同的表现形态，子类可以重新构造父类中出现的函数功能。

【示例 2.41】类的多态。

代码如下：

```
"""
这是一个类多态性的示例：
"""
# 定义类 Stock
class Stock:
    # 构造函数，传入 code 参数
    def __init__(self, code):
        self.code = code

    def get_code(self):
        print(r"执行父类 get_code 函数，股票代码为：{}".format(self.code))

# 定义类 MyStock，继承父类 Stock
class MyStock(Stock):
    # 构造函数，传入 code、name 参数
    def __init__(self, code, name):
        # 调用 super 函数，引入父类构造函数
        super().__init__(code)
        self.name = name

    def get_code(self):
        print("执行 MyStock，{}代码为{}。".format(self.name, self.code))

# 定义类 YourStock，继承父类 Stock
class YourStock(Stock):
    # 构造函数，传入 code、name 参数
    def __init__(self, code, name):
        # 调用 super 函数，引入父类构造函数
        super().__init__(code)
        self.name = name

    def get_code(self):
        print("执行 YourStock，股票代码为：{}，股票名称为：{}。"
.format(self.code,self.name))

my_stock = MyStock("600000.SS", "浦发银行")
my_stock.get_code()
```

```
your_stock = YourStock("000001.SZ", "平安银行")
your_stock.get_code()
```

程序输出结果如下：

```
执行 MyStock，浦发银行代码为 600000.SS。
执行 YourStock，股票代码为：000001.SZ，股票名称为：平安银行。
```

说明：

- 子类 MyStock 与子类 YourStock 均继承 Stock 类。
- 子类 MyStock 与子类 YourStock 均定义了内部函数 get_code，当实例 my_stock 对象与实例 your_stock 对象调用 get_code 函数时，分别执行对应类的成员函数。

2.6　模 块 应 用

在使用 Python 进行金融量化分析过程中，通常都会调用外部或内置的各类模块，这些模块一般都是以“.py”结尾的模块文件的形式进行保存。Python 提供了简单、方便地调用这些模块文件的函数。

2.6.1　模块的安装、卸载与调用

Python 中有部分模块为系统内置的模块，如 time 模块、datetime 模块和 os 模块等，这些模块在安装 Python 后就已同步完成安装，无须另行单独安装。很多可以实现特定功能的模块，如金融量化分析经常要用到的 Pandas 模块、NumPy 模块和 Matplotlib 模块等都不属于内置模块，这些模块需要执行安装操作后才可调用。本节仅以 Windows 操作系统环境为例，介绍 Python 模块的安装方法。

Python 模块的安装通常有两种方式，一种是通过 pip 命令自动安装，另一种是通过下载 whl 文件，再安装该 whl 文件。

1．pip命令自动安装

在进行金融量化分析时，我们经常使用 Anaconda 作为 Python 的执行环境，可以通过打开 Anaconda Prompt 命令窗口（Windows 系统自带的 dos 命令行窗口也可以），在命令窗口的命令行中输入“pip install 模块名称”。

【示例 2.42】安装 akshare 模块。

代码如下：

```
"""
安装 akshare 模块（akshare 模块提供了下载各类金融市场行情数据的方法）
"""
pip install akshare
# 从清华大学源中使用 pip 方法安装 akshare 模块
pip install -i https: //pypi.tuna.tsinghua.edu.cn/simple akshare
```

说明：

使用 pip 安装 akshare 模块时，可以选择不同的下载源，具体如下：

- 清华大学的下载源为 https://pypi.tuna.tsinghua.edu.cn/simple。
- 阿里云的下载源为 http://mirrors.aliyun.com/pypi/simple/。
- 中国科技大学的下载源为 https://pypi.mirrors.ustc.edu.cn/simple/。
- 豆瓣的下载源为 http://pypi.douban.com/simple/。

2．手工安装“.whl”文件

在安装模块的过程中会遇到各种报错，此时可以将适合本地系统的模块安装文件（通常以“.whl”为后缀名的文件）下载到本地，然后通过“pip install 本地模块文件”命令进行安装。

