HEC MONTRÉAL

Empirical Finance

Modern Research with Python

Vincent Grégoire, CFA, Ph.D.

January 2026

Table of contents

About This Book

This book is aimed at students in the MSc in Finance program at HEC Montréal taking the course *MATH 60230 Empirical Finance*. I make it publicly available for anyone interested in learning Python for finance, but some examples and explanations are tailored to the HEC Montréal context.

The main objective of this book is to offer a thorough and accessible practical guide to empirical finance research using Python. To achieve this goal, I cover statistical and econometric techniques used in empirical finance, with a focus on practical applications using Python. I believe that learning by doing is the most effective way to master new skills. Thus, I present real-world scenarios and datasets, enabling you to see the power and efficacy of these techniques in action.

No prior programming experience is required. If you are already familiar with Python, I still recommend you at least skim Part I because I propose not only an introduction to Python, but an introduction to a modern workflow for data analysis with Python.

My goal is that by the end of this book, you will have an advanced understanding of the main econometric techniques used in empirical finance and a solid grounding in modern Python programming for data analysis. I look forward to guiding you through this exciting journey into the world of empirical finance, statistics, econometrics, and Python programming.

> **Work in Progress**
>
> This book is a work in progress. I am constantly adding new content and refining existing content. If you have any suggestions or feedback, please reach out at `vincent.gregoire@hec.ca`.

Structure

This book is organized into seven parts plus appendices, each designed to build upon the knowledge from the previous section, ultimately guiding you to a robust understanding of empirical finance using Python.

Part I: Python Fundamentals, Environment, and Best Practices

The first part of the book is devoted to familiarizing you with Python and setting up the necessary coding environment. We begin with instructions on installing Python and understanding its basic syntax. We then introduce various tools that are part of the programmer's toolbox, including the terminal, VS Code, Git, and GitHub. You will learn about managing Python environments with uv, writing clean and well-documented code, testing your code, and object-oriented programming concepts.

Part II: Working with Data

In the second part, we focus on data manipulation and analysis. You will learn how to load data from various sources and formats, work with DataFrames using pandas and Polars, clean and transform data, structure datasets for analysis, merge multiple data sources, and reshape data between wide and long formats.

Part III: Visualization and Research Output

The third part covers how to communicate your findings effectively. We explore data visualization with matplotlib and seaborn, creating publication-quality tables for regression results and summary statistics, and using Quarto to create reproducible research documents that combine code, text, and results.

Part IV: Statistical Foundations

In the fourth part, we dive into the statistical foundations needed for empirical finance. We cover numerical computing with NumPy, probability distributions and random number generation, and descriptive statistics for financial data.

Part V: Regression Methods

The fifth part focuses on econometric techniques central to empirical finance research. We cover linear regression with statsmodels, panel data methods for working with firm-time observations, and instrumental variables estimation for addressing endogeneity.

Part VI: Machine Learning and Artificial Intelligence in Research

The sixth part introduces modern AI and machine learning tools relevant to finance research. We discuss how to use AI assistants effectively for coding and research, machine learning techniques for prediction and classification, and natural language processing methods for analyzing textual data.

Part VII: Reproducibility and Replication

The final part addresses the critical importance of reproducibility in research. We cover best practices for organizing research projects, managing dependencies, and ensuring that your results can be replicated by others.

Learning Approach

The learning approach adopted in this book is designed to be practical and closely linked with the real-world challenges encountered in empirical finance. My philosophy is grounded in the belief that the best way to learn is by doing, especially when it comes to mastering complex concepts like econometrics and programming.

YouTube Video Tutorials

Throughout the book, I provide links to YouTube videos that offer alternative explanations of the concepts covered in the chapters. These videos are not meant to replace the book, but rather to provide additional perspectives and clarifications. Some of these videos are created by me and are available on my YouTube channel, Vincent Codes Finance.

Tech Stack

This book is structured around a tech stack formed by a specific set of tools that has been carefully chosen based on their wide adoption, robustness, versatility, and compatibility with each other. While alternative tools exist and may be equally capable, the book takes an opinionated approach, focusing on this particular stack for clarity and consistency. It's worth noting that the concepts and techniques covered in this book can be applied with other tools as well, but the specific examples and code use the following:

- Python 3.14 (released in October 2025)
- uv for managing Python versions and environments
- Visual Studio Code for writing code
- Git and GitHub for version control and collaboration
- Claude, ChatGPT, and Microsoft Copilot
- Claude Code, OpenAI Codex, and GitHub Copilot for coding assistance
- Quarto for writing technical content

Use of AI

This book was written with substantial assistance from AI tools, primarily ChatGPT and Claude Code. AI was used for all aspects of the book's creation, including idea generation, creating outlines, drafting content, proofreading, and generating code examples. This reflects the modern reality of software development and technical writing, where AI assistants have become valuable collaborators.

However, all content has been reviewed and edited by a human. I take full responsibility for the accuracy and quality of the material presented. Any errors or omissions remain my responsibility.

About the Author

I'm Vincent Grégoire, CFA, a Professor of Finance at HEC Montréal and the Canada Research Chair in Finance and Technology. I teach empirical finance with a strong emphasis on Python-based data analysis. I earned a Ph.D. in Finance from the University of British Columbia, along with degrees in Computer Engineering and Financial Engineering from Université Laval, and previously served as Chief Data Scientist at Berkindale Analytics, a fintech startup.

My work focuses on how information is produced, processed, and priced in financial markets. I study market structure through the lens of big data, machine learning, algorithmic trading, and cybersecurity, with an emphasis on methods that actually scale outside toy examples.

Acknowledgments

I am grateful to Charles Martineau and Saad Ali Khan for their feedback and suggestions on the book.

Part I.

Python

This part introduces Python, an open-source, high-level programming language that has become indispensable for empirical finance. I aim to provide readers with a foundational understanding of Python's capabilities and how to leverage it for financial research. Starting from the basics, we will explore why Python is the go-to tool for researchers in finance.

What is Python?

Python is a versatile and user-friendly programming language that emphasizes readability and simplicity. Its design philosophy promotes code that is easy to write and understand, making it an excellent choice for both beginners and experienced programmers. The name Python also refers to the interpreter, the program that runs Python code.

Why Python for Finance?

Python has been widely adopted by the finance industry and finance researchers for several compelling reasons:

Open source

Python is free and open-source. Yes, that means you can use it for free, but open-source is much more than that. Python is distributed under the Python Software Foundation License, which is a permissive license that allows you to use, modify, and distribute the code and derived works based on it. This has led to a vibrant ecosystem of libraries and tools that are freely available to all, and to private forks of Python that are used internally by large financial institutions know collectively as bank Python.

Ease of learning

Python's syntax is designed to be intuitive and human-readable, making it an accessible language for beginners and experts alike. This simplicity allows finance professionals—many of whom may not have a computer science background—to quickly learn Python and apply it to their work. Python code often reads like plain English, reducing the learning curve and enabling users to focus on problem-solving rather than struggling with the syntax.

For academic researchers, this ease of learning means that Python can be introduced in undergraduate or graduate programs with minimal friction. Students can rapidly transition from learning the basics of the language to applying it in real-world financial scenarios, such as data analysis, statistical modeling, and portfolio optimization.

Powerful

Python is exceptionally powerful due to its extensive library ecosystem. Libraries like NumPy, SciPy, and statsmodels provide robust tools for numerical and statistical computations, while pandas and polars facilitate data manipulation and analysis. These capabilities make Python an ideal choice for tasks ranging from simple data cleaning to complex econometric modeling.

In addition to its computational capabilities, Python integrates seamlessly with other programming languages and platforms, enabling finance practitioners to incorporate Python into larger, multi-language workflows. For example, it can call high-performance code written in C++ or Rust, interact with databases through SQL, or interface with scentific languages like R or Julia. This power and flexibility ensure that Python remains suitable for both small-scale analyses and enterprise-level financial systems.

Versatile

Python's versatility allows it to handle a wide range of tasks, making it a one-stop solution for financial workflows. Analysts can use Python for tasks such as data acquisition from APIs or through web scraping, performing statistical analyses, creating visualizations, and even building predictive models using machine learning libraries like scikit-learn.

Widely-used

In 2024, Python surpassed Javascript to become the most widely used programming language in the world according to GitHub's Octoverse. This rise in use is attributed to the growing importance of AI and data science, for which Python is the most popular language.

This popularity ensures that Python skills are highly transferable and in demand, making it a valuable asset for finance professionals. Large financial institutions, such as JPMorgan Chase and Goldman Sachs, use Python extensively for data analysis, trading algorithms, and risk modeling. Financial data providers such as LSEG (formerly Refinitiv and Thomson Reuters) and WRDS, an academinc data provider, offer Python-based platforms for accessing and analyzing financial data. Python skills are now a must-have for finance professionals, so much so that the CFA Institute has added Python practical skills to its 2024 curriculum.

Other languages

Python is not the only game in town. R and Stata offer better capabilities for econometric and statistical modeling, and are also widely used in academic research. Julia and Matlab offer better performance for numerical computing, and C++ and Rust are the languages of choice for performance-critical parts of financial applications. Finally, SAS and many database softwares use a variant of SQL, while some use their own proprietary language like q for kdb+ which is a staple of high-frequency trading firms. Finally, OCaml is the language of choice at Jane Street, a large quantitative trading firm. Overall, each language has its strengths and weaknesses, and the choice of language depends on the specific task at hand, but Python is a very good choice for most tasks.

Components of the Python Ecosystem

The Python ecosystem consists of various tools and components that make it a powerful platform for data analysis. This section introduces the key elements of the ecosystem and their roles in creating efficient workflows.

Python interpreter

Python is an interpreted language, meaning that your code is executed by an interpreter when you run it rather than compiled ahead of time to an executable. The Python interpreter is responsiable for executing your Python code. It comes in various implementations, with the most common being **CPython**, the default implementation distributed with official Python releases, which is the one we will use in this book. Other variants include PyPy, a just-in-time (JIT) compiled interpreter that enhances performance for specific tasks, and Pyodide, a port of CPython to WebAssembly that allows Python to run in web browsers.

Python libraries

The base Python language is simple, but it is extended through a large ecosystem of libraries. Python comes with a large standard library, that includes many features such as file input/output, basic data structures, and mathematical functions. However, most Python programs will leverage additional libraries. These libraries are pre-written modules that extend Python's functionality. For empirical finance, some key libraries include:

- **pandas and polars:** For data manipulation and analysis.
- **NumPy and SciPy:** For numerical computations.
- **matplotlib and seaborn:** For data visualization.
- **statsmodels and linearmodels:** For econometric modeling.
- **scikit-learn:** For machine learning.

These libraries form the backbone of financial analysis in Python, enabling everything from basic calculations to complex statistical modeling.

> ⚠ Warning
>
> Python libraries are published on PyPI, the Python Package Index. **Anyone** can publish a library to PyPI, so it is important to check the library's reputation before using it. Always keep in mind that librairies contain code that will be executed on your computer, so they can contain malware. Well-known libraries are less likely to contain vulnerabilities, but, as with any unreviewed software, there is always a risk. We will discuss security best practices in more detail in future chapters.

Environment and package management

Python versions are updated frequently.[1] Libraries follow their own release cycles and are not always updated at the same time as the Python interpreter. Some libraries depend on specific versions of other libraries, so a chain of dependencies may need to be updated. To complicate matters, updates to libraries may break your code. This means that code that worked last month may not work this month if you always use the latest version of everything.

[1]The latest version at the time of writing is 3.14. Since 2018, Python releases have been annual in October.

To minimize these issues, the best practice is to create a *virtual environment* for each project. This ensures that the versions of the Python interpreter and libraries are fixed and consistent for each project, and that you can easily update the libraries for a project without affecting your other projects.

There are several tools for managing Python environments. Python comes with venv, a built-in module for creating virtual environments, and pip, a package manager for installing and updating libraries. A popular alternative for data science projects is conda, part of the Anaconda distribution.[2] Another popular tool is poetry. Conda and poetry act as both environment and package manager.

In this book, we will use uv, a tool that replaces venv, pip, and a suite of other tools. It brings a lot of modern features to the table, such as a lockfile to ensure reproducibility, ability to install and manage Python versions, and more. For me, its main advantage is its increadible speed which is in part due to an advanced caching mechanism, the use of hardlinks to avoid keeping multiple copies of each library, and a very fast dependancy resolver.[3]

Integrated Development Environments (IDEs)

IDEs streamline coding by providing features such as syntax highlighting, debugging tools, and other tools to make your life easier. In this book, we will use Visual Studio Code (VS Code), an open-source code editor by Microsoft that is very popular in the data science community. It is not Python-specific, but it integrates seamlessly with Python through extensions. Because it is open-source, many forks have been created, such as Cursor, which adds a powerful AI engine to the editor, and Positron, a data science-oriented fork by Posit, the company behind RStudio and Quarto (still in beta at the time of writing).

Other popular IDEs include PyCharm by JetBrains (commercial, but free for students and academics) and neovim, a terminal-based text editor with a steep learning curve that is popular among developers for its extensibility. Finally, Spyder, is an open-source IDE that was very popular in the Python scientific community, but has since been eclipsed by VS Code.

Notebooks

Notebooks, such as Jupyter Notebooks, are interactive environments where code, text, and visualizations can coexist. They have significant drawbacks for their use in robust, replicable research, but are nonetheless very popular in data science because of their simplicity. VS Code supports Jupyter notebooks natively with an extension. We will notebooks them in more details in this book, along with their shortcomings and ways to mitigate them. marimo is a new Python notebook interface that aims to address some of the issues with Jupyter Notebooks. While it has a long way to go to overtake Jupyter in the data science community, it shows a lot of promise.

[2] While conda is open-source, it uses by default the Anaconda Repository, which requires a paid subscription under some conditions. At the time of writing, it is free for academic use.

[3] The main task of a package manager is to resolve dependencies, i.e. to figure out which versions of libraries need to be installed to satisfy the dependencies of a project. This is a very complex task (NP-hard) that requires a lot of computation and heuristics.

1. Installing Python

In this chapter, I cover installing Python 3.14 and the related tools for a complete coding environment.

> **Python in the cloud**
>
> Not sure if you want to install Python on your computer? No worries, you can use Python in the cloud. I recommend GitHub Codespaces, which allows you to run in your browser an environment almost identical to the one you would get from a local installation. You can find the instructions at the end of this post.

> **Videos**
>
> The following videos provide guided instructions, for Mac and Windows respectively, of the steps provided in this chapter.
>
> - Install Python 3.13 on macOS for Data Science
> - Install Python 3.13 on Windows for Data Science

1.1. What you need for a complete Python environment

The most common way to use Python is to install it locally on your computer. The instructions below will guide you through the process of installing the following tools:

- **uv**: A package manager for Python. I use it to manage the external libraries used in projects. uv makes it easy to install and update libraries on a per-project basis, and to make sure all collaborators use the same library version.
- **Python**: The Python interpreter, which allows you to run Python code. We will install multiple versions using uv.
- **Visual Studio Code**: Visual Studio Code is a free source code editor made by Microsoft. Features include support for debugging, syntax highlighting and intelligent code completion. Users can install extensions that add additional functionality.

We will also install the following tools that are not required to run Python code, but are useful when working on projects with code:

- **Git and GitHub**: I use Git to manage my code and GitHub to host my code online and collaborate with others. Git is a version control system that tracks code changes and keeps a full history of changes.

GitHub is a website that hosts Git repositories and provides additional features for collaboration such as issue tracking and pull requests.

uv vs Poetry and Anaconda

Most Python projects use external libraries. For example, we use the pandas library for data analysis. To manage these libraries, we need a package manager. I now recommend using uv instead of Poetry (also very good) and Anaconda. Anaconda was my package manager of choice for many years and it remains very popular, but like many I eventually switched to poetry and more recently to uv. Overall, uv brings many nice features, but the two main reasons why I settled on uv is that it is very fast and it lets you easily install and use multiple Python versions. While speed might seem a minor concern for a package manager, anyone who has spent minutes (plural) waiting for Anaconda to create a virtual environment and install all the dependencies will understand.

1.2. Installation

All the tools used in this book are available on Mac, Windows, and Linux. I focus on installation instructions for Mac and Windows because to be honest, if you're using Linux, you probably already know how to install Python and other tools.

uv

Installation instructions can be found here.

macOS

The simplest way to install uv on macOS is with their install script.[1] First, you need to open the Terminal app. You can find it in the `Applications/Utilities` folder, or by using Spotlight (press `Cmd+Space` and type `Terminal`). Then run the following script:[2]

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

Alternatively, you can also install it using Homebrew[3]:

```
brew install uv
```

To check that uv is properly installed, run the following command in the Terminal app:

```
uv --version
```

[3] See the uv website for more details and troubleshooting advice.

[3] `curl` is a program that will download the script, and `|`, the pipe operator, will take the result (the downloaded script) and pass it as input to `sh`, which will execute the script.

[3] Homebrew is a package manager for macOS that allows you to install and update software from the command line. It simplifies the installation process and makes it easy to keep your software up-to-date.

Windows

The simplest way to install uv on Windows is with their install script.[4] First, you need to open Powershell. Then, run the following script:

```
powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

Alternatively, you can also install it using WinGet[5]:

```
winget install --id=astral-sh.uv  -e
```

To check that uv is properly installed, run the following command in Powershell (you may have to close and re-open Powershell):

```
uv --version
```

Note: If this does not work but you had a successful installation messsage, you may have to restart your computer for the `uv` command to be available. You may have to manually add the path to uv in your environment variables. Do to so, you first need to figure out where uv was installed on your computer. It will tell you after installing, but the default is `C:\Users\YOURUSERNAME\.local\bin`. Once you have that path, add it to enviroment variables (`Control panel->Edit Environment variables`).

Visual Studio Code

Download Visual Studio Code from code.visualstudio.com.

Git and GitHub

macOS

Git might already be installed on your Mac as a command-line tool if you have installed the Xcode tools. If not, you can get the official installer. You can also use Git directly in VS Code, or using a GUI client such as GitHub Desktop. I prefer to use the VS Code integration or the command-line tool, but many beginners prefer to use GitHub Desktop.
Git is probably already installed on Linux as a command-line tool. You can also use Git directly in VS Code, or using a GUI client such as GitHub Desktop. I prefer to use the VS Code integration or the command-line tool, but many beginners prefer to use GitHub Desktop.

Windows

To use Git on Windows, you need to install the Git client, which is a command-line tool.
You can also use Git directly in VS Code, or using a GUI client such as GitHub Desktop, **but you need to first install the Git client**. I prefer to use the VS Code integration or the command-line tool, but

[5] See the uv website for more details and troubleshooting advice.
[5] WinGet is a package manager for Windows that allows you to install and update software from the command line. It simplifies the installation process and makes it easy to keep your software up-to-date.

> many beginners prefer to use GitHub Desktop.

1.3. Creating a sandbox environment

The recommended way to work with environments in Python is to have unique enviromnents for each project. However, not everything is a project, so I like to have a "sandbox" environment with all the libraries I use regularly. That way, when I want to try something quickly like reading a CSV file to explore it, I have this sandbox project ready to go. It used to be common to install these libraries in the default (or base) environment, but it can lead to issues when updating packages, so many Python distribution now lock the default environment to prevent you from installing packages.

For my sandbox environment, I will want at least the following libraries:

- pandas: Data analysis library
- numpy: Numerical computing library
- scipy: Scientific computing library
- matplotlib: Plotting library
- seaborn: Plotting library
- statsmodels: Statistical models
- scikit-learn: Machine learning library
- linearmodels: Linear models for Python
- pyarrow: Library for working with parquet files
- jupyter: for Jupyter notebooks and the VS Code Python interactive window
- pytest: Testing framework

To create this sandbox environment, I will use uv. First, I need to create a new directory for the environment. I will call it `sandbox`, but you can name name it whatever you want.[6] Then, I need to create a new project in this directory:

On macOS or Linux:

```
mkdir ~/Documents/sandbox
cd ~/Documents/sandbox
uv init
```

On Windows, first create a folder named `sandbox` where you want on your computer. Then, from Windows Explorer, open the folder in PowerShell using `File->Open Windows PowerShell`. You can then initialize your environment using uv:

```
uv init
```

[6] I avoid spaces and special characters as they can sometimes cause trouble.

This creates a `pyproject.toml` file in the `sandbox` directory. This file contains the list of dependencies for the project (which will be empty for now).

Once the project is created, you can add the dependencies:

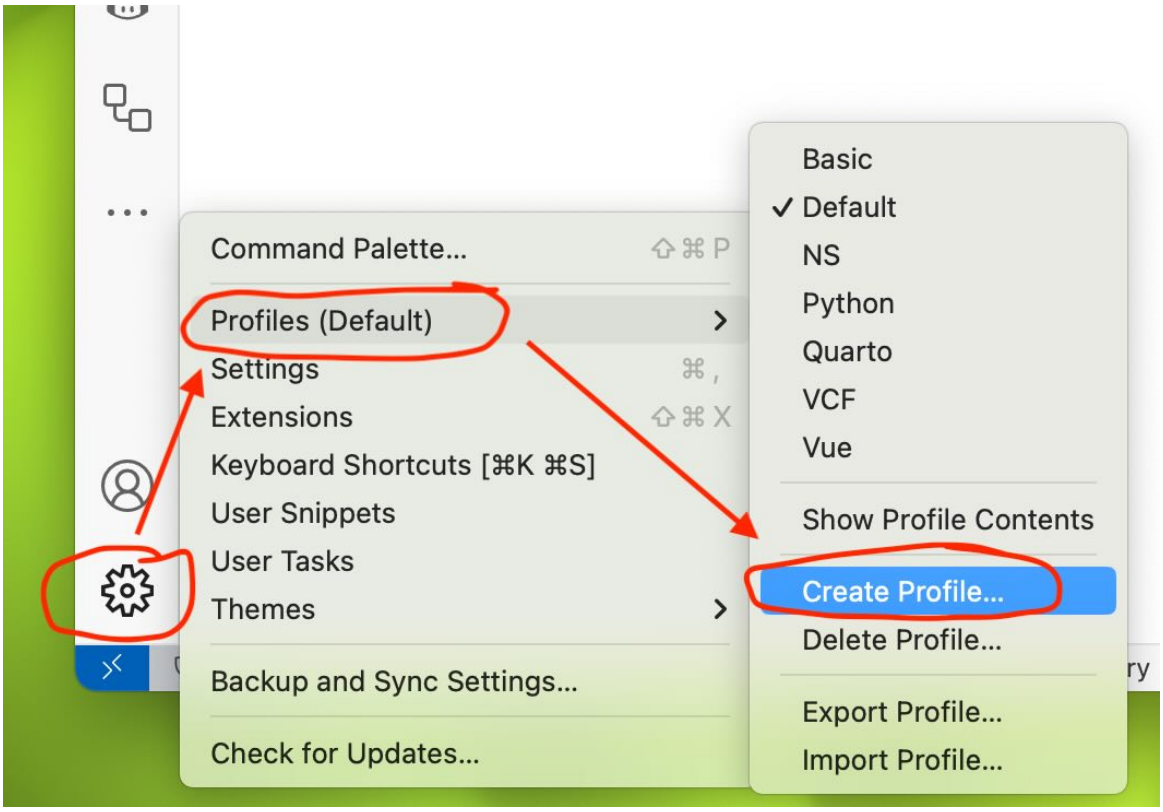
```
uv add pandas numpy scipy matplotlib seaborn statsmodels scikit-learn linearmodels pyarrow jupyter pytest
```

This step updates the `pyproject.toml` file and creates a `uv.lock` file, which contains the exact version of each dependency. This file is used to make sure that all collaborators use the same version of each library. Note that because our dependencies are built on top of other libraries, uv will also install the dependencies of our dependencies.

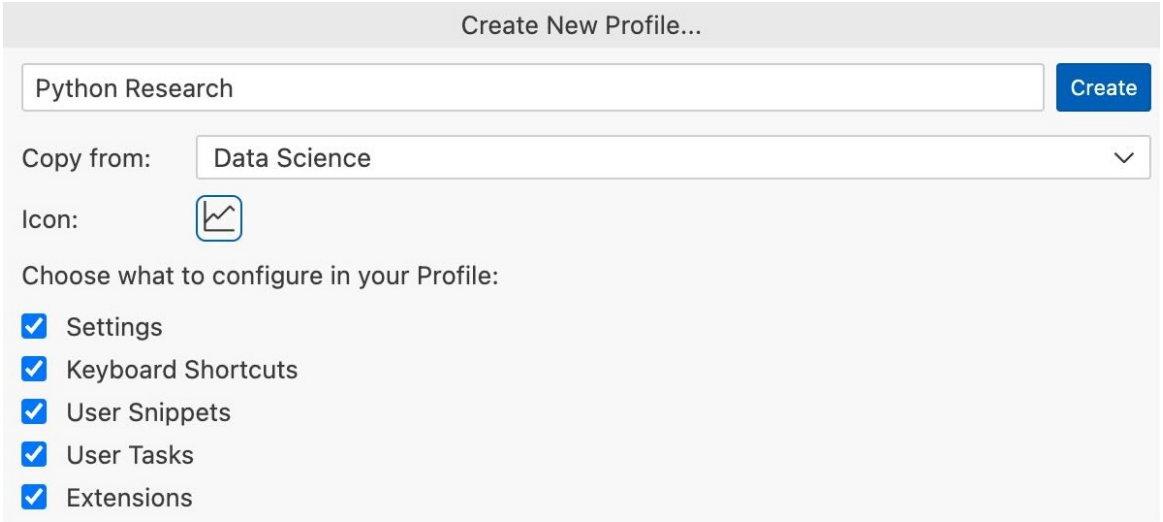
1.4. Configuring Visual Studio Code

Visual Studio Code is a free source code editor made by Microsoft. Features include support for debugging, syntax highlighting and intelligent code completion. While there are some built-in features for Python, most of the functionality comes from extensions. What I recommend is to use the profile feature of VS Code, which lets you define a set of extensions for each use case. For example, you can have a profile for Python development, another for R development, and another for LaTeX editing. This way, you can have a clean installation of VS Code and only install the extensions you need for each profile. Furthermore, each profile can have its specific settings and theming options.

To create a profile, click on the profile icon in the bottom left corner of the VS Code window. Then, under the `Profiles` section, click on `Create Profile`.



Give the profile a name and select a distinctive icon. Make sure to copy from the `Data Science` template, which will install all the extensions you need for data analysis with Python.



VS Code works best when you have a project (directory) open. To open a project, select `Open Folder` from the `File` menu and select the folder you want to open, for example, the `sandbox` folder we created earlier.

To open an interactive window, bring up the command palette by pressing `Cmd+Shift+P` (or `Ctrl+Shift+P` on Windows and Linux) and type `Python: Create Interactive Window`.

At this point, VS Code should have detected the virtual environment created by uv and should have asked you if you want to use it. If not, you can select it manually by clicking on the Python version in the top right corner of the interactive window.

1.5. GitHub.com (optional)

You do not need a GitHub account to have a complete Python environment. However, I recommend creating one because it will be useful later when we start working on projects.

To follow the some examples, you will need a GitHub account. You can create one for free at `https://github.com/`. GitHub offers many benefits to students and educators, including free access to GitHub Copilot and extra free hours for GitHub Codespaces. I highly recommend applying at GitHub Education if you are eligible.

While GitHub is the leader in the space, GitLab is their main competitor offering similar features. Gitea is a fully open-source solution for those who prefer to self-host.

1.6. Python in the cloud using Github Codespaces

Many online platforms allow you to develop and run Python code without installing anything on your computer. If you want to use a cloud-based solution, I recommend using GitHub Codespaces.

All you need is a GitHub account. However, note that GitHub Codespaces is **not free**. At the time of this writing, you get 60 hours per month for free, or 90 hours if you signed up for the GitHub Student Developer Pack (this is for a 2-core machine, which is the smallest machine available). After that, you have to pay for it (the current rate is USD 0.18 per hour).

Make sure to shut down your Codespace when you are not using it, otherwise you will run out of free hours very quickly.

1.6.1. Other cloud alternatives

There are many other cloud-based alternatives. However, most are based on Jupyter notebooks, which can be interesting when you are learning Python, but are not ideal for robust, replicable research. Some of the most popular alternatives are:

- Google Colab
- Cocalc
- WRDS Jupyter Hub (requires a WRDS subscription through your institution)

1.7. What's next?

Now that you have a complete Python environment, you can start learning Python. The next chapter introduces the basic Python syntax.

2. Python Syntax

2.1. Introduction

In this chapter, we lay the foundation for your programming skills by exploring the basic syntax of Python 3.14.

My aim is to make this process as accessible as possible for non-programmers while giving you the necessary tools to excel in the world of empirical finance research.

The objectives of this chapter are to:

1. Provide a gentle introduction to the basic syntax of Python, allowing you to read and understand Python code.
2. Enable you to write simple programs that will serve as building blocks for more advanced applications.
3. Equip you with the knowledge and confidence to further explore advanced topics in Python and its applications in finance.

By the end of this chapter, you will have a solid grasp of Python's basic syntax, empowering you to use it as a versatile tool for finance-related tasks. Remember, the key to success in learning any programming language is practice. As you work through this tutorial, be sure to experiment with the examples provided and try writing your own code to reinforce your understanding.

I am purposefully leaving out of this chapter more advanced topics such as object-oriented programming, testing, and logging. These topics are important, but they are not necessary to get started with Python, so will come later in the book.

2.2. Data types

The Python language offers many built-in fundamental data types. These data types serve as the building blocks for working with different kinds of data, which is critical in many applications. The basic data types you should be familiar with are presented in Table 2.1.

Table 2.1.: Main data types in Python

Name	Type	Description	Example
Integer	`int`	Integers represent whole positive and negative numbers. They are used for counting, indexing, and various arithmetic operations.	`1`

Name	Type	Description	Example
Float-Point Number	`float`	Floats represent real numbers with decimals. They are used for working with financial data that require precision, such as interest rates, stock prices, and percentages.	`1.0`
Complex	`complex`	Complex numbers consist of real and imaginary parts, represented as `a + bj`. While less commonly used in finance, they may be relevant in specific advanced applications, such as signal processing or quantitative finance.	`1.0 + 2.0j`
Boolean	`bool`	Booleans represent the truth values of True and False. They are used in conditional statements, comparisons, and other logical operations.	`True`
Text String	`str`	Strings are sequences of characters used for storing and manipulating text data, such as stock symbols, company names, or descriptions.	`"Hello"`
Bytes	`bytes`	Bytes are sequences of integers in the range of 0-255, often used for representing binary data or encoding text. Bytes may be used when working with binary file formats or network communication.	`b"Hello"`
None	`None`	`None` is a special data type representing the absence of a value or a null value. It is used to signify that a variable has not been assigned a value or that a function returns no value.	`None`

2.2.1. Literals

A literal is a notation for representing a fixed value in source code. For example, `42` is a literal for the integer value of forty-two. The following are examples of literals in Python. Each code block contains code and is followed by the output of the code.

`int`

`int` literals are written as positive and negative whole numbers.

```
42
```

```
42
```

```
-99
```

```
-99
```

They can also include underscores to make them more readable.

```
1_000_000
```

```
1000000
```

float

`float` literals are written as decimal numbers.

```
2.25
```

```
2.25
```

They can be written in scientific notation by using `e` to indicate the exponent.

```
2.25e8
```

```
225000000.0
```

To define a whole number literal as a `float` instead of an `int`, you can append a decimal point to the number.

```
2.0
```

```
2.0
```

complex

Complex numbers consist of a real part and an imaginary part, represented as `a + bj`.

```
2.3 + 4.5j
```

```
(2.3+4.5j)
```

None

`None` is a special data type that represents the absence of a value or a null value. It is used to signify that a variable has not been assigned a value or that a function returns no value.

```
None
```

bool

`bool` is a data type that represents the truth values of `True` and `False`. They are used in conditional statements, comparisons, and other logical operations.

```
True
```

```
True
```

2.2.2. str

Strings are sequences of characters. Strings literals are written by enclosing a sequence of characters in single or double quotes. Note that double quotes are preferred by the ruff formatter, which is used in this book, but most Python environments will use single quotes by default when displaying strings.

```
"USD"
```

```
'USD'
```

Strings are sequences of Unicode characters, which means they can represent any character in any language, including emojis, which unfortunately do not render properly in PDF for the example below.

```
"Bitcoin 🚀"
```

```
'Bitcoin 🚀'
```

String literals can span multiple lines by enclosing them in triple quotes or triple double quotes. This is useful for writing multiline strings.

```
# Multiline strings

"""GAFA is a group of companies:

- Google
- Apple
- Facebook
- Amazon

"""
```

```
'GAFA is a group of companies:\n\n- Google\n- Apple\n- Facebook\n- Amazon\n\n'
```

Multiline strings, or any strings with special characters, can be displayed using the `print` function.

```
print(
    """GAFA is a group of companies:

- Google
- Apple
- Facebook
- Amazon

"""
)
```

```
GAFA is a group of companies:

- Google
- Apple
- Facebook
- Amazon
```

2.2.3. `bytes`

`bytes` are sequences of integers in the range of 0-255. They are often used for representing binary data or encoding text. Bytes literals are written by prepending a string literal with `b`.

```
b"Hello"
```

```
b'Hello'
```

> **Bytes vs strings**
>
> Bytes can be confused with strings, but they are not the same. Strings are sequences of Unicode characters, while bytes are sequences of integers in the range of 0-255. Bytes are often used for representing binary data or encoding text. In most cases, you will be working with strings, but you may encounter bytes when working with binary file formats or network communication.

2.3. Variables

A variable in Python is a named location in the computer's memory that holds a value. It serves as a container for data, allowing you to reference and manipulate the data stored within it. Variables are created by assigning a value to a name using the assignment operator (`=`). They can store data of various types, such as integers, floats, strings, or even more complex data structures like lists.

Understanding the concept of variables and their naming conventions will help you write clean, readable, and maintainable code. An overview of variable naming rules in Python is presented in Table 2.2, and Table 2.3 presents some examples of valid and invalid variable names.

Table 2.2.: Variable naming rules

Rule	Description
Can contain letters, numbers, and underscores	Variable names can include any combination of letters (both uppercase and lowercase), numbers, and underscores (_). Python variable names support Unicode characters, enabling you to use non-English characters in your variable names. However, they must follow the other rules mentioned below.
Cannot start with a number	Although variable names can contain numbers, they must not begin with a number. For example, `1_stock` is an invalid variable name, whereas `stock_1` is valid.
Cannot be a reserved word	Python has a set of reserved words (e.g., `if`, `else`, `while`) that have special meanings within the language. You should not use these words as variable names.

Table 2.3.: Variable naming examples

Valid	Invalid
`ticker`	`1ceo`
`firm_size`	`@price`
`total_sum_2023`	`class`
`_tmp_buffer`	`for`

Case-sensitive

Python is case-sensitive, so `ret` and `RET` are two different variables.

Beyond the rules mentioned above, there are also some conventions that you should follow when naming variables. These conventions are not enforced by Python, but they are widely adopted by the Python community. Table 2.4 summarizes the most common conventions.

Table 2.4.: Variable naming conventions

Convention	Description
Use lowercase letters and underscores for variable names	To enhance code readability, use lowercase letters for variable names and separate words with underscores. For example, `market_cap` is a recommended variable name, whereas `MarketCap` or `marketCap` are not. This naming convention is known as snake case.
Use uppercase letters for constants	Constants are values that do not change throughout the program. Use uppercase letters and separate words with underscores to differentiate them from regular variables. For example, `INFLATION_TARGET` is a suitable constant name. Note that Python does not support constants like other languages, so this is just a convention, but Python won't stop you from changing the value of a constant.

By adhering to these guidelines, you will improve your coding style and ensure that your code is easier to understand, maintain, and collaborate on with your peers.

Reserved keywords

Reserved keywords cannot be used as variable names. You can check the complete list of reserved keywords by running the following command in the Python console:

```
help("keywords")
```

```
Here is a list of the Python keywords.  Enter any keyword to get more help.

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

Note that some reserved keywords may be confusing when thinking about finance problems. For example, `return`, `yield`, `raise`, `global`, `class`, and `lambda` are all reserved keywords, so you cannot use them as variable names. Most modern IDEs, such as Visual Studio Code, will highlight reserved keywords in a different color to help you avoid using them as variable names.

2.3.1. Declaring variables

A simple way to think about variables is to consider them labels that you can use to refer to values. For example, you can create a variable `x` and assign it a value of `42` using the assignment operator (`=`). You can then use the variable `x` to refer to the value `42` in your code.

```
x = 42
x
```

```
42
```

> **The walrus operator**
>
> In the previous example, we added `x` to the last line of the code to display the value of `x`. This is necessary in the interactive window and in Jupyer Notebooks, as they automatically display the result of the last line of the code. However, the assignment operator (`=`) does not return a value, so the value of `x` is not displayed without that last line.
>
> ```
> x = 42
> ```
>
> Introduced in Python 3.8, the `:=` operator, also known as the walrus operator, allows you to assign a value to a variable and return that value in a single expression. For example, you can use the walrus operator to assign a value of `10` to a variable `z` and use that variable in the same expression, assigning the result to `y`.
>
> ```
> y = (z := 10) * 2
> ```
>
> Note, however, that the walrus operator cannot be used to assign a value to a variable without using it in an expression. For example, the following code will raise an error.
>
> ```
> x := 42
> ```

You can reassign the value of a variable by assigning a new value to it. Once you reassign the value of a variable, the old value is lost. For example, you can reassign the value of `x` to `32` by running the following code.

```
x = 32
x
```

```
32
```

You can perform operations on variables, just like you would on values. For example, you can add `10` to `x`.

```
x + 10
```

```
42
```

You can assign the result of an operation to a new variable. For example, you can assign the result of `2 * 10` to a new variable `y`.

```
y = 2 * x
y
```

```
64
```

```
z = x + y
z
```

```
96
```

If you try to use a variable name that is invalid, Python will raise an error. For example, if you try to assign a variable `1ceo`, Python will raise an error because variable names cannot start with a number.

```
1ceo = 2
```

You can, however, use Unicode characters in variable names. For example, you can use accents such as `é` in a variable name.

```
cote_de_crédit = "AAA"
```

A leading underscore in a variable name indicates that the variable is private, which means that it should not be accessed outside of the module or scope in which it is defined. For example, you can use a leading underscore in a variable name to indicate that the variable is private. This is a convention that is widely adopted by the Python community, but it is not enforced by Python.

```
_hidden = 30_000
```

Another convention is to use all caps for constants. For example, you can use all caps to indicate that `INFLATION_TARGET` is a constant.

```
INFLATION_TARGET = 0.02
```

Python will raise an error if you attempt to use a variable that has not been declared. For instance, if you try to use the variable `inflation_target` instead of `INFLATION_TARGET`, Python will generate an error. It's important to note that Python is case-sensitive, so variables must be referenced with the exact casing as their declaration.

2.3.2. Variable types

Python is a dynamically typed language, meaning you do not need to specify the variable type when you declare it. Instead, Python will automatically infer the type of a variable based on the value you assign to it. For example, if you assign an integer value to a variable, Python will infer that the variable is an integer. Similarly, if you assign a string value to a variable, Python will infer that the variable is a string. You can use the `type()` function to check the type of a variable. For example, you can check the type of `a` by running the following code.

```
a = 3.3
type(a)
```

```
float
```

```
b = 2
type(b)
```

```
int
```

```
market_open = True
type(market_open)
```

```
bool
```

```
currency = "CAD"
type(currency)
```

```
str
```

Variables explorer in Visual Studio Code

VS Code has a built-in variable explorer that allows you to view the variables in your workspace when using the interactive window or a Jupyer Notebook. You can open the *Variables View* by clicking on the *Variables* button in the top toolbar of the editor:

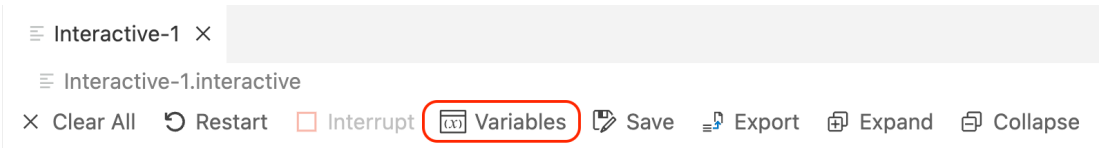


Figure 2.1.: Variable View button

The *Variables View* will appear at the bottom of the window, showing the variables in your workspace, along with their values, types, and size for collections. For example, the following screenshot shows the variables in the workspace after running the code in this section:

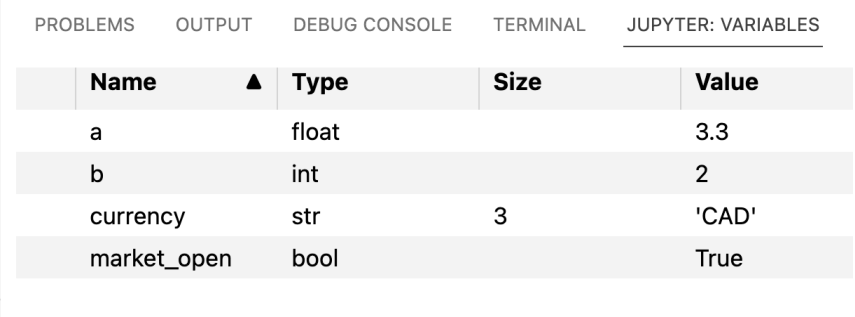


Figure 2.2.: Variable View

Converting between types

You can convert a variable from one type to another using the built-in functions `float()`, `int()`, `str()`, and `bool()`. For example, you can convert the variable `x`, which is currently an `int`, to a `float` by running the following code.

```
float(x)
```

```
32.0
```

The same way, you can convert the variable `y`, which is currently a `float`, to an `int` by running the following code. Note that the `int()` function will round down the value of `y` to the nearest integer.

```
int(a)
```

```
3
```

Similarly, you can convert the variable `x` to a string by running the following code.

```
str(x)
```

```
'32'
```

You can convert a string to an integer or a float if the string contains a valid representation of a number. For example, you can convert the string `"42"` to an integer by running the following code.

```
int('42')
```

```
42
```

However, you cannot convert a string that does not contain a valid representation of a number to an integer. For example, you cannot convert the string `"42.5"` to an integer.

When converting to a boolean value, most values will be converted to `True`, except for `0`, `0.0`, and `""`, which will be converted to `False`.

```
bool(0)
```

```
False
```

```
bool(1)
```

```
True
```

```
bool("")
```

```
False
```

```
bool("33")
```

```
True
```

The `None` value is a special type in Python that represents the absence of a value. You can use the `None` value to initialize a variable without assigning it a value. For example, you can initialize a variable `problem` to `None` by running the following code.

```
problem = None
type(problem)
```

```
NoneType
```

2.4. Comments

Comments are an essential part of writing clear, maintainable code. They help explain the purpose, logic, or any specific details of the code that might not be obvious at first glance. However, excessive or unnecessary commenting can clutter your code and make it harder to read. To strike the right balance, consider the guidelines listed in Table 2.5 when deciding when to use comments and when to avoid them:

Table 2.5.: Guidelines for comments

Guideline	Description
Use comments when the code is complex or non-obvious	When your code involves complex algorithms, calculations, or logic that may be difficult for others (or yourself) to understand at a glance, use comments to explain the reasoning behind the code or to provide additional context.
Avoid comments for simple or self-explanatory code	For code that is simple, clear, and easy to understand, avoid adding comments. Instead, use descriptive variable and function names that convey the purpose of the code.
Use comments to explain the 'why', not the 'how'	Good comments explain the purpose of a piece of code or the reasoning behind a decision. Focus on providing context and insight that isn't immediately apparent from reading the code. Avoid repeating what the code is doing, as this can be redundant and clutter the code.
Avoid commenting out large blocks of code	Instead of leaving large blocks of commented-out code in your final version, remove them. It's better to use version control systems like Git to keep track of previous versions of your code.
Keep comments up-to-date	Ensure that your comments are always up-to-date with the code they describe. Outdated comments can be confusing and misleading, making it harder to understand the code.
Use comments to provide additional information	Use comments to provide references to external resources, such as links to relevant documentation, papers, or articles. This can be helpful for providing additional context or background information related to the code.
Use consistent commenting style	Follow a consistent commenting style throughout your codebase. This makes it easier for others to read and understand your comments.

2.4.1. Writing comments

In Python, comments are created using the `#` symbol. Any text that follows the `#` symbol on the same line is ignored by the Python interpreter. Comments can be placed on a separate line or at the end of a line of code.:

```
# This is a single-line comment

price = 150  # This is an inline comment
```

You can also create multi-line comments by enclosing the text in triple quotes (`"""` or `'''`). Multi-line comments are often used to provide docstrings (documentation strings) for functions and classes. We'll learn more about

functions and classes in a later section. Note that multi-line comments are technically strings, but the Python interpreter ignores them and does not store them in memory because they are not assigned to a variable.

```
"""
This is a multi-line comment.
You can write your comments across multiple lines.
"""
```

```
'\nThis is a multi-line comment.\nYou can write your comments across multiple lines.\n'
```

Comments can occur alongside code to document its purpose or explain the logic.

```
# Calculate compound interest
principal = 1000  # Principal amount
rate = 0.05  # Annual interest rate
time = 5  # Time in years

# Future value with compound interest formula
future_value = principal * (1 + rate) ** time

# Display the result
print(f"Future value: {future_value:.2f}")
```

```
Future value: 1276.28
```

> **Don't overdo it**
>
> Comments are useful for providing additional context or explanation, but they can also be overdone. Avoid adding comments for trivial or self-explanatory code. For example, the code above is simple and clear enough to understand without comments, so adding comments is decreasing readability instead of improving it.

Comments are usually written in English, but you can use any language as long as the file is UTF-8 encoded. You can also use emojis in comments if you like 🙂.

2.5. Numbers

Python provides built-in functions and operators to perform mathematical operations on numbers. Some commonly used mathematical functions include `abs()`, `round()`, `min()`, `max()`, and `pow()`. Additionally, Python's `math` library offers more advanced functions like trigonometry and logarithms.

Rounding errors

Floating-point numbers may be subject to rounding errors due to the limitations of their binary representation. Keep this in mind when comparing or performing calculations with floats. Consider using the `Decimal` data type from Python's `decimal` library to avoid floating-point inaccuracies when dealing with high-precision financial data.

Performance

When working with large datasets or performing complex calculations, consider using third-party libraries like NumPy and pandas, which are covered in later chapters, for improved performance and additional functionality.

2.5.1. Operations

The Python language supports many mathematical operations. Table 2.6 lists some of the most commonly used operators.

Table 2.6.: Basic Arithmetic Operations

Operator	Name	Example	Result
`+`	Addition	`1 + 2`	`3`
`-`	Subtraction	`1 - 2`	`-1`
`*`	Multiplication	`3 * 4`	`12`
`/`	Division	`1 / 2`	`0.5`
`**`	Exponentiation	`2 ** 3`	`8`
`//`	Floor division	`14 // 3`	`4`
`%`	Modulo (remainder)	`14 % 3`	`2`

```
a = 5
b = 3

print(f"Addition: a + b = {a + b}")
print(f"Subtraction: a - b = {a - b}")
print(f"Multiplication: a * b = {a * b}")
print(f"Division: a / b = {a / b}")
print(f"Exponentiation: a ** b = {a ** b}")
print(f"Floor Division: a // b = {a // b}")
print(f"Modulo: a % b = {a % b}")
```

```
Addition: a + b = 8
Subtraction: a - b = 2
```

```
Multiplication: a * b = 15
Division: a / b = 1.6666666666666667
Exponentiation: a ** b = 125
Floor Division: a // b = 1
Modulo: a % b = 2
```

> **f-strings**
>
> The previous examples use a special type of strings called f-strings to format the output. f-strings are a convenient way to embed variables and expressions inside strings. They are denoted by the `f` prefix and curly braces (`{}`) containing the variable or expression to be evaluated.
> We cover f-strings in more details in Section 2.7.2.

2.5.2. Common mathematical functions

To round numbers, use the `round()` function. The `round()` function takes two arguments: the number to be rounded and the number of decimal places to round to. The number is rounded to the nearest integer if the second argument is omitted.

```
rounded_num = round(5.67, 1)

print(rounded_num)
print(type(rounded_num))


rounded_to_int = round(5.67)

print(rounded_to_int)
print(type(rounded_to_int))
```

```
5.7
<class 'float'>
6
<class 'int'>
```

Some mathematical functions will require the use of the math module from the Python Standard Library. The standard library is a collection of modules included with every Python installation. You can use the functions and types in these modules by importing them into your code using the `import` statement.

For example, to calculate the square root of a number, you can use the `sqrt()` function from the `math` module:

```
import math                                                    ①

math.sqrt(25)
```

① Imports the math module, making its functions available in the current code.

```
5.0
```

This is only one of the many functions in the `math` module. You can view the complete list of functions in the module documentation. The `math` module also contains constants like `pi` and `e`, which you can access using the dot notation.

```
math.pi
```

```
3.141592653589793
```

2.5.3. Random numbers

It is often useful to generate random numbers for simulations and other applications. Python's `random` module provides functions for generating pseudo-random[1] numbers from different distributions.

> **!** Pseudo-random number generator
>
> The `random` module uses the Mersenne Twister algorithm to generate pseudo-random numbers. This algorithm is deterministic, meaning that given the same seed value, it will produce the same sequence of numbers every time. This is useful for debugging and testing but not for security purposes. If you need a cryptographically secure random number generator, use the `secrets` module instead.

The `random.seed()` function initializes the pseudo-random number generator. If you do not call this function, Python will automatically call it the first time you generate a random number. The `random.seed()` function takes an optional argument that can be used to set the seed value. This can be useful for debugging and testing, allowing you to generate the same sequence of random numbers every time. If you do not specify a seed, Python will use the system time as the seed value, so you will get a different sequence of random numbers every time.

```
import random

random.seed(42)  ①
```

① Sets the seed value to 42. Why 42? Because it's the answer to life, the universe, and everything.

`random.random()` generates a random float between 0 and 1 (exclusive).

```
rand_num = random.random()

rand_num
```

[1] A pseudo-random number is a sequence of numbers that appear random but are generated using a deterministic algorithm.

```
0.6394267984578837
```

`random.randint(a, b)` generates a random integer between `a` and `b` (inclusive).

```
rand_int = random.randint(1, 10)

rand_int
```

```
1
```

`random.uniform(a, b)` generates a random float between `a` and `b` (exclusive).

```
rand_float = random.uniform(0, 1)

rand_float
```

```
0.7415504997598329
```

`random.normalvariate(mu, sigma)` generates a random float from a normal distribution with mean `mu` and standard deviation `sigma`.

```
rand_norm = random.normalvariate(0, 1)

rand_norm
```

```
-0.508616386057752
```

The full list of functions in the `random` module can be found in the module documentation.

2.5.4. Floats and decimals

Because of the way computers store numbers, floating-point numbers are not exact. This can lead to unexpected results when performing arithmetic operations on floats.

```
2.33 + 4.44
```

```
6.7700000000000005
```

To avoid this problem when exact results are needed, use the `Decimal` type from the `decimal` module to perform arithmetic operations on decimal numbers. You could import the module using `import decimal` but this would require you to prefix all the functions and types in the module with `decimal`. To avoid this, you can directly import the `Decimal` type from the `decimal` module using `from decimal import Decimal`.

```
from decimal import Decimal                                    ①

Decimal("2.33") + Decimal("4.44")
```

① Imports the `Decimal` type from the `decimal` module. You can now refer to the `Decimal` type directly without having to prefix it with `decimal`.

```
Decimal('6.77')
```

Decimals vs. floats

Using the Decimal type in Python provides precise decimal arithmetic and avoids rounding errors, making it suitable for financial and monetary calculations, while floats offer faster computation and are more memory-efficient but can introduce small inaccuracies due to limited precision and binary representation.

2.5.5. Financial formulas

Many financial calculations involve performing arithmetic operations on financial data. Here are two examples of common calculations in finance and how they can be implemented in Python.

Calculating the present value of a future cash flow

The formula for calculating the present value of a future cash flow is:

$$PV = \frac{FV_t}{(1+r)^t},$$

where FV_t is the future value of the cash flow at time t, r is the discount rate, and t is the number of periods.

```
future_value = 1000
discount_rate = 0.05
periods = 5

present_value = future_value / (1 + discount_rate) ** periods

present_value
```

```
783.5261664684588
```

Calculating the future value of an annuity

The formula for calculating the future value of an annuity is:

$$FV = PMT\frac{(1+r)^t - 1}{r},$$

where PMT is the payment, r is the interest rate, and t is the number of periods.

It can be written in Python as:

```
payment = 100
rate = 0.05
periods = 5

future_value_annuity = payment * ((1 + rate) ** periods - 1) / rate

future_value_annuity
```

```
552.5631250000007
```

> Parentheses and operator precedence
>
> Python, just like mathematics, follows a specific order of operations when evaluating expressions. The complete list of precedence rules can be found in the Python documentation.
> **When in doubt, use parentheses to make the order of operations explicit.**
> For arithmetic operations, the order of operations is as follows:
>
> 1. Exponents
> 2. Negative (-)
> 3. Multiplication and division
> 4. Addition and subtraction

2.6. Defining functions

Functions are blocks of organized and reusable code that perform a specific action. They allow you to encapsulate a set of instructions, making your code modular and easier to maintain. Functions can take input parameters, perform operations on those inputs, and return a result.

Defining a function in Python involves the following steps:

1. **Use the `def` keyword:** Start by using the `def` keyword, followed by the function name and parentheses that enclose any input parameters.

2. **Add input parameters:** Specify any input parameters within the parentheses, separated by commas. These parameters allow you to pass values to the function, which it can then use in its calculations or operations.
3. **Write the function body:** After the parentheses, add a colon (`:`) and indent the following lines to create the function body. This block of code contains the instructions that the function will execute when called.
4. **Return a result (optional):** If your function produces a result, use the `return` statement to send the result back to the caller. If no `return` statement is specified, the function will return `None` by default.

> **Best practices**
>
> When defining functions, keep the following best practices in mind:
>
> - **Choose descriptive function names:** Use meaningful names that reflect the purpose of the function, making your code more readable and easier to understand.
> - **Keep functions small and focused:** Each function should have a single responsibility, making it easier to test, debug, and maintain.

We can define functions to perform a wide variety of tasks. For example, we can define a function to calculate the present value of a future cash flow:

```
def present_value(future_value, discount_rate, periods):                    ①
    return future_value / (1 + discount_rate) ** periods                    ②


# Example usage:
future_value = 1000
discount_rate = 0.05
periods = 5
result = present_value(future_value, discount_rate, periods)                ③
print(f"Present value: {result:.2f}")
```

① Defines a function called `present_value` that takes three input parameters: `future_value`, `discount_rate`, and `periods`.

② Calculates the present value of a future cash flow using the formula from the previous section and returns the result to the caller. The code in the function body is indented to indicate that it is part of the function.

③ Calls the `present_value` function with the specified input values and stores the returned value in a variable called `result`. When the function is called, the input values are passed to the function as arguments in the same order as the parameters were defined. The function body is then executed, and the result is returned to the caller.

```
Present value: 783.53
```

Indentation

Indentation refers to the spaces or tabs used at the beginning of a line to organize code. It helps Python understand the program's structure and which lines of code are grouped together.
The Python language specification mandates the use of consistent indentation for code readability and proper execution. Indentation is typically achieved using four spaces per level. It plays a crucial role in determining the scope and hierarchy of statements within control structures, such as loops and conditional statements. For example, the statements that are part of a function body must be indented to indicate that they are part of the function. The Python interpreter knows that the function body ends when the indentation level returns to the previous level.

We will learn more about functions in Section 2.11.

2.7. Strings

Text data is often encountered in finance in the form of stock symbols, company names, descriptions, or financial reports. Understanding how to work with strings is essential for processing and manipulating text data effectively.

Strings are sequences of characters, and they can be created using single quotes (`' '`), double quotes (`" "`), or triple quotes (`''' '''` or `""" """`) for multi-line strings.

Special characters

Some characters have special meanings in Python strings. The backslash (`\`) is used to escape characters that have special meaning, such as newline (`\n`) or tab (`\t`). To include a backslash in a string, you need to escape it by adding another backslash before it (`\\`). Alternatively, you can use raw strings by prefixing the strings with `r` or `R`, which will treat backslashes as literal characters. For example, these two strings are equivalent:

```
str1 = "C:\\Users\\John"
str2 = r"C:\Users\John"
```

2.7.1. String operations

The Python language provides many common string operations. Table 2.7 lists some of the most commonly used operations.

Table 2.7.: Common string operations

Operation	Example	Description
Concatenate strings	`result = str1 + " " + str2`	Combines two or more strings together

Operation	Example	Description
Repeat strings	`result = repeat_str * 3`	Repeats a string a specified number of times
Length of a string	`length = len(text)`	Gets the length (number of characters) of a string
Access characters in a string	`first_char = text[0]`	Retrieves a specific character in a string
Slice a string	`slice_text = text[0:12]`	Extracts a part of a string
Convert case	`upper_text = text.upper()`	Converts a string to uppercase
	`lower_text = text.lower()`	Converts a string to lowercase
Join a list of strings	`text = ", ".join(companies)`	Joins a list of strings using a delimiter
Split a string	`companies = text.split(", ")`	Splits a string into a list based on a delimiter
Replace a substring	`new_text = text.replace("Finance", "Python")`	Replaces a specified substring in a string
Check substring existence	`result = substring in text`	Checks if a substring exists in a string

We can concatenate (combine) two or more strings into a single string using the `+` operator.

```
str1 = "Hello"
str2 = "World"
result = str1 + " " + str2
print(result)
```

```
Hello World
```

The `*` operator repeats a string multiple times.

```
repeat_str = "Python "
result = repeat_str * 3
print(result)
```

```
Python Python Python
```

The `len()` function returns the string's length (number of characters).

```
text = "Finance"
length = len(text)
print(length)
```

```
7
```

Single characters in a string can be accessed using the index of the character within square brackets (`[]`). Python uses zero-based indexing, so the first character in a string has index 0, the second character has index 1, and so on. You can also use negative indices to access characters from the end of a string, with the last character having index -1, the second last character having index -2, and so on.

```
text = "Python"
first_char = text[0]
last_char = text[-1]
print(first_char)
print(last_char)
```

```
P
n
```

Extracting a portion of a string by specifying a start and end index is called *slicing*. In Python, you can slice a string using the following syntax: `text[start:end]`. The start index is inclusive, while the end index is exclusive. If the start index is omitted, it defaults to 0. If the end index is omitted, it defaults to the length of the string.

```
text = "empirical finance Python"
slice_text = text[0:7]
print(slice_text)
```

```
empiric
```

The `upper()` and `lower()` *methods* convert a string to uppercase or lowercase, respectively.

```
text = "Finance"
upper_text = text.upper()
lower_text = text.lower()
print(upper_text)
print(lower_text)
```

```
FINANCE
finance
```

> **Methods vs functions**
>
> A method is similar to a function but associated with a specific object or data type. In this case, `upper()` and `lower()` are methods specific to the `str` (string) data type. When we call the upper method on the `text` object using the dot notation (`text.upper()`), Python knows to transform the string stored in the `text` variable. Methods are particularly useful because they allow us to perform actions or operations specific to the object or data type they belong to, and they improve code readability by making it clear what

> object the method is being called on.

The `join()` method joins a list of strings into a single string using a delimiter. The delimiter can be specified as an argument to the `join()` method. Lists are introduced in the next section.

```
companies = ["Apple", "Microsoft", "Google"]
text = " | ".join(companies)
print(text)
```

```
Apple | Microsoft | Google
```

The `split()` method splits a string into a list of substrings based on a delimiter. The delimiter can be specified as an argument to the `split()` method. If no delimiter is specified, the string is split on whitespace characters.

```
text = "Apple, Microsoft, Google"
companies = text.split(", ")
print(companies)
```

```
['Apple', 'Microsoft', 'Google']
```

The `replace()` method replaces a substring in a string with another string. It takes two arguments: the substring to replace and the string to replace it with.

```
text = "Introduction to Finance"
new_text = text.replace("Finance", "Python")
print(new_text)
```

```
Introduction to Python
```

The `in` operator checks if a substring exists in a string. It returns a boolean value, `True` if the substring exists in the string, and `False` otherwise.

```
text = "Introduction to Python"
substring = "Python"
result = substring in text
print(result)
```

```
True
```

The Python documentation provides a complete list of string methods that you can refer to for more details.

2.7.2. Formatting strings

You will often encounter situations where you must present or display data in a formatted, human-readable manner. F-strings are a powerful tool for formatting strings and embedding expressions or variables directly within the string. They provide a concise and easy-to-read way of formatting strings, making them an essential tool for working with text data.