以安装 TA-Lib 模块为例（TA-Lib 模块提供了金融量化分析所需的各类技术指标的下载方法），具体操作步骤如下：

（1）登录网址：https://www.lfd.uci.edu/~gohlke/Pythonlibs/。

（2）在网页中找到 TA-Lib 栏目，选择适合本地环境的版本（版本说明：cp310 表示 Python 3.10 版，win_amd64 表示 Windows 64 位操作系统），如 TA_Lib‑0.4.24‑cp310‑cp310‑win_amd64.whl。

（3）将适合本地环境的“.whl”安装文件保存至本地环境，如本地的 Python 版本为 3.10，硬件是 64 位，可以下载 TA_Lib‑0.4.24‑cp310‑cp310‑win_amd64.whl 至 D 盘根目录（“D：/”）。

（4）在 Anaconda Prompt 命令窗口（或 Windows 系统自带的 Dos 命令窗口）中输入命令 pip install D：/ TA_Lib‑0.4.24‑cp310‑cp310‑win_amd64.whl。

3．模块的卸载

当需要卸载模块时，在 Anaconda Prompt 命令窗口（或 Windows 系统自带的 Dos 命令窗口）中输入命令“pip uninstall 模块名称”即可。

4．模块的调用

有两种方法可以调用 Python 模块，第一种是直接使用 import 关键字导入模块，第二种是使用 from…import 语句导入模块，具体语法如下：

```
# 直接导入模块
import 模块名称
# 从模块中导入部分函数
from 模块名称 import 函数
```

在导入模块后，可使用下面的方式调用模块内函数或类：

```
# 调用模块中的函数
模块名称.模块内部函数()
# 调用模块中的类
模块名称.模块内部类名称()
```

Python 允许在模块名称中使用别名代替（如 import pandas as pd），具体语法如下：

```
# 用别名代替模块名
import 模块名称 as 模块别名
# 调用函数
模块别名.模块内部函数名称()
# 调用类
模块别名.模块内部类名称()
```

【示例 2.43】调用 os 模块函数。

代码如下：

```
# 引入 os 模块
import os
# 调用 os 模块的 listdir 函数，读取“D:/”盘下的全部文件名称并存入列表变量
file_name_list = os.listdir('D:/')
```

2.6.2　Python 内置模块示例 1：datetime 模块

datetime 模块比 time 模块可以更为便捷地处理与日期、时间相关的操作。datetime 模块定义了 5 个类，即 date 类（处理日期的类）、time 类（处理时间的类）、datetime 类（处理时间日期的类）、timedelta 类（处理两个 datetime 对象的差值的类）与 tzinfo 类（处理时区相关信息的类）。datetime 模块处理时间相关问题的函数主要包括以下几个方面。

- datetime.datetime(year, month, day[, hour[, minute[, second[, microsecond[, tzinfo]]]]])：创建一个表示特定日期和时间的 datetime 对象。
- datetime.date(year, month, day)：创建一个表示特定日期的 date 对象。
- datetime.time([hour[, minute[, second[, microsecond[, tzinfo]]]]])：创建一个表示特定时间的 time 对象。
- datetime.timedelta([days[, seconds[, microseconds[, milliseconds[, minutes[, hours[, weeks]]]]]]])：表示时间间隔的对象，可以在 datetime 和 date 对象之间进行算术运算。
- datetime.datetime.now([tz])：返回当前日期和时间的 datetime 对象。
- datetime.datetime.utcnow：返回当前 UTC 日期和时间的 datetime 对象。
- datetime.datetime.fromtimestamp(timestamp[, tz])：根据时间戳创建一个 datetime 对象。
- datetime.datetime.strptime(date_string, format)：将一个字符串解析为 datetime 对象。
- datetime.datetime.strftime(format)：将 datetime 对象格式化为字符串。
- datetime.date.weekday：返回日期的星期几，星期一为 0，星期日为 6。