F-strings, also known as "formatted string literals," allow you to embed expressions, variables, or even basic arithmetic directly into a string by enclosing them in curly braces `{}` within the string. The expressions inside the curly braces are evaluated at runtime and then formatted according to the specified format options.

Some key features of f-strings that are useful include:

1. Expression Evaluation: You can embed any valid Python expression within the curly braces, including variables, arithmetic operations, or function calls. This feature enables you to generate formatted strings based on your data dynamically.
2. Formatting Options: F-strings support various formatting options, such as alignment, width, precision, and thousand separators. These options can be specified within the curly braces after the expression, separated by a colon (`:`).
3. Format Specifiers: You can use format specifiers to control the display of numbers, such as specifying the number of decimal places, using scientific notation, or adding a percentage sign. Format specifiers are especially useful in finance when working with currency, percentages, or large numbers.

To create an f-string, prefix the string with an `f` character, followed by single or double quotes. You can then embed expressions or variables within the string by enclosing them in curly braces (`{}`). For example, this lets you concatenate strings and variables together in a single statement:

```
ticker = "AAPL"
exchange = "NASDAQ"
company_name = "Apple, Inc."
full_name = f"{company_name} ({exchange}:{ticker})"
print(full_name)
```

```
Apple, Inc. (NASDAQ:AAPL)
```

Python will convert the expression within the curly braces to a string, which can be used to convert numbers to strings.

```
num = 42
num_str = f"{num}"
print(num_str)
```

```
42
```

Python evaluates the expression within the curly braces at runtime and then formats the string according to the specified format options. For example, you can use the `:,.2f` format option to display a number with a thousand separator and two decimal places.

```
amount = 12345.6789
formatted_amount = f"${amount:,.2f}"
print(formatted_amount)
```

```
$12,345.68
```

You can also use the `:.2%` format option to display a number as a percentage with two decimal places.

```
rate = 0.05
formatted_rate = f"{rate:.2%}"
print(formatted_rate)
```

```
5.00%
```

The `datetime` module provides a `datetime` class to represent dates and times. The `datetime` class has a `now()` method that returns the current date and time. You can use the `:%Y-%m-%d` format option to display the date in `YYYY-MM-DD` format.

```
from datetime import datetime

current_date = datetime.now()
formatted_date = f"{current_date:%Y-%m-%d}"
print(formatted_date)
```

```
2025-12-31
```

You can also use f-strings to align text to the left (`<`), right (`>`), or center (`^`) within a fixed-width column:

```
ticker = "AAPL"
price = 150.25
change = -1.25

formatted_string = f"|{ticker:<10}|{price:^10.2f}|{change:>10.2f}|"  ①
print(formatted_string)
```

① The `:<10` format option aligns the text to the left within a 10-character column. The `:^10.2f` format option aligns the number to the center within a 10-character column and displays it with two decimal places. The `:>10.2f` format option aligns the number to the right within a 10-character column and displays it with two decimal places.

```
|AAPL      |  150.25  |     -1.25|
```

Multiline f-strings work the same way as multiline strings, except that they are prefixed with an `f` character. You can use multiline f-strings to create formatted strings that span multiple lines.

```
stock = "AAPL"
price = 150.25
change = -1.25

formatted_string = f"""
Stock:  \t{stock}
Price:  \t${price:.2f}
Change: \t${change:.2f}
"""
print(formatted_string)
```

```
Stock:      AAPL
Price:      $150.25
Change:     $-1.25
```

Alternative formatting methods

When reading code or answers on websites such as Stack Overflow or receiving suggestions from AI-assisted coding assistant, you may encounter other string formatting methods. Before f-strings, the two primary string formatting methods in Python were `%`-formatting and `str.format()`.

%-formatting

Also known as printf-style formatting, `%`-formatting uses the `%` operator to replace placeholders with values. Inspired by the `printf` function in C, it has been available since early versions of Python. It is less readable and more error-prone than other methods.

Example:

```
formatted_string = "%s has a balance of $%.2f" % (name, balance)
```

str.format()

The `str.format()` method embeds placeholders using curly braces `{}` and replaces them with the format() method. Introduced in Python 2.6, it offers improved readability and more advanced formatting options compared to `%`-formatting.

Example:

```
formatted_string = "{} has a balance of ${:,.2f}".format(name, balance)
```

Advantages of f-strings

I recommend using f-strings instead of `%`-formatting or `str.format()` for string formatting for the following reasons:

1. **Readability:** Concise syntax with expressions and variables embedded directly.
2. **Flexibility:** Supports any valid Python expression within curly braces.
3. **Performance:** Faster than other methods, evaluated at runtime.
4. **Simplicity:** No need to specify variable order or maintain separate lists.

2.8. Collections

Sequences and collections are fundamental data structures in Python that allow you to store and manipulate multiple elements in an organized manner. They differ along three dimensions: order, mutability, and indexability. An ordered collection is one where the elements are stored in a particular order, the order of the elements is important, and you can iterate over the elements in that order. A collection is mutable if you can add, remove, or modify elements, after it is created. A collection is indexable if you can refer to its elements by their index (position or key).

Table 2.8 presents the main types of sequences and collections in Python. You are already familiar with the string type, an ordered, immutable, and indexable sequence of characters.

Table 2.8.: Sequences and collections in Python

Name	Type	Description	Example
List	`list`	Ordered, mutable, and indexed. Allows duplicate members.	`[1, 2, 3]`
Tuple	`tuple`	Ordered, immutable, and indexed. Allows duplicate members.	`(1, 2, 3)`
Set	`set`	Unordered, mutable, and unindexed. No duplicate members.	`{1, 2, 3}`
Dictionary	`dict`	Unordered, mutable, and indexed. No duplicate index entries. Elements are indexed according to a key.	`{"a": 1, "b": 4}`
String	`string`	Ordered, immutable, and indexed. Allows duplicate characters.	`"abc"`

2.8.1. Lists

Lists in Python are ordered collections of items that can hold different data types. They are mutable, meaning that elements can be added, removed, or modified. Lists are versatile and commonly used to store and manipulate sets of related data. The elements within a list are accessed using indexes, which allow for easy retrieval and modification. Lists also support various built-in methods and operations for efficient data manipulation, such as appending, extending, sorting, and slicing.

A list is created by enclosing a comma-separated sequence of elements within square brackets (`[ ]`). The elements can be of any data type, including other lists. The following code snippet creates a list of strings and a list of integers.

```
stocks = ["AAPL", "GOOG", "MSFT"]
prices = [150.25, 1200.50, 250.00]
```

You can access the elements of a list using their index. The index of the first element is 0, the index of the second element is 1, and so on. You can also use negative indexes to access elements from the end of the list. The index of the last element is -1, the index of the second to last element is -2, and so on. The following code snippet illustrates how to access the elements of the `stocks` and `prices` lists.

```
first_stock = stocks[0]
print(first_stock)

last_price = prices[-1]
print(last_price)
```

```
AAPL
250.0
```

You can replace the elements of a list by assigning new values to their indexes, add new elements to the list using the `append()` method, or delete elements from the list using the `remove()` method.

```
# Replace an element
stocks[1] = "GOOGL"
print(stocks)

# Adding an element to the list
stocks.append("AMZN")
print(stocks)

# Removing an element from the list
stocks.remove("MSFT")
print(stocks)
```

```
['AAPL', 'GOOGL', 'MSFT']
['AAPL', 'GOOGL', 'MSFT', 'AMZN']
['AAPL', 'GOOGL', 'AMZN']
```

You can also use the `len()` function to get the length of a list, and the `in` operator to check if an element is present in a list.

```
# Length of the list
list_length = len(stocks)
print(f"Length: {list_length}")

# Checking if an element is in the list
is_present = "AAPL" in stocks
print(f"Is AAPL in the list? {is_present}")
```

```
Length: 3
Is AAPL in the list? True
```

2.8.2. Tuples

Tuples are ordered collections of elements that are immutable, meaning they cannot be modified after creation. They are typically used to store related pieces of data as a single entity, and their immutability provides benefits such as ensuring data integrity and enabling safe data sharing across different parts of a program.

> **Tuples vs lists**
>
> Tuples and lists in Python differ in mutability, syntax, and use cases. Tuples are commonly used for fixed data, have a slight performance advantage over lists and can be more memory-efficient. Lists are commonly used for variable data, and provide more flexibility in terms of operations and methods.

A tuple is created by enclosing a comma-separated sequence of elements within parentheses (`( )`). The elements can be of any data type, including other tuples. The following code snippet creates a tuple of integers and a tuple of strings.

```
mu = 0.1
sigma = 0.2
theta = 0.5

parameters = (mu, sigma, theta)
print(parameters)
```

```
(0.1, 0.2, 0.5)
```

You can access the elements of a tuple using their index, find their length using `len()`, just like with lists.

```
# Accessing elements in a tuple
sigma0 = parameters[1]
print(sigma0)

# Length of the tuple
tuple_length = len(parameters)
print(f"Length: {tuple_length}")
```

```
0.2
Length: 3
```

Tuples are immutable, so you cannot add, remove, or replace their elements directly. You can, however, create a new tuple with the modified elements. Also note that you can modify mutable elements within a tuple, such as a list.

```
a = [1, 2, 3]
b = ("c", a, 2)
print(f"Before appending to list a: {b}")

a.append(4)
print(f"After appending to list a: {b}")
```

```
Before appending to list a: ('c', [1, 2, 3], 2)
After appending to list a: ('c', [1, 2, 3, 4], 2)
```

`b` still contains the same elements, but the list `a` within the tuple has been modified.

Tuple unpacking

Tuple unpacking is a powerful feature of Python that allows you to assign multiple variables from the elements of a tuple in a single line of code. It is a form of "destructuring assignment" that provides a concise way to extract the elements of a tuple into individual variables.

To perform tuple unpacking, you use a sequence of variables on the left side of an assignment statement, followed by a tuple on the right side. When the assignment is made, each variable on the left side will be assigned the corresponding value from the tuple on the right side.

Here is an example:

```
# Create a tuple
t = (1, 2, 3)

# Unpack the tuple into three variables
a, b, c = t

# Display the values of the variables
print(f"a: {a}, b: {b}, c: {c}")
```

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

In this example, the tuple `t` contains three elements: 1, 2, and 3. When the tuple is unpacked into the variables `a`, `b`, and `c`, each variable gets assigned the corresponding value from the tuple: `a` gets 1, `b` gets 2, and `c` gets 3.

Tuple unpacking can be useful in various situations. For example, when working with functions that return multiple values as a tuple, you can use tuple unpacking to assign the return values to individual variables. Here's an example:

```
# Define a function that returns a tuple
def get_top3_stocks():
    return ("AAPL", "MSFT", "AMZN")

# Unpack the returned tuple into three variables
stock1, stock2, stock3 = get_top3_stocks()

# Display the values of the variables
print(f"Largest: {stock1}, 1nd: {stock2}, 3rd: {stock3}")
```

```
Largest: AAPL, 1nd: MSFT, 3rd: AMZN
```

Note that the number of variables on the left side of the assignment must match the number of elements in the tuple being unpacked.

2.8.3. Sets

Sets are unordered collections of unique elements. They are defined using curly braces `{ }` or the `set()` constructor. Sets do not allow duplicate values and support various operations such as intersection, union, and difference. Sets are commonly used for tasks like removing duplicates from a list, membership testing, and mathematical operations on distinct elements.

```
# Creating a set
unique_numbers = {1, 2, 3, 2, 1}
print(unique_numbers)

# Adding an element to the set
unique_numbers.add(4)
print(f"Added 4: {unique_numbers}")

# Removing an element from the set
unique_numbers.remove(1)
print(f"Removed 1: {unique_numbers}")

# Checking if an element is in the set
is_present = 2 in unique_numbers
print(f"Is 2 in the set? {is_present}")

# Length of the set
set_length = len(unique_numbers)
print(f"Length: {set_length}")
```

```
{1, 2, 3}
Added 4: {1, 2, 3, 4}
Removed 1: {2, 3, 4}
Is 2 in the set? True
Length: 3
```

Sets support operations such as intersection, union, and difference, which are performed using the `&`, `|`, and `-` operators respectively.

```
set1 = {1, 2, 3, 4}
set2 = set([3, 4, 5, 6])

# Intersection
print(f"Intersection: {set1 & set2}")

# Union
print(f"Union: {set1 | set2}")

# Difference
print(f"Difference: {set1 - set2}")
```

```
Intersection: {3, 4}
Union: {1, 2, 3, 4, 5, 6}
Difference: {1, 2}
```

> **Sets and data types**
>
> Sets can contain elements of different data types, including numbers, strings, and tuples. However, sets only support immutable elements, so you cannot add lists or dictionaries to a set.

2.8.4. Dictionaries

Dictionaries are key-value pairs that provide a way to store and retrieve data using unique keys. They are defined with curly braces `{ }` like sets, but contain pairs of elements called items, where each item is a key-value pair separated by a colon (`:`).

Dictionaries are unordered and mutable, allowing for efficient data lookup and modification. They are commonly used for mapping and associating values with specific keys, making them useful for tasks like storing settings, organizing data, or building lookup tables.

```
stock_prices = {"AAPL": 150.25, "GOOGL": 1200.50, "MSFT": 250.00}
print(stock_prices)
```

```
{'AAPL': 150.25, 'GOOGL': 1200.5, 'MSFT': 250.0}
```

You access the value for a specific key using square brackets `[ ]`, and modify the value for a key using the assignment operator `=`. You add new key-value pairs to a dictionary using a new key and assignment operator and remove a key-value pair using the `del` keyword. The `len()` function returns the number of key-value pairs in a dictionary.

```
# Accessing elements in a dictionary
price_aapl = stock_prices["AAPL"]
print(f"Price for AAPL: {price_aapl:0.2f}")

# Modifying an element
stock_prices["GOOGL"] = 1205.00
print(f"Modified GOOGL: {stock_prices}")

# Adding a new element to the dictionary
stock_prices["AMZN"] = 3300.00
print(f"Added AMZN: {stock_prices}")

# Removing an element from the dictionary
del stock_prices["MSFT"]
print(f"Removed MSFT: {stock_prices}")

# Length of the dictionary
dict_length = len(stock_prices)
print(f"Length: {dict_length}")
```

```
Price for AAPL: 150.25
Modified GOOGL: {'AAPL': 150.25, 'GOOGL': 1205.0, 'MSFT': 250.0}
Added AMZN: {'AAPL': 150.25, 'GOOGL': 1205.0, 'MSFT': 250.0, 'AMZN': 3300.0}
Removed MSFT: {'AAPL': 150.25, 'GOOGL': 1205.0, 'AMZN': 3300.0}
Length: 3
```

The `collection` module from Python's standard library provides many other data structures such as `defaultdict`, `OrderedDict`, `Counter`, and `deque`. You can learn more about these data structures in the Python documentation.

2.9. Comparison operators and branching

2.9.1. Comparison operators

Python provides several comparison operators that allow you to compare values and evaluate expressions. Comparison operators can be used with various data types, such as numbers, strings, or even complex data

structures, and return a boolean value (True or False). Table 2.9 lists the comparison operators available in Python.

Table 2.9.: Comparison operators in Python

Operator	Name	Example	Result
`==`	Equal	`1 == 2`	`False`
`!=`	Not equal	`1 != 2`	`True`
`>`	Greater than	`1 > 2`	`False`
`<`	Less than	`1 < 2`	`True`
`>=`	Greater or equal	`1 >= 2`	`False`
`<=`	Less or equal	`1 <= 2`	`True`
`in`	Membership	`1 in [1, 2, 3]`	`True`
`is`	Identity comparison	`1 is None`	`False`

To create more complex conditions, you can chain multiple comparisons in a single expression using logical operators like `and`, `or`, or `not`. The result of a logical operator is a boolean value (`True` or False). Table 2.10 lists the logical operators available in Python.

Table 2.10.: Logical operators in Python

a	b	a and b	a or b	not a
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

⚠ `&` and `|` are bitwise operators, not logical operators

Python also provides bitwise operators that perform bitwise operations on integers. These operators are `&` (bitwise AND), `|` (bitwise OR), `^` (bitwise XOR), `~` (bitwise NOT), `<<` (bitwise left shift), and `>>` (bitwise right shift). Most casual Python users will not need to use these operators, but they can be confusing for new users due to their similar syntax to logical operators in other programming languages. To add to the confusion, popular Python libraries like NumPy and Pandas overload the bitwise operators to perform logical operations on arrays.
It is crucial for beginners to understand the distinction between logical and bitwise operators and to use the appropriate operators (which are usually `and`, `or`, or `not`) based on their intended purpose to ensure the desired logical evaluations are achieved.

Longer expressions can be grouped using parentheses to ensure the desired order of operations. For example, `a and b or c` is equivalent to `(a and b) or c`, whereas `a and (b or c)` is different. Python does not offer a built-in *exclusive or* (XOR) operator, but it can be achieved using a combination of other operators.

```
def xor(a, b):
    return (a and not b) or (not a and b)


print(f"xor(True, True)) = {xor(True, True)}")
print(f"xor(True, False)) = {xor(True, False)}")
print(f"xor(False, True)) = {xor(False, True)}")
print(f"xor(False, False)) = {xor(False, False)}")
```

```
xor(True, True)) = False
xor(True, False)) = True
xor(False, True)) = True
xor(False, False)) = False
```

2.9.2. Branching

Branching allows your code to execute different actions based on specific conditions. The primary branching construct in Python is the `if` statement, which can be combined with `elif` (short for "else if") and `else` clauses to create more complex decision-making structures.

Like other compound statements in Python, the `if` statement uses indentation to group statements together. The general syntax for an `if` statement is to start with the `if` keyword followed by a condition, then a colon (`:`), and, finally, an indented block of code that will be executed if the condition evaluates to `True`. The `elif` and `else` clauses are optional and can be used to specify additional conditions and code blocks to execute if the initial condition evaluates to `False`. The `elif` clause is used to chain multiple conditions together, whereas the `else` clause is used to specify a default code block to execute if none of the previous conditions evaluate to `True`.

```
price = 150

if price > 100:
    print("The stock price is high.")
```

```
The stock price is high.
```

In the previous example, the code block is executed because the `price = 150`, therefore the condition `price > 100` evaluates to `True`.

We can add an `else` clause to specify a default code block to execute if the condition evaluates to `False`.

```
price = 50

if price > 100:
    print("The stock price is high.")
else:
    print("The stock price is low.")
```

```
The stock price is low.
```

We can add an `elif` clause to specify additional conditions to evaluate if the initial condition evaluates to `False`. The `elif` clause can be used multiple times to chain multiple conditions together. The `elif` clause is optional, but if it is used, it must come before the `else` clause. The `else` clause is also optional, but if it is used, it must come last.

In all cases, the code block associated with the first condition that evaluates to `True` will be executed, and the remaining conditions will be skipped. If none of the conditions evaluate to `True`, then the code block associated with the `else` clause will be executed. If there is no `else` clause, then nothing will be executed.

```
price = 75

if price > 100:
    print("The stock price is high.")
elif price > 50:
    print("The stock price is moderate.")
else:
    print("The stock price is low.")
```

```
The stock price is moderate.
```

You can nest `if` statements inside other `if` statements to create more complex branching structures. The code block associated with the nested `if` statement must be indented further than the outer `if` statement. The nested `if` statement will only be evaluated if the condition associated with the outer `if` statement evaluates to `True`. When reading nested `if` statements, it is helpful to read from the top down and to keep track of the indentation level to understand which code blocks are associated with which conditions.

```
price = 150
volume = 1000000

if price > 100:
    if volume > 500000:
        print("The stock price is high and has high volume.")
    else:
        print("The stock price is high but has low volume.")
else:
    print("The stock price is not high.")
```

```
The stock price is high and has high volume.
```

Conditions can be combined using the logical operators `and`, `or`, and `not` to create more complex conditions.

```
price = 150
volume = 1000000

if price > 100 and volume > 500000:
    print("The stock price is high and has high volume.")
elif price > 100 or volume > 500000:
    print("The stock price is high or has high volume.")
else:
    print("The stock price is not high and has low volume.")
```

```
The stock price is high and has high volume.
```

2.9.3. Conditional assignment

Python provides a convenient shorthand for assigning a value to a variable based on a condition. This is known as conditional assignment. The syntax for conditional assignment is `variable = value1 if condition else value2`. If the condition evaluates to True, the variable is assigned `value1`; otherwise, it is assigned `value2`.

```
price = 150

message = "The stock price is high." if price > 100 else "The stock price is low."
print(message)
```

```
The stock price is high.
```

2.10. Typing

Python is a dynamically typed language. This means that you don't have to specify the type of a variable when you define it. The Python interpreter will automatically infer the type based on the value assigned to the variable.