【示例 2.44】datetime 模块的应用 1。

代码如下：

```
from datetime import date, time, datetime, timedelta, timezone

# date 类的 today 函数
today = date.today()
```

```
print(f"1---date.today(): 今天的日期是{today}")

# 生成日期变量（仅含有年月日）
specific_date = date(2022, 1, 1)
print(f"2---date(2022, 1, 1): 特定的日期是{specific_date}")

# 返回日期为周几
day_of_week = today.weekday()
print(f"3---today.weekday(): 今天是一周中的第 {day_of_week + 1} 天")

# 使用 time 类函数生成时间变量（包含小时、分钟、秒）
current_time = time(14, 30, 45)
print(f"4---time(14, 30, 45): 特定的时间是{current_time}")

# 使用 datetime 类的 now 函数获取当前时间
now = datetime.now()
print(f"5---datetime.now(): 现在的日期和时间是{now}")

# 构建 datetime 实例（包含年、月、日、时、分、秒）
specific_datetime = datetime(2022, 1, 1, 14, 30, 45)
print(f"6---datetime(2022, 1, 1, 14, 30, 45): 特定的日期和时间是
{specific_datetime}")

# 将 datetime 对象按指定格式转换成文本形式
specific_datetime_format = specific_datetime.strftime(format='%Y/%m/%d')
print(f"7---datetime 实例.strftime(formate): 将 datetime 对象格式化输出
{specific_datetime_format}")

# 将指定格式的文本转换为 datetime 对象
dt_str = '2023-04-22 15:30:00'
dt = datetime.strptime(dt_str, '%Y-%m-%d %H:%M:%S')
print(f"8---datetime.strptime(text,formate): 将文字转成 datetiem 对象{dt}")

# 使用 timedelta 类函数生成新的 datetime 变量
future_date = today + timedelta(days=10)
print(f"9---（today + timedelta(days=10)）：十天后的日期是：{future_date}")

# datetime 类型变量之间可以进行加减运算
date_diff = future_date - today
print(f"10---（future_date - today）：两个日期之间相差 {date_diff.days} 天")

# 通过 timezone 类的 utc 属性显示时区信息
UTC = timezone.utc
print(f"11---timezone.utc: UTC 时区：{UTC}")

# timezone 函数也可以通过使用 timedelta 函数进行时区变化
new_timezone = timezone(timedelta(hours=8))
print(f"12---timezone(timedelta(hours=8)): 新时区（UTC+8）：{new_timezone}")

# 生成特定时区的当前时间
local_datetime = datetime.now(new_timezone)
print(f"13---datetime.now(new_timezone): 特定时区下的当前日期和时间：
{local_datetime}")
```

程序输出结果如下：

```
1---date.today()：今天的日期是 2023-04-22
2---date(2022, 1, 1)：特定的日期是 2022-01-01
3---today.weekday()：今天是一周中的第 6 天
4---time(14, 30, 45)：特定的时间是 14:30:45
5---datetime.now()：现在的日期和时间是 2023-04-22 08:47:55.672203
6---datetime(2022, 1, 1, 14, 30, 45)：特定的日期和时间是 2022-01-01 14:30:45
7---datetime 实例.strftime(formate)：将 datetime 对象格式化输出 2022/01/01
8---datetime.strptime(text,formate)：将文字转成 datetiem 对象 2023-04-22
15:30:00
9---（today + timedelta(days=10)）：十天后的日期是：2023-05-02
10---（future_date - today）：两个日期之间相差 10 天
11---timezone.utc：UTC 时区：UTC
12---timezone(timedelta(hours=8))：新时区（UTC+8）：UTC+08:00
13---datetime.now(new_timezone)：特定时区下的当前日期和时间：2023-04-22
08:47:55.673199+08:00
```

说明：

在 Python 中，datetime 类型的数据可以进行加减运算，但需要满足以下条件。

- 参与运算的两个 datetime 数据类型必须是同一类型，例如都是 datetime.date 类型或 datetime.datetime 类型。
- 两个 datetime 数据类型相减得到的结果是一个 datetime.timedelta 类型的对象，表示两个时间的时间差。
- datetime 数据类型可以与 datetime.timedelta 类型进行加减运算，从而得到一个新的 datetime 数据类型。