Python also supports optional type annotations, also called *type hints*, since version 3.5. This allows you to specify the types of variables, function arguments, and return values to improve code readability and catch potential errors early. The Python interpreter will ignore the type annotations and run the code normally. However, you can use external tools like mypy to analyze the code and check for type errors, and modern IDEs like VS Code provide built-in support for type checking.

```
ticker: str = "AAPL"
```

```
stock_prices: list[float] = [150.25, 1200.50, 250.00]
```

```python
# Old version, not needed since Python 3.9

from typing import List

stock_prices: List[float] = [150.25, 1200.50, 250.00]
```

```python
# You can also specify function parameters and return types:


def calculate_profit(revenue: float, expenses: float) -> float:
    return revenue - expenses


revenue = 1000.00
expenses = 500.00
profit = calculate_profit(revenue, expenses)
```

Type hints in Visual Studio Code

Type hints are not required to run Python code, but they can be very useful to improve code readability and catch potential errors early. Modern IDEs like VS Code provide built-in support for type checking that you can enable.
I find this quite overwhelming, so I prefer to enable type-checking only when needed. However, VS Code still uses type hints to provide useful features like hover info.

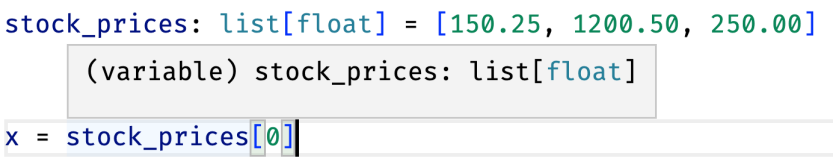


Figure 2.3.: Tooltip when hovering over variable.

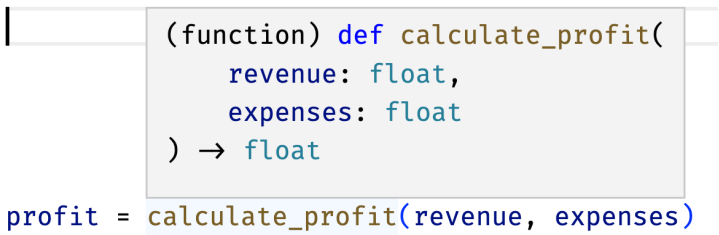


Figure 2.4.: Tooltip when hovering over function.

The `typing` module provides a set of special types that can be used in type hints. Here are some of the most commonly used ones:

- `Any`: Any type
- `Optional`: An optional value (can be `None`)
- `Callable`: A function
- `Iterable`: An iterable object (e.g., list, tuple, set)

2.11. Functions: parameters and return values

Functions help you organize and structure your code by encapsulating specific tasks or calculations. They allow you to define input parameters, perform operations, and return the results, making your code more flexible and maintainable. We have already written simple functions in Section 2.6; we will now look in more detail at how to define parameters and return values.

> **Type hints in examples**
>
> In the examples below, I use type hints to indicate the type of the function parameters and return values. Type hints are not required to run Python code, but using them is a good practice as they provide helpful information to other developers (including your future self!) and tools such as linters and type checkers.

2.11.1. Parameters

Parameters are variables defined within the function signature, enabling you to pass input values to the function when it is called. Parameters can have a default value assigned to them when no value is provided during the function call. Default values can make your functions more flexible and easy to use. Using the `*args` and `**kwargs` syntax, you can pass a variable number of positional or keyword arguments to a function, providing greater flexibility for handling different input scenarios.[2]

> **Parameter vs. argument**
>
> The terms *function parameter* and *argument* refer to different concepts related to functions.
> **Function parameter:** A function parameter is a variable defined in the function's definition or signature. It represents a value that the function expects to receive when it is called. Parameters act as placeholders for the actual values that will be passed as arguments when the function is invoked.
> **Argument:** An argument is the actual value that is passed to a function when it is called. It corresponds to a specific function parameter and provides the actual data or input that the function operates on. Arguments are supplied in the function call, within parentheses, and are passed to the corresponding function parameters based on their position or using keyword arguments.

[2] I do not recommend using variable-length parameters unless you have a specific need for them, as they can make your code more complex and harder to read and understand.

```
def calculate_roi(investment: float, profit: float) -> float:
    return (profit / investment) * 100.0


investment = 2000.00
profit = 500.00
roi = calculate_roi(investment, profit)
print(roi)
```

```
25.0
```

In the previous example, variables `investment` and `profit` are passed to the function calculate_roi() in the same order as they are defined in the function definition. This is called positional arguments. The names of the variables do not matter, only the order in which they are passed to the function. If we invert the order of the variables, the result will be different.

```
bad_roi = calculate_roi(profit, investment)
print(bad_roi)
```

```
400.0
```

Positional arguments can be confusing when the function has many parameters or when the order of the arguments is not obvious. To avoid this, you can use keyword arguments.

```
roi1 = calculate_roi(investment=2000.00, profit=500.00)
print(roi1)

roi2 = calculate_roi(profit=500.00, investment=2000.00)
print(roi2)

roi3 = calculate_roi(profit=profit, investment=investment)
print(roi3)
```

```
25.0
25.0
25.0
```

Note that the name of the original variables does not matter, only the name of the parameters in the function definition. In the last example, we use the same names for the variables and the parameters, but the Python interpreter does not care about that. The following code is equivalent to the previous one:

```
x = 2000.00
y = 500.00

roi4 = calculate_roi(profit=y, investment=x)
print(roi4)
```

```
25.0
```

You can also mix positional and keyword arguments. However, positional arguments must always come before keyword arguments.

```
roi4 = calculate_roi(investment, profit=profit)
print(roi4)
```

```
25.0
```

Default parameters are useful when you want to provide a default value for a parameter, which is used when no value is provided during the function call. This makes your functions more flexible and easy to use, as you can omit parameters that have a default value.

To define a default parameter, assign a value to the parameter in the function definition using `=`. When the function is called, the default value will be used if no value is provided for that parameter. If a value is provided, it will override the default value.

```
def calculate_present_value(cashflow: float, discount_rate: float = 0.1) -> float:
    return cashflow / (1 + discount_rate)


# Uses default discount_rate of 10%
pv = calculate_present_value(cashflow=100.00)
print(f"Present Value: {pv}")

# Uses discount_rate of 5%
pv2 = calculate_present_value(cashflow=100.00, discount_rate=0.05)
print(f"Present Value: {pv2}")
```

```
Present Value: 90.9090909090909
Present Value: 95.23809523809524
```

Parameters with default values must come after parameters without default values. Otherwise, the function call will raise a `SyntaxError`. When you call a function with default parameters, you can omit any parameters that have a default value. However, when you omit a parameter, you must use keyword parameters to specify the values for the parameters that follow it.

2.11.2. Passing arguments: peek under the hood

Python uses a mechanism called "passing arguments by assignment." In simple terms, this means that when you pass an argument to a function, a copy of the reference to the object is made and assigned to the function parameter.

When an immutable object (like a number, string, or tuple) is passed as an argument, it is effectively passed by value. Any modifications made to the parameter within the function do not affect the original object outside the function. Changes to the parameter create a new object rather than modifying the original one.

On the other hand, when a mutable object (like a list or dictionary) is passed as an argument, it is effectively passed by reference. Any modifications made to the parameter within the function will affect the original object outside the function. This is because both the parameter and the original object refer to the same memory location, so changes are reflected in both.

2.11.3. Return values

Functions can return a value, multiple values, or no value at all. To return a value, use the `return` keyword followed by the value or expression you want to return. If a function doesn't include a return statement, it will implicitly return `None`. A function can contain multiple return statements, but the execution of the function will stop as soon as any of them is reached.

A function can return a single value, such as a number, string, or a more complex data structure. A function can also return multiple values, typically in the form of a tuple. This is useful when you need to return several related results from a single function call. If a function doesn't explicitly return a value using the `return` keyword, it will implicitly return `None` when it reaches the end of the function body.

```
def calculate_mean_and_median(numbers: list[float]) -> tuple[float, float]:
    mean = sum(numbers) / len(numbers)

    # Sort the numbers in ascending order using the sorted() function
    sorted_numbers = sorted(numbers)
    length = len(sorted_numbers)

    if length % 2 == 0:
        median = (sorted_numbers[length // 2 - 1] + sorted_numbers[length // 2]) / 2
    else:
        median = sorted_numbers[length // 2]

    return mean, median


prices = [150.25, 1200.50, 250.00]
mean, median = calculate_mean_and_median(prices)
print(f"Mean: {mean}, Median: {median}")
```

```
Mean: 533.5833333333334, Median: 250.0
```

2.11.4. Scope

In Python, the scope of a variable refers to the region of a program where the variable is accessible and can be referenced. The scope determines the visibility and lifetime of a variable, including where it can be accessed and modified.

When using Python functions, there are two main scopes to consider:

1. **Local scope (function scope):** Variables defined within a function have local scope. They are accessible only within the function where they are defined. Local variables are created when the function is called and destroyed when the function execution completes or reaches a return statement. They are not accessible outside the function.
2. **Global scope (module scope):** Variables defined outside of any function in the interactive window or in a Python script, have global scope. They are accessible from anywhere within the program, including all functions.

When a function is called, it creates a new local scope, which is independent of the global scope. Inside the function, the local scope takes precedence over the global scope. If a variable is referenced within a function, Python first checks the local scope for its existence. If not found, it then searches the global scope.

```
# Global variable
message = "Hello"
x = 123


def say_hello(m: str):
    # Local variable
    message = "Hello, World!"
    print(f"local message = {message}")

    # Local variable, copied from the argument
    print(f"local m = {m}")

    # print(f"global x inside function = {x}")
                                                                                                    ①


    # Local variable
    x = len(m)
    print(f"local x = {x}")

    return x
```

```
y = say_hello(message)

print(f"global message = {message}")
print(f"global x after function = {x}")
print(f"global y = {y}")
```

① This will access the global `x` if there is no local variable with the same name. In this specific case, it will cause an error because `x` is actually defined later in the function.

```
local message = Hello, World!
local m = Hello
local x = 5
global message = Hello
global x after function = 123
global y = 5
```

If you want to modify a global variable from within a function, you can use the `global` keyword to indicate that the variable being referred to is a global variable rather than creating a new local variable. However, this is generally not recommended, as it can lead to unexpected side effects and make the code difficult to understand and debug.

It is important to carefully manage variable scope to avoid naming conflicts and unintended side effects. Understanding the scope of variables helps in organizing and managing data within functions and modules effectively.

2.12. Loops

Loops enable you to easily perform repetitive tasks or iterate through data structures, such as sequences and collections. Python provides two primary loop constructs: the `for` loop and the `while` loop.

2.12.1. `for` loops

For loops in Python are used to iterate over a sequence (e.g., a list, tuple, or string) or other iterable objects. The loop iterates through each item in the sequence, executing a block of code for each item. The 'for' loop has the following syntax:

```
for item in sequence:
    # code to execute for each item in the sequence
```

As for function bodies and conditional statements, the code block in a loop must be indented.

2.12.2. `range()` function

The built-in `range()` function in Python is often used in conjunction with `for` loops to generate a sequence of numbers. This function can be used to create a range of numbers with a specified start, end, and step size. The syntax for the range function is:

```
range(start, stop, step)
```

The 'start' and 'step' arguments are optional, with default values of 0 and 1, respectively. The 'stop' argument is required and defines the upper limit of the range (exclusive).

```
for i in range(5):                                                    ①
    print(i)
```

① The range(5) function generates a sequence of numbers from 0 to 4 (inclusive) with a step of 1.

```
0
1
2
3
4
```

```
for i in range(2, 7):                                                 ①
    print(i)
```

① The range(2, 7) function generates a sequence of numbers from 2 to 6 (inclusive) with a step of 1.

```
2
3
4
5
6
```

```
for i in range(1, 11, 2):                                             ①
    print(i)
```

① The range(1, 11, 2) function generates a sequence of numbers from 1 to 10 (inclusive) with a step of 2.

```
1
3
5
7
9
```

You can use a negative step size to generate a sequence of numbers in reverse order.

```
for i in range(5, 0, -1):                                                    ①
    print(i)
```

① The range(5, 0, -1) function generates a sequence of numbers from 5 to 1 (inclusive) with a step of -1 (decreasing).

```
5
4
3
2
1
```

You can use the `range()` function to generate a sequence of numbers and iterate through them using a `for` loop to execute some code for each number in the sequence. In this example, we use the `range()` function to generate a sequence of numbers from 1 to 5 (inclusive) and calculate the compound interest for each year of an investment.

```
principal = 1000
rate = 0.05

for year in range(1, 6):
    interest = principal * ((1 + rate) ** year - 1)
    print(f"Year {year}: Interest = {interest:.2f}")
```

```
Year 1: Interest = 50.00
Year 2: Interest = 102.50
Year 3: Interest = 157.63
Year 4: Interest = 215.51
Year 5: Interest = 276.28
```

2.12.3. `continue` and `break` statements

You can use the `continue` and `break` statements to control the flow of a for loop. The `continue` statement skips the current iteration and continues with the next one. The `break` statement terminates the loop and transfers execution to the statement immediately following the loop.

```
for i in range(10):
    if i == 3:
        continue                                                             ①
    elif i == 5:
        break                                                                ②
    print(i)
```

① Skip the rest of the code in the loop and go to the next iteration
② Exit the loop

```
0
1
2
4
```

2.12.4. `for` loops with lists

You can also loop over a collection of items using the `for` loop. For example, you can loop over a list of numbers to calculate the sum of all numbers in the list.

```
daily_profit_losses = [1500, 1200, 1800, 2300, 900]

total_pl = 0
for pl in daily_profit_losses:
    total_pl += pl

print(f"Total P&L for the period: {total_pl}")
```

```
Total P&L for the period: 7700
```

> **Built-in functions**
>
> Python provides several built-in functions that can be used to perform common tasks. For example, the `sum()` function can be used to calculate the sum of all numbers in a list.
>
> ```
> daily_profit_losses = [1500, 1200, 1800, 2300, 900]
> total_pl = sum(daily_profit_losses)
> print(f"Total P&L for the period: {total_pl}")
> ```
>
> ```
> Total P&L for the period: 7700
> ```

You can combine two lists of the same length using the `zip()` built-in function to loop over both lists at the same time.

```
stock_prices = [150.25, 1200.50, 250.00]
quantities = [10, 5, 20]

total_value = 0
for price, quantity in zip(stock_prices, quantities):
    total_value += price * quantity
```

```
print(f"Total value of the portfolio: {total_value:.2f}")
```

```
Total value of the portfolio: 12505.00
```

You can use the `enumerate()` function to loop over a list and get the index of each item in the list.

```
cash_flows = [100, 200, 300, 400, 500]

discount_rate = 0.1

present_values = []
for year, cash_flow in enumerate(cash_flows):
    present_value = cash_flow / (1 + discount_rate) ** year
    print(f"Year {year}: Present Value = {present_value:.2f}")
```

```
Year 0: Present Value = 100.00
Year 1: Present Value = 181.82
Year 2: Present Value = 247.93
Year 3: Present Value = 300.53
Year 4: Present Value = 341.51
```

Iterables and iterators

In Python, an iterable is an object capable of returning its elements one at a time, such as a list, tuple, or string. An iterator is an object that keeps track of its current position within an iterable and provides a way to access the next element when required.
Many built-in data types and functions return values in Python are iterables or iterators. For example, the `range()` function returns an iterator that produces a sequence of numbers. The `zip()` function returns an iterator that produces tuples containing elements from the input iterables. The `enumerate()` function returns an iterator that produces tuples containing the index and value of each item in the input iterable. There are many benefits to iterators, such as better memory efficiency and allowing you to work with infinite sequences, such as the sequence of all prime numbers. However, you can't print the result of calling an iterator like `zip()` directly. Instead, you must convert the iterator to a list or another collection using the `list()` function.

```
stock_prices = [150.25, 1200.50, 250.00]
quantities = [10, 5, 20]

zipped = zip(stock_prices, quantities)

print(f"zipped: {zipped}")
print(f"list(zipped): {list(zipped)}")
```

```
zipped: <zip object at 0x13689df80>
list(zipped): [(150.25, 10), (1200.5, 5), (250.0, 20)]
```

2.12.5. Nested `for` loops

You can nest loops. The inner loop will be executed one time for each iteration of the outer loop.

In this example, we have a list of products, each with a name, per-item profit margin, and a list of quantities sold at different times. The outer loop iterates through each product, while the inner loop iterates through the quantities for each product. The product's profit is calculated by multiplying its margin by the quantity sold and adding it to the `product_profit` variable. The `total_profit` variable accumulates the profit for all products.

```
products = [
    {"name": "Product A", "margin": 10, "quantities": [5, 10, 15]},
    {"name": "Product B", "margin": 20, "quantities": [2, 4, 6]},
    {"name": "Product C", "margin": 30, "quantities": [1, 3, 5]},
]

total_profit = 0

for product in products:
    product_profit = 0
    for quantity in product["quantities"]:
        product_profit += product["margin"] * quantity
    total_profit += product_profit
    print(f"Profit for {product['name']}: {product_profit}")

print(f"Total profit: {total_profit}")
```

```
Profit for Product A: 300
Profit for Product B: 240
Profit for Product C: 270
Total profit: 810
```

2.12.6. `while` loops

While loops are used to repeatedly execute a block of code as long as a specified condition is True. The `while` loop has the following syntax:

```
while condition:
    # code to execute while the condition is True
```

In this example, we use a while loop to calculate the number of years it takes for an investment to double at a given interest rate.

```
principal = 1000
rate = 0.05
balance = principal
target = principal * 2
years = 0

while balance < target:
    interest = balance * rate
    balance += interest
    years += 1

print(f"It takes {years} years for the investment to double.")
```

```
It takes 15 years for the investment to double.
```

You can use the `continue` statement to skip the rest of the code in the current iteration and continue with the next one and the `break` statement to exit a while loop before the condition becomes `False`.

2.13. List and dictionary comprehensions

Python supports list and dictionary comprehensions, which allow you to create lists and dictionaries in a concise and efficient manner by transforming or filtering items from another iterable, such as a range, a tuple or another list. Comprehensions can be confusing at first, but they are a powerful tool worth learning.

2.13.1. List comprehensions

List comprehension syntax consists of square brackets containing an expression followed by a for clause, then zero or more for or if clauses. The expression can be anything, meaning you can put in all kinds of objects in lists.

```
# This is perfectly valid:
squared1 = []
for x in range(10):
    squared1.append(x * x)

print(squared1)

# This is much shorter!
squared2 = [x * x for x in range(10)]
```

```
print(squared2)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

You can use list comprehensions to transform items from a list into a new list using complex expressions.

```
# Calculate the percentage change for a list of stock prices
stock_prices = [150.25, 1200.50, 250.00, 175.00, 305.75]
percentage_changes = [
    (stock_prices[i + 1] - stock_prices[i]) / stock_prices[i] * 100
    for i in range(len(stock_prices) - 1)
]

print("Percentage changes:", percentage_changes)
```

```
Percentage changes: [699.0016638935108, -79.17534360683048, -30.0, 74.71428571428571]
```

2.13.2. Dictionary comprehensions

Similar to list comprehensions, dictionary comprehensions use a single line of code to define the structure of the new dictionary.

```
# Create a dictionary mapping numbers to their squares
squares = {i: i**2 for i in range(1, 6)}

print(squares)
```

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

2.13.3. Filtering and transforming

You can use conditional statements in list and dictionary comprehensions to filter items from the source iterable.

```
# Create a dictionary mapping even numbers to their cubes
even_cubes = {i: i**3 for i in range(1, 6) if i % 2 == 0}

print(even_cubes)
```

```
{2: 8, 4: 64}
```

You can transform the items in the source iterable before adding them to the new list or dictionary.

```
# List of stock symbols and prices
stock_data = [("AAPL", 150.25), ("GOOG", 1200.50), ("MSFT", 250.00)]

# Create a dictionary mapping lowercase stock symbols to their prices
stocks = {symbol.lower(): price for symbol, price in stock_data}

print(stocks)
```

```
{'aapl': 150.25, 'goog': 1200.5, 'msft': 250.0}
```

2.13.4. Nested comprehensions

You can nest comprehensions inside other comprehensions to create complex data structures.

```
# Create a list of (stock, prices) tuples
stock_data = [
    ("AAPL", [150.25, 150.50, 150.75]),
    ("GOOG", [1200.50, 1201.00]),
    ("MSFT", [250.00, 250.25, 250.50, 250.75]),
]

stocks = [(symbol, price) for symbol, prices in stock_data for price in prices]  ①
print(stocks)
```

① The comprehension is evaluated from left to right, so the `prices` variable is available in the second `for` clause.

```
[('AAPL', 150.25), ('AAPL', 150.5), ('AAPL', 150.75), ('GOOG', 1200.5), ('GOOG', 1201.0), ('MSFT', 250.0), ('MSFT', 250.25), ('
```

> **Nested comprehensions vs readability**
>
> Nested comprehensions with more than two levels can be difficult to read, so you should avoid them if possible. If you find yourself nesting comprehensions, it's probably a good idea to use a regular for loop instead. Remember, code readability is more important than brevity.

2.14. Working with files and the `with` statement

So far, we've focused on working with data in memory. But in practice, you'll often need to read data from files or save results to disk. Python provides powerful tools for working with files, and understanding them requires introducing an important concept: context managers and the `with` statement.

2.14.1. The `with` statement

When you open a file, you acquire a resource that needs to be properly released when you're done. If you forget to close a file, you can run into problems: data might not be written to disk, you could run out of file handles, or other programs might not be able to access the file.

The `with` statement solves this problem elegantly. It ensures that cleanup happens automatically, even if an error occurs while you're working with the resource. Here's the basic pattern:

```
with some_resource as name:
    # work with the resource
    ...
# resource is automatically cleaned up here
```

The `with` statement works with any object that implements the context manager protocol. These objects define what should happen when entering the `with` block (acquiring the resource) and when exiting it (releasing the resource). Many built-in Python types support this, including files, database connections, and locks.

> Why use `with`?
>
> You might be tempted to open and close files manually:
>
> ```
> f = open("data.txt")
> content = f.read()
> f.close()  # Easy to forget!
> ```
>
> This works, but it's risky. If an error occurs before `close()` is called, the file might remain open. The `with` statement guarantees cleanup:
>
> ```
> with open("data.txt") as f:
>     content = f.read()
> # File is automatically closed, even if an error occurred
> ```
>
> Always prefer `with` when working with files and other resources.

2.14.2. Reading and writing files with `open()`

The built-in `open()` function is Python's standard way to work with files. It takes a file path and a mode, and returns a file object that you can read from or write to.

Common modes include:

- `"r"` - read mode (default)
- `"w"` - write mode (creates a new file or overwrites existing)
- `"a"` - append mode (adds to the end of an existing file)

- `"x"` - exclusive creation (fails if the file already exists)

For text files, you can read the entire contents at once or process line by line:

```
# First, let's create a sample file to work with
sample_data = """Date,Ticker,Price
2024-01-02,AAPL,185.64
2024-01-02,MSFT,374.58
2024-01-03,AAPL,184.25"""

with open("sample_prices.csv", "w") as f:
    f.write(sample_data)

# Now read it back
with open("sample_prices.csv", "r") as f:
    content = f.read()

print(content)
```

```
Date,Ticker,Price
2024-01-02,AAPL,185.64
2024-01-02,MSFT,374.58
2024-01-03,AAPL,184.25
```

For processing files line by line, you can iterate directly over the file object:

```
with open("sample_prices.csv", "r") as f:
    for line in f:
        print(f"Line: {line.strip()}")  # strip() removes the newline character
```

```
Line: Date,Ticker,Price
Line: 2024-01-02,AAPL,185.64
Line: 2024-01-02,MSFT,374.58
Line: 2024-01-03,AAPL,184.25
```

This approach is memory-efficient for large files because it only loads one line at a time.

> **File encoding**
>
> When working with text files, be aware of character encoding. The default encoding depends on your system, which can cause problems when sharing files. It's good practice to specify the encoding explicitly:
>
> ```
> with open("data.txt", "r", encoding="utf-8") as f:
>     content = f.read()
> ```

> UTF-8 is the most common encoding for modern text files and handles international characters well.

2.14.3. Working with paths using `pathlib`

Python's `pathlib` module provides a modern, object-oriented approach to working with file paths. Instead of manipulating path strings directly, you work with `Path` objects that understand the structure of file paths on your operating system.

```
from pathlib import Path

# Create a Path object
data_dir = Path("data")
file_path = data_dir / "prices.csv"  # Use / to join paths

print(f"Full path: {file_path}")
print(f"Parent directory: {file_path.parent}")
print(f"File name: {file_path.name}")
print(f"File extension: {file_path.suffix}")
```

```
Full path: data/prices.csv
Parent directory: data
File name: prices.csv
File extension: .csv
```

The `/` operator for joining paths is particularly elegant. It works correctly across different operating systems, so you don't need to worry about whether to use forward slashes or backslashes.

`Path` objects have convenient methods for common file operations:

```
from pathlib import Path

# Check if a file exists
sample_file = Path("sample_prices.csv")
print(f"File exists: {sample_file.exists()}")

# Read text directly (no need for open!)
if sample_file.exists():
    content = sample_file.read_text()
    print(f"First 50 characters: {content[:50]}...")
```

```
File exists: True
First 50 characters: Date,Ticker,Price
2024-01-02,AAPL,185.64
2024-01-0...
```

For simple read and write operations, Path methods like read_text() and write_text() are often more concise than using open():

```
from pathlib import Path

# Write text to a file
output_file = Path("output.txt")
output_file.write_text("Hello from pathlib!")

# Read it back
print(output_file.read_text())

# Clean up
output_file.unlink()  # Delete the file
```

```
Hello from pathlib!
```

When you need more control, such as reading line by line or appending to a file, Path objects work seamlessly with open():

```
from pathlib import Path

file_path = Path("sample_prices.csv")

with file_path.open("r") as f:
    header = f.readline()
    print(f"Header: {header.strip()}")
```

```
Header: Date,Ticker,Price
```

Path vs string paths

While you can use plain strings for file paths, Path objects offer several advantages:

1. **Cross-platform compatibility**: Path handles the differences between Windows (\) and Unix (/) automatically
2. **Clear intent**: Code using Path clearly signals you're working with file paths
3. **Convenient methods**: Methods like exists(), is_dir(), suffix, and stem make common operations easy
4. **Composability**: The / operator makes building paths readable and safe

Most modern Python code uses pathlib for path manipulation. You'll see it throughout this book when working with files.

2.15. Pattern matching

Pattern matching is a powerful feature introduced in Python 3.10. It allows you to match the structure of data and execute code based on the shape and contents of that data. It is particularly useful for working with complex data structures and can lead to cleaner and more readable code.

> **Python 3.10+ only**
>
> Pattern matching is a new feature introduced in Python 3.10, released on October 4, 2021. If you're using an older version of Python, or your code will be running on a system with an older version of Python, you should avoid using pattern matching.

Pattern matching is implemented using the `match` statement, which is similar to a switch-case statement in other languages but with more advanced capabilities. The `match` statement takes an expression and a series of cases. Each `case` is a pattern that is matched against the expression in turn. If the pattern matches, the code in that case is executed, otherwise, the next case is checked. The `match` statement can also have a case with the wildcard pattern _, which will always match. If no pattern matches, a `MatchError` is raised.

```
def process_transaction(transaction: tuple):
    match transaction:
        case ("deposit", amount):
            print(f"Deposit: {amount:.2f}")
        case ("withdraw", amount):
            print(f"Withdraw: {amount:.2f}")
        case ("transfer", amount, recipient):
            print(f"Transfer {amount:.2f} to {recipient}")
        case _:
            print("Unknown transaction")

process_transaction(("deposit", 1000))
process_transaction(("burn", 100.00))
process_transaction(("transfer", 500.00, "John Doe"))
process_transaction(("withdraw", 250.00))
```

```
Deposit: 1000.00
Unknown transaction
Transfer 500.00 to John Doe
Withdraw: 250.00
```

2.16. Additional resources

- Python 3.14 documentation
- Lubanovic, Bill. *Introducing Python*, 2nd Edition, O'Reilly Media, Inc., 2019

3. Object-Oriented Programming Basics

Object-Oriented Programming (OOP) is a programming paradigm that organizes code around "objects" rather than functions and logic. While Python fully supports OOP, you don't need to use it for everything you do. In fact, for many data analysis tasks, a procedural or functional approach is simpler and more appropriate.

That said, understanding the basics of OOP is valuable for several reasons. First, many of the libraries you'll use in empirical finance are built using OOP principles, so understanding these concepts will help you use them more effectively. Second, there are certain situations in research code where OOP can make your code cleaner, more organized, and easier to maintain. Finally, OOP provides a way to model real-world entities and relationships in your code, which can be particularly useful when working with financial concepts like trades and limit order books.

In this chapter, we'll cover the fundamentals of OOP in Python, focusing on practical applications relevant to empirical finance. We'll start with the basic concepts of classes and objects, then discuss when OOP is genuinely useful in research code, and finally introduce Python's data classes, which provide a streamlined way to work with structured data.

> **i** A Pragmatic Approach
>
> This chapter takes a pragmatic approach to OOP. We won't cover every feature or delve into advanced design patterns. Instead, we'll focus on the subset of OOP that's most useful for research code in finance. If you find yourself writing highly object-oriented code with deep inheritance hierarchies, you are most likely overengineering your research scripts.

3.1. Classes and Objects

At its core, object-oriented programming is about creating custom data types that bundle together related data and the functions that operate on that data. Let's break down the key concepts.

3.1.1. What is a Class?

A class is essentially a blueprint or template for creating objects. It defines what data an object will hold (attributes) and what operations can be performed on that data (methods). Think of a class as a cookie cutter and objects as the cookies made from that cutter.

Let's start with a simple example. Suppose you're working on a project that involves tracking individual trades. Each trade has certain properties: a ticker symbol, a quantity, a price, and whether it's a buy or sell. You could represent each trade as a dictionary:

```
trade1 = {
    "ticker": "AAPL",
    "quantity": 100,
    "price": 150.50,
    "side": "buy"
}

trade2 = {
    "ticker": "MSFT",
    "quantity": 50,
    "price": 280.25,
    "side": "sell"
}
```

This works, but it has some limitations. There's no guarantee that every trade dictionary has the same keys. You might accidentally misspell a key, or forget to include one. And if you want to calculate the total value of a trade, you need to write that logic separately.

A class provides a better solution. Here's how we might define a `Trade` class:

```
class Trade:
    def __init__(self, ticker: str, quantity: int, price: float, side: str):          ①
        self.ticker = ticker                                                           ②
        self.quantity = quantity
        self.price = price
        self.side = side

    def value(self) -> float:                                                          ③
        """Calculate the total value of the trade."""
        return self.quantity * self.price

    def __repr__(self) -> str:                                                         ④
        """Return a string representation of the trade."""
        return f"Trade({self.ticker}, {self.quantity}, ${self.price}, {self.side})"
```

① We define the class with `class Trade:`. By convention, class names use CamelCase. The `__init__` method is a special method called a *constructor*. It runs automatically when you create a new object from the class. The `self` parameter refers to the instance being created.

② Inside `__init__`, we set attributes on the object using `self.attribute_name`. These become the object's data.

③ The `value` method is a regular method that calculates the trade's total value. Like all methods, it takes `self` as its first parameter.

④ The `__repr__` method is another special method that defines how the object should be displayed. Methods that start and end with double underscores are called "dunder" (double underscore) methods or magic methods.

Now we can create trades as objects:

```
trade1 = Trade("AAPL", 100, 150.50, "buy")
trade2 = Trade("MSFT", 50, 280.25, "sell")

print(trade1)
print(f"Trade value: ${trade1.value():.2f}")
```

```
Trade(AAPL, 100, $150.5, buy)
Trade value: $15050.00
```

This is cleaner and more robust. Every `Trade` object is guaranteed to have the required attributes, and the logic for calculating value is bundled with the data.

3.1.2. Attributes and Methods

Let's clarify some terminology:

- **Attributes** are variables that belong to an object. In our example, `ticker`, `quantity`, `price`, and `side` are attributes.
- **Methods** are functions that belong to a class. They operate on the object's data. In our example, `value()` is a method.
- **Instance** refers to a specific object created from a class. `trade1` and `trade2` are instances of the `Trade` class.

You access attributes and call methods using dot notation:

```
print(trade1.ticker)      # Accessing an attribute
print(trade1.value())     # Calling a method
```

```
AAPL
15050.0
```

3.1.3. Adding More Functionality

Let's expand our `Trade` class to include more useful functionality. Suppose we want to compare trades and calculate profit and loss:

```python
class Trade:
    def __init__(self, ticker: str, quantity: int, price: float, side: str):
        self.ticker = ticker
        self.quantity = quantity
        self.price = price
        self.side = side

    def value(self) -> float:
        """Calculate the total value of the trade."""
        return self.quantity * self.price

    def pnl(self, current_price: float) -> float:                                    ①
        """Calculate profit/loss relative to a current price."""
        if self.side == "buy":
            return self.quantity * (current_price - self.price)
        else:  # sell
            return self.quantity * (self.price - current_price)

    def __repr__(self) -> str:
        return f"Trade({self.ticker}, {self.quantity}, ${self.price:.2f}, {self.side})"

    def __eq__(self, other: object) -> bool:                                         ②
        """Check if two trades are equal."""
        if not isinstance(other, Trade):
            return NotImplemented
        return (self.ticker == other.ticker and
                self.quantity == other.quantity and
                self.price == other.price and
                self.side == other.side)
```

① The `pnl` method calculates the profit or loss based on a current market price. The logic differs for buys (profit when price goes up) and sells (profit when price goes down).

② The `__eq__` method defines what it means for two `Trade` objects to be equal. Here, two trades are equal if all their attributes match.

The `__eq__` method is one of several special comparison methods Python supports. Others include `__ne__` (not equal, `!=`), `__lt__` (less than, `<`), `__le__` (less than or equal, `<=`), `__gt__` (greater than, `>`), and `__ge__` (greater than or equal, `>=`). For a complete list of these "rich comparison" methods and other special methods, see the Python documentation on basic customization.

Now we can do more with our trades:

```python
trade = Trade("AAPL", 100, 150.50, "buy")
print(f"Trade value: ${trade.value():.2f}")

# Calculate P&L at a current price
```

```
current_price = 160.00
pnl = trade.pnl(current_price)
print(f"P&L at ${current_price:.2f}: ${pnl:.2f}")

# Test equality
trade2 = Trade("AAPL", 100, 150.50, "buy")
trade3 = Trade("MSFT", 50, 280.25, "sell")
print(f"trade == trade2: {trade == trade2}")  # Same attributes
print(f"trade == trade3: {trade == trade3}")  # Different attributes
```

```
Trade value: $15050.00
P&L at $160.00: $950.00
trade == trade2: True
trade == trade3: False
```

3.1.4. A More Complex Example: Portfolio Class

Let's build a more sophisticated example: a `Portfolio` class that manages a collection of trades. This demonstrates how objects can contain other objects:

```
class Portfolio:
    def __init__(self, name):
        self.name = name
        self.trades = []

    def add_trade(self, trade):
        """Add a trade to the portfolio."""
        self.trades.append(trade)

    def total_value(self):
        """Calculate the total value of all trades."""
        return sum(trade.value() for trade in self.trades)

    def positions(self):
        """Calculate net position for each ticker."""
        positions = {}
        for trade in self.trades:
            if trade.ticker not in positions:
                positions[trade.ticker] = 0

            if trade.side == "buy":
                positions[trade.ticker] += trade.quantity
            else:  # sell
```

```
                positions[trade.ticker] -= trade.quantity

        return positions

    def __repr__(self):
        return f"Portfolio('{self.name}', {len(self.trades)} trades)"

    def summary(self):
        """Print a summary of the portfolio."""
        print(f"Portfolio: {self.name}")
        print(f"Total trades: {len(self.trades)}")
        print(f"Total value: ${self.total_value():.2f}")
        print("\nPositions:")
        for ticker, quantity in self.positions().items():
            print(f"  {ticker}: {quantity} shares")
```

Now we can use our `Portfolio` class:

```
# Create a portfolio
portfolio = Portfolio("My Research Portfolio")

# Add some trades
portfolio.add_trade(Trade("AAPL", 100, 150.50, "buy"))
portfolio.add_trade(Trade("AAPL", 50, 155.00, "buy"))
portfolio.add_trade(Trade("MSFT", 75, 280.25, "buy"))
portfolio.add_trade(Trade("AAPL", 25, 152.00, "sell"))

# View summary
portfolio.summary()
```

```
Portfolio: My Research Portfolio
Total trades: 4
Total value: $47618.75

Positions:
  AAPL: 125 shares
  MSFT: 75 shares
```

This example shows how OOP allows you to build up layers of abstraction. A `Portfolio` is a collection of `Trade` objects, and both have methods that make sense for their level of abstraction.

When to Use Classes vs. Functions

Don't create a class just to group functions together. If your class only has one or two methods and no meaningful state (attributes), it should probably just be a function. Classes are most useful when you need to maintain state across multiple operations.

3.1.5. String Representations: `__repr__` vs. `__str__`

Python provides two different methods for converting objects to strings: `__repr__` and `__str__`. Understanding the difference between them helps you write more useful classes.

- `__repr__` is meant to produce an unambiguous representation of the object, primarily for developers and debugging. Ideally, it should look like a valid Python expression that could recreate the object.
- `__str__` is meant to produce a readable, user-friendly string. It's what gets displayed when you use `print()` on an object.

If you only implement one, implement `__repr__`. Python will use it as a fallback for `__str__` if `__str__` isn't defined. Here's an example showing both:

```
class Trade:
    def __init__(self, ticker: str, quantity: int, price: float, side: str):
        self.ticker = ticker
        self.quantity = quantity
        self.price = price
        self.side = side

    def __repr__(self) -> str:
        """Unambiguous representation for developers."""
        return f"Trade({self.ticker!r}, {self.quantity}, {self.price}, {self.side!r})"

    def __str__(self) -> str:
        """User-friendly representation."""
        action = "Buy" if self.side == "buy" else "Sell"
        return f"{action} {self.quantity} shares of {self.ticker} @ ${self.price:.2f}"

trade = Trade("AAPL", 100, 150.50, "buy")

# __str__ is used by print()
print(trade)

# __repr__ is used in the REPL and for debugging
print(repr(trade))
```

```
Buy 100 shares of AAPL @ $150.50
Trade('AAPL', 100, 150.5, 'buy')
```

3.1.6. Rich Display in Jupyter and Quarto

Jupyter notebooks (and Quarto, a publishing system for creating documents from notebooks and other sources) support special methods for rich display. These methods allow your objects to render as HTML, Markdown, or LaTeX instead of plain text:

- `_repr_html_()` returns HTML that will be rendered in the notebook
- `_repr_markdown_()` returns Markdown text
- `_repr_latex_()` returns LaTeX for mathematical notation

Here's a simple example:

```python
class Trade:
    def __init__(self, ticker: str, quantity: int, price: float, side: str):
        self.ticker = ticker
        self.quantity = quantity
        self.price = price
        self.side = side

    def __repr__(self) -> str:
        return f"Trade({self.ticker!r}, {self.quantity}, {self.price}, {self.side!r})"

    def _repr_html_(self) -> str:
        """Rich HTML display for Jupyter/Quarto."""
        color = "green" if self.side == "buy" else "red"
        return f"""
        <div style="border: 1px solid #ccc; padding: 10px; border-radius: 5px; width: fit-content;">
            <strong>{self.ticker}</strong><br>
            <span style="color: {color};">{self.side.upper()}</span>
            {self.quantity} shares @ ${self.price:.2f}
        </div>
        """

    def _repr_latex_(self) -> str:
        """Rich LaTeX display for PDF output."""
        action = "Buy" if self.side == "buy" else "Sell"
        return (
            rf"\textbf{{{self.ticker}}}: "
            rf"{action} {self.quantity} shares @ \${self.price:.2f}"
        )
```