【示例 2.45】datetime 模块的应用 2。

代码如下：

```
import datetime

# 创建一个 datetime.date 类型的对象，表示今天的日期
today = datetime.date.today()
print("今天的日期是：", today)

# 创建一个 datetime.timedelta 类型的对象，表示一天的时间差
one_day = datetime.timedelta(days=1)

# 计算明天的日期
tomorrow = today + one_day
print("明天的日期是：", tomorrow)

# 创建一个 datetime.datetime 类型的对象，表示当前日期和时间
now = datetime.datetime.now()
print("当前日期和时间是：", now)

# 创建一个 datetime.timedelta 类型的对象，表示一小时的时间差
one_hour = datetime.timedelta(hours=1)
# 计算一小时后的时间
one_hour_later = now + one_hour
print("一小时后的时间是：", one_hour_later)
```

程序输出结果如下：

```
今天的日期是： 2023-04-27
明天的日期是： 2023-04-28
当前日期和时间是： 2023-04-27 20:22:11.654940
一小时后的时间是： 2023-04-27 21:22:11.654940
```

2.6.3 Python 内置模块示例 2：os 模块

os 模块是 Python 内置的一个标准模块，它提供了访问操作系统文件相关功能的接口，可以方便地进行文件和目录操作以及设置文件的权限等。os 模块的主要函数包括以下几点。

- os.name：返回当前的操作系统名称。
- os.getcwd：返回当前工作目录的路径。
- os.chdir(path)：改变当前工作目录为指定的路径。
- os.listdir(path='.')：返回指定目录下的所有文件和目录的名称列表。
- os.mkdir(path[, mode])：创建一个新的目录。
- os.makedirs(name[, mode])：递归创建多层目录。
- os.rmdir(path)：删除一个空目录。
- os.removedirs(path)：递归删除多层空目录。
- os.remove(path)：删除一个文件。
- os.rename(src, dst)：重命名或移动一个文件或目录。
- os.path.abspath(path)：返回指定路径的绝对路径。
- os.path.exists(path)：判断指定路径是否存在。
- os.path.isdir(path)：判断指定路径是否为目录。
- os.path.isfile(path)：判断指定路径是否为文件。
- os.path.join(path1[, path2[, ...]])：将多个路径组合成一个完整的路径。

【示例 2.46】os 模块的应用。

代码如下：

```
import os

# 获取当前的工作目录
current_dir = os.getcwd()
print(f'当前工作目录：{current_dir}')
# 切换工作目录
os.chdir('d:/backtrader_test/')
current_dir = os.getcwd()
print(f'当前工作目录：{current_dir}')
# 列出目录中的文件和子目录
dir_contents = os.listdir(current_dir)
print(f'当前目录中的文件和子目录：{dir_contents}')
```

程序输出结果如下：

```
当前工作目录：d:\backtrader_test
当前工作目录：d:\backtrader_test
当前目录中的文件和子目录：['.idea', 'backtrader_indicators.py']
```

2.7　本 章 小 结

本章介绍了 Python 的基本语法与基本应用。实际上，Python 语法内容非常庞杂，需要讲解的内容非常多，但是，鉴于本书是聚焦金融量化分析及交易策略开发的，因此，本章仅介绍了开发金融量化交易策略过程中所必须具备的 Python 基础知识，如变量、逻辑判断、函数、类、模块等。只要掌握了这些基础知识，再结合后面章节中的 Pandas 和 NumPy 库的学习，读者就可以在不借助任何第三方回测平台的条件下独立完成开发量化交易策略回测的工作。

2.8　思　考　题

1．列表变量有哪些函数？这些函数的应用场景是什么？
2．字典变量有哪些函数？这些函数的应用场景是什么？
3．列表与元组的相似点和不同点是什么？二者的应用场景有何差异？