```
trade = Trade("AAPL", 100, 150.50, "buy")
trade  # In Jupyter/Quarto, this displays as formatted HTML or LaTeX
```

AAPL: Buy 100 shares @ $150.50

Many of the data analysis libraries you'll use later in this course, such as pandas, use these methods to display data frames as nicely formatted tables.

3.1.7. Class Variables vs. Instance Variables

So far, we've been working with instance variables—attributes that are unique to each object. Python also supports class variables, which are shared by all instances of a class:

```
class Trade:
    # Class variable
    commission_rate = 0.001  # 0.1% commission

    def __init__(self, ticker, quantity, price, side):
        # Instance variables
        self.ticker = ticker
        self.quantity = quantity
        self.price = price
        self.side = side

    def value(self):
        """Calculate the total value of the trade."""
        return self.quantity * self.price

    def net_value(self):
        """Calculate value after commission."""
        gross_value = self.value()
        commission = gross_value * Trade.commission_rate
        return gross_value - commission

    def __repr__(self):
        return f"Trade({self.ticker}, {self.quantity}, ${self.price:.2f}, {self.side})"

# All trades share the same commission rate
trade1 = Trade("AAPL", 100, 150.50, "buy")
trade2 = Trade("MSFT", 50, 280.25, "sell")

print(f"Trade 1 net value: ${trade1.net_value():.2f}")
print(f"Trade 2 net value: ${trade2.net_value():.2f}")
```

```
# Changing the class variable affects all instances
Trade.commission_rate = 0.002
print(f"Trade 1 net value (new rate): ${trade1.net_value():.2f}")
```

```
Trade 1 net value: $15034.95
Trade 2 net value: $13998.49
Trade 1 net value (new rate): $15019.90
```

Class variables are useful for values that should be consistent across all instances, like constants, default settings, or shared configuration.

3.1.8. Property Decorators

Sometimes you want to compute a value on-the-fly rather than storing it as an attribute. Python's `@property` decorator makes this look like a simple attribute access:

```
class Trade:
    def __init__(self, ticker, quantity, price, side):
        self.ticker = ticker
        self.quantity = quantity
        self.price = price
        self.side = side

    @property
    def value(self):
        """Calculate the total value of the trade."""
        return self.quantity * self.price

    @property
    def is_buy(self):
        """Check if this is a buy trade."""
        return self.side == "buy"

    def __repr__(self):
        return f"Trade({self.ticker}, {self.quantity}, ${self.price:.2f}, {self.side})"

trade = Trade("AAPL", 100, 150.50, "buy")

# No parentheses needed - looks like an attribute
print(f"Trade value: ${trade.value:.2f}")
print(f"Is buy: {trade.is_buy}")
```

```
Trade value: $15050.00
Is buy: True
```

The advantage of using `@property` is that it allows you to start with a simple attribute and later change it to a computed value without changing how the class is used. It also makes the code more readable when the value is conceptually an attribute rather than an action.

3.2. When OOP is Useful in Research Code

Now that you understand the basics of classes and objects, an important question remains: when should you actually use OOP in your research code?

The truth is, many data analysis tasks in empirical finance don't require OOP. A straightforward script that loads data, performs some analysis, and generates output can be perfectly fine without defining any classes. In fact, overusing OOP can make simple tasks more complicated than they need to be.

However, there are several scenarios where OOP becomes genuinely useful in research code.

3.2.1. Scenario 1: Managing Complex State

If you find yourself passing around many related variables to multiple functions, a class might be appropriate. Consider a simulation configuration that needs to be loaded, validated, and used across multiple functions:

Without OOP:

```
import json
from pathlib import Path

def load_simulation_config(filepath):
    with open(filepath) as f:
        data = json.load(f)
    return {
        "name": data["name"],
        "n_simulations": data["n_simulations"],
        "initial_value": data["initial_value"],
        "drift": data["drift"],
        "volatility": data["volatility"],
        "results": None
    }

def validate_config(config):
    if config["n_simulations"] <= 0:
        raise ValueError("n_simulations must be positive")
    if config["initial_value"] <= 0:
        raise ValueError("initial_value must be positive")
    if config["volatility"] < 0:
        raise ValueError("volatility must be non-negative")
```

```
    return config

def run_simulation(config):
    if config["n_simulations"] <= 0:
        raise ValueError("Invalid config")
    # Simulation logic would go here...
    config["results"] = {"mean": 105.2, "std": 12.3}
    return config

# Usage - must remember the correct sequence of function calls
config = load_simulation_config("sim_config.json")
config = validate_config(config)
config = run_simulation(config)
print(config["results"])
```

With OOP:

```
import json
from pathlib import Path
import pprint                                                    ①

class SimulationConfig:
    def __init__(
        self,
        name: str,
        n_simulations: int,
        initial_value: float,
        drift: float,
        volatility: float,
    ):
        self.name = name
        self.n_simulations = n_simulations
        self.initial_value = initial_value
        self.drift = drift
        self.volatility = volatility
        self.results: dict | None = None

        # Validation happens automatically on creation
        self._validate()

    def _validate(self) -> None:
        """Validate the configuration parameters."""
        if self.n_simulations <= 0:
            raise ValueError("n_simulations must be positive")
```

```
        if self.initial_value <= 0:
            raise ValueError("initial_value must be positive")
        if self.volatility < 0:
            raise ValueError("volatility must be non-negative")

    @classmethod                                                    ②
    def from_json(cls, filepath: str | Path) -> "SimulationConfig":
        """Create a SimulationConfig from a JSON file."""
        with open(filepath) as f:
            data = json.load(f)
        return cls(
            name=data["name"],
            n_simulations=data["n_simulations"],
            initial_value=data["initial_value"],
            drift=data["drift"],
            volatility=data["volatility"],
        )

    def run(self) -> "SimulationConfig":
        """Run the simulation."""
        # Simulation logic would go here...
        self.results = {"mean": 105.2, "std": 12.3}
        return self

    def __repr__(self) -> str:
        status = "completed" if self.results else "not run"
        return f"SimulationConfig({self.name!r}, {self.n_simulations} sims, {status})"
```

① The `pprint` (pretty print) module from Python's standard library formats complex data structures like dictionaries and lists in a more readable way, with proper indentation and line breaks. This is especially useful when displaying nested structures or long lists.

② The `@classmethod` decorator creates a *class method*—a method that receives the class itself (conventionally named `cls`) as its first argument instead of an instance. Class methods are often used as alternative constructors, like `from_json()` here. In contrast, a `@staticmethod` doesn't receive any implicit first argument and behaves like a regular function that happens to live inside a class.

Now we can use the class:

```
# Load from file using the class method
config = SimulationConfig.from_json("sim_config.json")

# Or create directly
config = SimulationConfig(
    name="Test Simulation",
    n_simulations=1000,
```

```
    initial_value=100.0,
    drift=0.05,
    volatility=0.2,
)

# Run and display results
config.run()
pprint.pprint(config.results)  # Pretty print the results dictionary
```

```
{'mean': 105.2, 'std': 12.3}
```

The OOP version is cleaner because the state is bundled together, and you don't need to pass around a configuration dictionary. The simulation object maintains its own state, making the code more organized and less error-prone.

Beyond reducing errors, the class also makes your code more clearly *defined*. With a dictionary, nothing prevents you from accessing a misspelled key like `config["n_simulaitons"]`—you'll only discover the typo at runtime. With a class, your editor (like VS Code) can immediately flag `config.n_simulaitons` as an error because it knows exactly which attributes `SimulationConfig` has. This kind of immediate feedback makes development faster and catches bugs before you even run the code.

3.2.2. Scenario 2: Multiple Related Variants

If you need to implement several variants of a similar concept, OOP with inheritance can reduce code duplication. For example, different return calculation methods:

```
import math

class Returns:
    """Base class for return calculations."""

    def __init__(self, prices: list[float]):
        self.prices = prices

    def calculate(self) -> list[float]:
        raise NotImplementedError("Subclasses must implement calculate()")

class SimpleReturns(Returns):
    """Calculate simple returns: (P_t / P_{t-1}) - 1"""

    def calculate(self) -> list[float]:
        return [
            (self.prices[i] / self.prices[i - 1]) - 1
            for i in range(1, len(self.prices))
```

```
        ]

class LogReturns(Returns):
    """Calculate log returns: log(P_t / P_{t-1})"""

    def calculate(self) -> list[float]:
        return [
            math.log(self.prices[i] / self.prices[i - 1])
            for i in range(1, len(self.prices))
        ]

class ExcessReturns(Returns):
    """Calculate excess returns over risk-free rate."""

    def __init__(self, prices: list[float], risk_free_rate: float):
        super().__init__(prices)
        self.risk_free_rate = risk_free_rate

    def calculate(self) -> list[float]:
        simple_returns = [
            (self.prices[i] / self.prices[i - 1]) - 1
            for i in range(1, len(self.prices))
        ]
        return [r - self.risk_free_rate for r in simple_returns]

# Usage
prices = [100.0, 102.0, 101.0, 105.0, 108.0]

simple = SimpleReturns(prices)
print("Simple returns:", [f"{r:.4f}" for r in simple.calculate()])

log_ret = LogReturns(prices)
print("Log returns:", [f"{r:.4f}" for r in log_ret.calculate()])

excess = ExcessReturns(prices, risk_free_rate=0.001)
print("Excess returns:", [f"{r:.4f}" for r in excess.calculate()])
```

```
Simple returns: ['0.0200', '-0.0098', '0.0396', '0.0286']
Log returns: ['0.0198', '-0.0099', '0.0388', '0.0282']
Excess returns: ['0.0190', '-0.0108', '0.0386', '0.0276']
```

This pattern is useful when you want to ensure different variants share a common interface or when you want to write code that works with any of the variants.

⚠ Don't Overuse Inheritance

Inheritance can create tight coupling between classes and make code harder to understand. Often, composition (having one class use another as an attribute) is a better choice. Only use inheritance when you have a genuine "is-a" relationship and need to substitute one type for another.
For cases where you want classes to share a common interface without inheritance, Python 3.8+ offers *Protocols* (from the `typing` module). A Protocol defines what methods and attributes a class should have, without requiring the class to explicitly inherit from anything. This is sometimes called "structural subtyping" or "duck typing with type hints."

3.2.3. Scenario 3: Encapsulating Complex Data Structures

When working with complex data structures that need validation or computed properties, classes provide a clean way to manage this complexity:

```
class EventStudyWindow:
    """Represents an event study window with validation."""

    def __init__(self, event_date, estimation_start, estimation_end,
                 event_start, event_end):
        self.event_date = event_date
        self.estimation_start = estimation_start
        self.estimation_end = estimation_end
        self.event_start = event_start
        self.event_end = event_end

        # Validate the window
        self._validate()

    def _validate(self):
        """Validate that the window makes sense."""
        if self.estimation_end >= self.event_date:
            raise ValueError("Estimation window must end before event date")

        if self.event_start > self.event_date:
            raise ValueError("Event window start must be at or before event date")

        if self.event_end < self.event_date:
            raise ValueError("Event window end must be at or after event date")

    @property
    def estimation_length(self):
        """Length of the estimation window in days."""
        return (self.estimation_end - self.estimation_start).days
```

```
    @property
    def event_length(self):
        """Length of the event window in days."""
        return (self.event_end - self.event_start).days

    def __repr__(self):
        return (f"EventStudyWindow(event={self.event_date}, "
                f"estimation={self.estimation_length} days, "
                f"event_window={self.event_length} days)")

# Usage
from datetime import date, timedelta

event_date = date(2024, 6, 15)
window = EventStudyWindow(
    event_date=event_date,
    estimation_start=event_date - timedelta(days=260),
    estimation_end=event_date - timedelta(days=10),
    event_start=event_date - timedelta(days=1),
    event_end=event_date + timedelta(days=1)
)

print(window)
print(f"Estimation period: {window.estimation_length} days")
print(f"Event window: {window.event_length} days")
```

```
EventStudyWindow(event=2024-06-15, estimation=250 days, event_window=2 days)
Estimation period: 250 days
Event window: 2 days
```

The class encapsulates both the data and the logic for validation and computation, making it easier to work with event study windows correctly.

3.2.4. Scenario 4: Building Reusable Components

If you're building functionality that will be reused across multiple projects, classes provide a clean interface:

```
import statistics
import random

class RollingWindow:
    """Calculate rolling window statistics."""
```

```
    def __init__(self, data: list[float], window_size: int):
        self.data = data
        self.window_size = window_size

        if len(self.data) < window_size:
            raise ValueError("Data must be longer than window size")

    def mean(self) -> list[float]:
        """Calculate rolling mean."""
        return [
            statistics.mean(self.data[i : i + self.window_size])
            for i in range(len(self) )
        ]

    def std(self) -> list[float]:
        """Calculate rolling standard deviation."""
        return [
            statistics.stdev(self.data[i : i + self.window_size])
            for i in range(len(self))
        ]

    def sharpe(self, risk_free_rate: float = 0) -> list[float]:
        """Calculate rolling Sharpe ratio."""
        means = self.mean()
        stds = self.std()
        return [(m - risk_free_rate) / s for m, s in zip(means, stds)]

    def __len__(self) -> int:
        return len(self.data) - self.window_size + 1

    def __repr__(self) -> str:
        return f"RollingWindow(data_length={len(self.data)}, window={self.window_size})"

# Generate some sample returns
random.seed(42)
returns = [random.gauss(0.001, 0.02) for _ in range(100)]
rolling = RollingWindow(returns, window_size=10)

print(f"Rolling windows: {len(rolling)}")
print(f"Mean rolling mean: {statistics.mean(rolling.mean()):.4f}")
print(f"Mean rolling Sharpe: {statistics.mean(rolling.sharpe()):.4f}")
```

```
Rolling windows: 91
Mean rolling mean: 0.0019
```

```
Mean rolling Sharpe: 0.1290
```

3.2.5. When to Avoid OOP

Just as important as knowing when to use OOP is knowing when not to use it. Avoid OOP when:

1. **You're doing one-off analysis:** If you're exploring data or doing a quick calculation, a simple script is fine.
2. **Your code is primarily a sequence of transformations:** Data pipelines that transform data step-by-step are often clearer as functions rather than classes.
3. **You're wrapping a single function:** Don't create a class with only one method. Just use a function.
4. **It makes the code more complex:** If OOP is making your code harder to understand, you're probably not in a situation where it helps.

Remember: the goal is clarity and maintainability, not using OOP for its own sake.

3.3. Data Classes

> **Video**
>
> The following video provides a good introduction to data classes.
>
> - This Is Why Python Data Classes Are Awesome

Python 3.7 introduced data classes, which provide a streamlined way to create classes that are primarily used to store data. They automatically generate common methods like `__init__`, `__repr__`, and `__eq__`, reducing boilerplate code significantly.

3.3.1. Basic Data Classes

Let's revisit our `Trade` class, but this time using a data class:

```python
from dataclasses import dataclass

@dataclass
class Trade:
    ticker: str
    quantity: int
    price: float
    side: str

    @property
```

```
    def value(self) -> float:
        """Calculate the total value of the trade."""
        return self.quantity * self.price

# Create trades
trade1 = Trade("AAPL", 100, 150.50, "buy")
trade2 = Trade("AAPL", 100, 150.50, "buy")
trade3 = Trade("MSFT", 50, 280.25, "sell")

print(trade1)
print(f"Value: ${trade1.value:.2f}")
print(f"trade1 == trade2: {trade1 == trade2}")
print(f"trade1 == trade3: {trade1 == trade3}")
```

```
Trade(ticker='AAPL', quantity=100, price=150.5, side='buy')
Value: $15050.00
trade1 == trade2: True
trade1 == trade3: False
```

With just the `@dataclass` decorator and type annotations, we get:

- An `__init__` method that accepts all the attributes
- A `__repr__` method that shows a useful string representation
- An `__eq__` method that compares instances by their attributes

This is much less code than writing these methods manually, and it's less error-prone.

3.3.2. Default Values

Data classes make it easy to specify default values:

```
from dataclasses import dataclass
from typing import Optional

@dataclass
class Trade:
    ticker: str
    quantity: int
    price: float
    side: str = "buy"  # default value
    commission: float = 0.0
    notes: Optional[str] = None

    def value(self):
```

```
        """Calculate the total value of the trade."""
        return self.quantity * self.price

    def net_value(self):
        """Calculate value after commission."""
        return self.value() - self.commission

# Use defaults
trade1 = Trade("AAPL", 100, 150.50)
print(trade1)

# Override defaults
trade2 = Trade("MSFT", 50, 280.25, side="sell", commission=14.00)
print(trade2)
print(f"Net value: ${trade2.net_value():.2f}")
```

```
Trade(ticker='AAPL', quantity=100, price=150.5, side='buy', commission=0.0, notes=None)
Trade(ticker='MSFT', quantity=50, price=280.25, side='sell', commission=14.0, notes=None)
Net value: $13998.50
```

3.3.3. Immutable Data Classes

You can make a data class immutable by setting `frozen=True`. This means that once created, the attributes cannot be changed:

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Trade:
    ticker: str
    quantity: int
    price: float
    side: str

    def value(self):
        return self.quantity * self.price

trade = Trade("AAPL", 100, 150.50, "buy")
print(trade)

# This would raise an error:
# trade.price = 160.00  # FrozenInstanceError
```

```
Trade(ticker='AAPL', quantity=100, price=150.5, side='buy')
```

Immutable data classes are useful when you want to ensure that data doesn't change unexpectedly, or when you need to use instances as dictionary keys or in sets.

3.3.4. Data Classes vs. Regular Classes

When should you use a data class instead of a regular class?

Use data classes when:

- Your class is primarily for storing data
- You want automatic generation of common methods
- You want type hints for all attributes
- You need value-based equality (comparing by content, not identity)

Use regular classes when:

- You need more control over initialization
- The class has complex behavior with little data
- You need inheritance from non-dataclass parents

3.3.5. Data Classes and Type Checking

Data classes work particularly well with static type checkers. The type annotations are not just documentation—they can be validated by tools like `ty` (a fast type checker from Astral, the creators of `uv` and `ruff`) or directly in VS Code.

```
from dataclasses import dataclass

@dataclass
class Trade:
    ticker: str
    quantity: int
    price: float
    side: str

# These work fine
trade1 = Trade("AAPL", 100, 150.50, "buy")
trade2 = Trade(ticker="MSFT", quantity=50, price=280.25, side="sell")

# Runtime Python won't stop these, but type checkers will flag them:
# trade3 = Trade(ticker="AAPL", quantity="100", price=150.50, side="buy")  # wrong type
# trade4 = Trade("AAPL", 100, 150.50)  # missing argument
```

The real advantage is that VS Code (with the Python or Pylance extension) can highlight these errors as you type, before you even save the file. This immediate feedback helps catch bugs early and makes development faster.

> **Pydantic for Data Validation**
>
> If you need runtime data validation (not just static type checking), consider Pydantic. It's a third-party library that offers functionality similar to dataclasses but validates data types at runtime, converts values to the correct types when possible, and provides detailed error messages when validation fails. Pydantic is particularly useful when working with external data sources like JSON files or API responses.

4. Code Quality and Documentation

Writing code is like writing prose for a dual audience: computers that execute it and humans who read, maintain, and extend it. While any code that runs correctly serves its immediate purpose, the real measure of quality lies in how easily others (including your future self) can understand, trust, and build upon your work.

In empirical research, where reproducibility and transparency are paramount, code quality takes on additional importance. A subtle bug in your data processing pipeline can invalidate months of work. Poorly documented functions can make it impossible for reviewers to verify your methodology. Code that works but cannot be understood becomes a liability rather than an asset.

This chapter covers the practical tools and techniques for writing clean, clear, readable, reproducible, and reliable code. We will explore how to organize your code for readability, document it effectively, catch errors before they cause problems, and maintain consistent style across your projects. These practices are not about perfectionism—they are about making your research more reliable, your collaboration more effective, and your future work easier.

4.1. Code Organization and Readability

The foundation of code quality is organization. Well-organized code reveals its structure and intent at a glance, making it easier to navigate, debug, and modify. This section covers the principles that make code readable and the practical techniques for achieving them.

4.1.1. The Principle of Least Surprise

Good code should behave the way readers expect. This means following established conventions, using descriptive names, and structuring your logic in clear, predictable ways. When you need to deviate from conventions, document why.

Consider two approaches to calculating portfolio returns:

```
# Unclear: cryptic names and unexpected structure
def calc(d, w):
    r = []
    for i in range(len(d)):
        r.append(sum([d[i][j] * w[j] for j in range(len(w))]))
    return r
```

```python
# Clear: descriptive names and explicit structure
def calculate_portfolio_returns(asset_returns, weights):
    """Calculate portfolio returns given asset returns and weights."""
    portfolio_returns = []
    for period_returns in asset_returns:
        period_portfolio_return = sum(
            asset_return * weight
            for asset_return, weight in zip(period_returns, weights)
        )
        portfolio_returns.append(period_portfolio_return)
    return portfolio_returns
```

The second version is longer, but its intent is immediately clear. The function name describes what it does, parameter names indicate what inputs are expected, and the logic is explicit rather than compressed.

4.1.2. Project Directory Structure

A well-organized directory structure makes projects easier to navigate and understand. For empirical research projects, we recommend the following layout:

```
my-project/
├── conf/                                                    ①
│   └── config.yaml
├── data/                                                    ②
│   ├── raw/                                                 ③
│   ├── clean/                                               ④
│   └── results/                                             ⑤
├── notebooks/                                               ⑥
├── notes/                                                   ⑦
├── paper/                                                   ⑧
├── slides/                                                  ⑨
├── src/my_project/                                          ⑩
│   ├── __init__.py
│   ├── pipeline.py                                          ⑪
│   └── utils/                                               ⑫
├── tests/                                                   ⑬
├── .gitignore                                               ⑭
├── pyproject.toml                                           ⑮
└── README.md                                                ⑯
```

① Configuration files for your analysis pipeline
② All data files, organized by processing stage
③ Original unprocessed data (never modify these files)
④ Processed and cleaned datasets

⑤ Analysis outputs like regression results
⑥ Jupyter notebooks for exploration and prototyping
⑦ Research notes and documentation
⑧ Paper manuscript (Quarto or LaTeX)
⑨ Presentation slides
⑩ Main Python package with your reusable code
⑪ Main analysis pipeline script
⑫ Utility functions and helpers
⑬ Unit tests for your code
⑭ Files to exclude from version control
⑮ Project dependencies and metadata
⑯ Project overview and setup instructions

This structure separates concerns clearly: raw data stays pristine, processed data is reproducible, and code is organized into reusable modules. The `src/` directory pattern keeps your package importable while maintaining a clean project root.

When working with version control, we usually want to keep data and results in the different location. We discuss this in Chapter 8.

4.1.3. Function Design

> **Video**
>
> The following video covers function design. Note that this is an external resource that may present concepts differently than those covered here.
>
> - The Ultimate Guide to Writing Functions

Functions should do one thing well. A function that does multiple unrelated tasks is harder to test, harder to reuse, and harder to understand. Consider this example:

```
# Poor: one function doing too much
def analyze_data(filepath):
    # Read data
    with open(filepath) as f:
        lines = f.readlines()

    # Parse and clean data
    values = []
    for line in lines:
        parts = line.strip().split(',')
        if len(parts) >= 2 and parts[1]:
            values.append(float(parts[1]))
```

```
    # Calculate statistics
    mean = sum(values) / len(values)
    squared_diffs = [(x - mean) ** 2 for x in values]
    std = (sum(squared_diffs) / len(values)) ** 0.5

    # Save results
    with open('stats.txt', 'w') as f:
        f.write(f'Mean: {mean}\nStd: {std}')

    return values

# Better: separate concerns into focused functions
def read_data_file(filepath):
    """Read lines from a data file."""
    with open(filepath) as f:
        return f.readlines()

def parse_values(lines, column=1):
    """Extract numeric values from CSV lines."""
    values = []
    for line in lines:
        parts = line.strip().split(',')
        if len(parts) > column and parts[column]:
            values.append(float(parts[column]))
    return values

def calculate_mean(values):
    """Calculate the arithmetic mean of a list of numbers."""
    if not values:
        raise ValueError("Cannot calculate mean of empty list")
    return sum(values) / len(values)

def calculate_std(values):
    """Calculate the standard deviation of a list of numbers."""
    if len(values) < 2:
        raise ValueError("Need at least 2 values for standard deviation")
    mean = calculate_mean(values)
    squared_diffs = [(x - mean) ** 2 for x in values]
    return (sum(squared_diffs) / len(values)) ** 0.5

def save_statistics(stats, output_path):
    """Save statistics dictionary to a text file."""
    with open(output_path, 'w') as f:
        for key, value in stats.items():
```

```
            f.write(f'{key}: {value}\n')
```

The refactored version is more verbose, but each function is now:

- **Testable**: You can verify each step independently
- **Reusable**: Functions can be used in other analyses
- **Readable**: Each function has a clear, single purpose
- **Maintainable**: Changes to one step don't affect others

4.1.4. Managing Complexity with Abstraction

As your analysis grows more sophisticated, you will build up layers of abstraction. Lower-level functions handle details; higher-level functions orchestrate workflow:

```
# Low-level: handle specific calculations
def calculate_mean(values):
    """Calculate arithmetic mean."""
    return sum(values) / len(values)

def calculate_variance(values):
    """Calculate population variance."""
    mean = calculate_mean(values)
    squared_diffs = [(x - mean) ** 2 for x in values]
    return sum(squared_diffs) / len(values)

def calculate_covariance(x_values, y_values):
    """Calculate covariance between two lists of values."""
    if len(x_values) != len(y_values):
        raise ValueError("Lists must have same length")
    x_mean = calculate_mean(x_values)
    y_mean = calculate_mean(y_values)
    products = [(x - x_mean) * (y - y_mean) for x, y in zip(x_values, y_values)]
    return sum(products) / len(x_values)

# Mid-level: combine calculations
def calculate_descriptive_stats(values):
    """Calculate common descriptive statistics."""
    mean = calculate_mean(values)
    variance = calculate_variance(values)
    std = variance ** 0.5
    return {'mean': mean, 'variance': variance, 'std': std}

# High-level: orchestrate entire analysis
def analyze_dataset(filepath, column=1):
```

```
    """
    Perform comprehensive analysis of a data file.

    Reads data, calculates descriptive statistics, and
    returns a complete summary.
    """
    lines = read_data_file(filepath)
    values = parse_values(lines, column)
    stats = calculate_descriptive_stats(values)
    stats['n'] = len(values)
    stats['min'] = min(values)
    stats['max'] = max(values)
    return stats
```

4.1.5. Code Layout and Readability

Python's readability comes partly from its use of whitespace. Use it deliberately:

```
# Cramped and hard to parse
def process_records(data,filters=None,transform=True):
    if filters is not None:data=[x for x in data if all(f(x) for f in filters)]
    if transform:data=[{'id':x['id'],'value':x['amount']*100} for x in data]
    return data

# Readable with proper spacing
def process_records(data, filters=None, transform=True):
    """Process records with optional filtering and transformation."""
    if filters is not None:
        data = [x for x in data if all(f(x) for f in filters)]

    if transform:
        data = [
            {'id': x['id'], 'value': x['amount'] * 100}
            for x in data
        ]

    return data
```

Guidelines for spacing:

- Use blank lines to separate logical sections
- Add spaces around operators (`=`, `+`, `==`, etc.)
- Avoid spaces immediately inside parentheses or brackets
- Group related items visually

We discuss how this can be automated in Section 4.4.

4.2. Docstrings and Documentation Standards

Video

The following video covers code documentation. Note that this is an external resource that may present concepts differently than those covered here.

- How to Document Your Code Like a Pro

Documentation bridges the gap between what your code does and what users (including future you) need to know to use it correctly. In Python, this documentation primarily takes the form of docstrings—string literals that appear as the first statement in a module, class, or function.

4.2.1. Why Docstrings Matter

Unlike comments, which explain how code works, docstrings explain what code does and how to use it. They serve multiple purposes:

1. **IDE integration**: Modern editors display docstrings as tooltips and in autocomplete
2. **Generated documentation**: Tools like Sphinx can extract docstrings to create HTML documentation
3. **Interactive help**: The `help()` function displays docstrings in the Python REPL
4. **Code review**: Reviewers can understand intent without reading implementation
5. **AI assistance**: AI coding assistants use docstrings to understand your code and provide better suggestions

Consider the difference:

```
# Without docstring
def clip_values(values, lower, upper):
    return [max(lower, min(upper, x)) for x in values]

# With docstring
def clip_values(values, lower, upper):
    """
    Limit values to fall within specified bounds.

    Parameters
    ----------
    values : list of float
        Data values to clip
    lower : float
        Lower bound (values below this are set to lower)
```

```
    upper : float
        Upper bound (values above this are set to upper)

    Returns
    -------
    list of float
        Clipped values with the same length as input

    Examples
    --------
    >>> clip_values([1, 5, 10, 15], lower=3, upper=12)
    [3, 5, 10, 12]

    Notes
    -----
    This function is useful for reducing the impact of outliers while
    retaining more information than simple outlier removal.
    """
    return [max(lower, min(upper, x)) for x in values]
```

When you call `help(clip_values)` or hover over the function in VS Code, you will see the formatted docstring with all the information needed to use the function correctly.

There are multiple standard docstring styles, but the most relevant for research projects are the NumPy and Google styles. Both provide clear structure for documenting parameters, return values, and examples. AI assistants can help generate well-formatted docstrings, but you should always review the descriptions to ensure they accurately reflect what your code does.

4.2.2. NumPy Documentation Style

The NumPy documentation style is the standard in the scientific Python community. It is more verbose than some alternatives, but its structured format makes it ideal for technical and scientific code.

The basic structure includes:

1. **One-line summary**: Brief description of what the function does
2. **Extended description** (optional): Additional context and details
3. **Parameters**: Each parameter with its type and description
4. **Returns**: What the function returns and its type
5. **Examples** (optional): Usage examples with expected output
6. **Notes** (optional): Additional information, warnings, or references

Here's a complete example:

```
def calculate_rolling_mean(values, window, min_periods=None):
    """
    Calculate rolling mean over a sliding window.

    Computes the arithmetic mean over a rolling window of specified size.
    The function handles edge cases at the beginning of the series.

    Parameters
    ----------
    values : list of float
        Sequence of numeric values
    window : int
        Number of values to include in each rolling window
    min_periods : int, optional
        Minimum number of observations required to calculate mean.
        If None, defaults to window size. Windows with fewer observations
        will return None.

    Returns
    -------
    list of float or None
        Rolling mean values. Returns None for positions where there
        are fewer than min_periods observations.

    Raises
    ------
    ValueError
        If window is less than 1
    ValueError
        If min_periods is greater than window

    Examples
    --------
    >>> values = [1.0, 2.0, 3.0, 4.0, 5.0]
    >>> calculate_rolling_mean(values, window=3)
    [None, None, 2.0, 3.0, 4.0]
    >>> calculate_rolling_mean(values, window=3, min_periods=1)
    [1.0, 1.5, 2.0, 3.0, 4.0]

    Notes
    -----
    The rolling mean at position i is calculated as the average of values
    from position (i - window + 1) to i, inclusive.
```

```
    See Also
    --------
    calculate_mean : Calculate mean of entire sequence
    calculate_rolling_std : Calculate rolling standard deviation
    """
    if window < 1:
        raise ValueError("window must be at least 1")

    if min_periods is None:
        min_periods = window

    if min_periods > window:
        raise ValueError("min_periods cannot exceed window")

    result = []
    for i in range(len(values)):
        start = max(0, i - window + 1)
        window_values = values[start:i + 1]
        if len(window_values) >= min_periods:
            result.append(sum(window_values) / len(window_values))
        else:
            result.append(None)

    return result
```

4.2.3. Google Documentation Style

Google style is an alternative that is more compact while still providing structure. It uses indented sections rather than underlined headers:

```
def normalize_values(values, method='zscore'):
    """Normalize a list of values using the specified method.

    Transforms values to have comparable scales, which is useful for
    combining variables measured in different units.

    Args:
        values (list of float): Numeric values to normalize
        method (str): Normalization method. Use 'zscore' for zero mean
            and unit variance, 'minmax' for scaling to [0, 1] range.

    Returns:
        list of float: Normalized values
```

```
    Raises:
        ValueError: If values has zero standard deviation (for zscore)
        ValueError: If all values are equal (for minmax)
        ValueError: If method is not recognized

    Example:
        >>> data = [10, 20, 30, 40, 50]
        >>> normalize_values(data, method='minmax')
        [0.0, 0.25, 0.5, 0.75, 1.0]
    """
    if method == 'zscore':
        mean = sum(values) / len(values)
        squared_diffs = [(x - mean) ** 2 for x in values]
        std = (sum(squared_diffs) / len(values)) ** 0.5
        if std == 0:
            raise ValueError("Cannot zscore normalize: zero standard deviation")
        return [(x - mean) / std for x in values]

    elif method == 'minmax':
        min_val, max_val = min(values), max(values)
        if min_val == max_val:
            raise ValueError("Cannot minmax normalize: all values are equal")
        return [(x - min_val) / (max_val - min_val) for x in values]

    else:
        raise ValueError(f"Unknown method: {method}")
```

For this course, we recommend using NumPy style for longer, more complex functions and Google style for simpler utilities. The key is to be consistent within a project. The docstring standards also define conventions for module-level and class-level docstrings, which follow similar patterns.

> **Documentation and Research Transparency**
>
> In empirical research, good documentation is not just helpful—it is essential for reproducibility. When writing up your analysis, you should be able to point reviewers to specific, well-documented functions that implement your methodology. This makes peer review more effective and helps establish trust in your results.

4.3. Type Hints and Static Typing

Python is a dynamically typed language, meaning you do not need to declare variable types. However, Python 3.5+ supports optional type hints that document expected types without changing runtime behavior. Type hints improve code quality by:

1. Making function interfaces explicit and self-documenting
2. Enabling static analysis tools to catch type errors before runtime
3. Improving IDE autocomplete and error detection
4. Serving as machine-checked documentation
5. Providing additional context to AI coding assistants

4.3.1. Basic Type Hints

Type hints specify the expected type of variables, parameters, and return values:

```
def calculate_return(initial_price: float, final_price: float) -> float:
    """Calculate simple return between two prices."""
    return (final_price - initial_price) / initial_price


def read_config(filepath: str) -> dict[str, str]:
    """Read configuration from a file."""
    config = {}
    with open(filepath) as f:
        for line in f:
            key, value = line.strip().split('=')
            config[key] = value
    return config
```

The syntax `parameter: type` indicates the expected type, and `-> type` indicates the return type.

4.3.2. Common Type Hints

Here are the type hints you will use most frequently:

```
def calculate_weighted_sum(
    values: list[float],
    weights: list[float]
) -> float:
    """Calculate weighted sum of values."""
    return sum(v * w for v, w in zip(values, weights))


def process_records(
    records: list[dict[str, str]],
    key: str = 'id'
) -> dict[str, dict[str, str]]:
    """Index records by a key field."""
```

```
    return {record[key]: record for record in records}


def calculate_statistics(
    values: list[float]
) -> tuple[float, float, float]:
    """Calculate mean, min, and max."""
    mean = sum(values) / len(values)
    return mean, min(values), max(values)
```

> **Legacy Type Hint Syntax**
>
> When type hints were first introduced in Python 3.5, you had to import special types from the `typing` module like `List`, `Dict`, `Tuple`, and `Union`. Starting with Python 3.9+, you can use the built-in types directly: `list[str]` instead of `List[str]`, `dict[str, int]` instead of `Dict[str, int]`, and `int | float` instead of `Union[int, float]`. You will still see the old style in many code examples and libraries, but for new code, prefer the modern syntax.

4.3.3. Optional and Union Types

Use `| None` when a parameter might be `None`:

```
def calculate_mean(values: list[float], default: float | None = None) -> float:
    """
    Calculate mean of values, with optional default for empty lists.

    Parameters
    ----------
    values : list of float
        Values to average
    default : float, optional
        Value to return if list is empty. If None, raises ValueError.
    """
    if not values:
        if default is None:
            raise ValueError("Cannot calculate mean of empty list")
        return default
    return sum(values) / len(values)
```

Use `|` (pipe) when a parameter can be one of several types:

```python
def format_value(value: int | float | str) -> str:
    """Format a value as a string with appropriate formatting."""
    if isinstance(value, float):
        return f"{value:.2f}"
    return str(value)
```

4.3.4. Type Checking with ty

Type hints become even more valuable when combined with static type checkers. We recommend `ty`, a fast type checker from Astral (the same company behind `uv` and `ruff`). Run it with:

```
uvx ty check your_script.py
```

The type checker will detect type inconsistencies:

```python
def calculate_return(initial_price: float, final_price: float) -> float:
    return (final_price - initial_price) / initial_price

# This will trigger a type error
result = calculate_return("100", "110")  # Error: expected float, got str
```

Type hints do not affect runtime behavior; Python will not enforce types unless you use a type checker. This means type hints are documentation and analysis tools, not runtime constraints. You can gradually add type hints to your codebase without breaking existing code.

For practical use, VS Code can highlight type errors the same way Word highlights typos, which is very useful to catch bugs early on.

For research code, focus type hints where they add the most value: public function interfaces that others will use, complex data transformations where types clarify expected structures, and critical calculations where you want to make assumptions explicit. You do not need to type hint every variable in every function—use judgment about where type information improves clarity.

4.4. Code Style and Linting with ruff

Consistent code style makes collaboration easier and reduces cognitive load when reading code. Rather than debating style choices, the Python community has converged on automated tools that enforce consistent formatting. This section introduces ruff, the modern standard for Python code quality. Ruff is made by Astral, the same company behind `uv` and `ty`.

4.4.1. Why Automated Formatting Matters

Manual formatting is time-consuming and leads to inconsistency. Different developers have different preferences for spacing, line breaks, and indentation. These differences create noise in version control, make code reviews harder, and waste mental energy on decisions that don't affect functionality.

Automated formatters solve this by making formatting decisions for you. While you might not agree with every choice, the consistency and time savings far outweigh any aesthetic preferences. Additionally, when you get used to a specific style, it increases readability—your eyes learn to scan consistently formatted code more quickly.

4.4.2. ruff: The All-in-One Linter

Ruff is an extremely fast Python linter and code formatter written in Rust. It replaces multiple tools (flake8, isort, pyupgrade, and more) with a single, consistent interface. Ruff can:

- Check for common errors and bugs
- Enforce code style guidelines
- Sort and organize imports
- Suggest modernizations and improvements
- Automatically fix many issues

The easiest way to use ruff is through the VS Code extension. Install the "Ruff" extension from the VS Code marketplace, and VS Code can be configured to automatically format and fix your code each time you save. This makes code quality effortless—just write your code and save.

You can also run ruff from the command line. Check your code with:

```
uvx ruff check your_script.py
```

Ruff will identify issues:

```
# example.py
import json
import os  # Unused import

def load_data(filepath):
    with open(filepath) as f:
        data = json.load(f)
    return data

# Unused variable
result = load_data("data.json")
```

Running `uvx ruff check example.py` produces:

```
example.py:2:8: F401 [*] `os` imported but unused
example.py:10:1: F841 [*] Local variable `result` is assigned to but never used
Found 2 errors.
[*] 2 potentially fixable with the `--fix` option.
```

Auto-fix issues:

```
uvx ruff check --fix example.py
```

Ruff will automatically remove the unused import and variable.

4.4.3. Configuring ruff

Configure ruff using a `pyproject.toml` file in your project root:

```
[tool.ruff]
# Set maximum line length (default is 88)
line-length = 88

# Target Python version
target-version = "py311"

[tool.ruff.lint]
# Enable specific rule sets
select = [
    "E",    # pycodestyle errors
    "F",    # Pyflakes
    "I",    # isort (import sorting)
    "B",    # flake8-bugbear (common bugs)
    "SIM",  # flake8-simplify
    "UP",   # pyupgrade (modernize syntax)
]

# Disable specific rules if needed
ignore = [
    "E501",  # Line too long (handled by formatter)
]

# Allow auto-fixing for these rule types
fixable = ["ALL"]

[tool.ruff.lint.per-file-ignores]
# Allow unused imports in __init__.py files
"__init__.py" = ["F401"]
```

```
# Relaxed rules for test files
"tests/**/*.py" = ["S101"]  # Allow assert statements
```

This configuration enables helpful checks while avoiding overly strict rules that might interfere with research workflows.

4.4.4. Common ruff Rules

Some particularly useful ruff rules:

Import Organization (I)

This rule organizes imports in alphabetical order and grouping them in three groups: standard library imports, third-party imports, and imports from the current project. It will also automatically remove imports that are not used in the code.

Before ruff:

```
import json
import pandas as pd
import sys
import os
from myproject.utils import helper
import csv
```

After ruff with isort rules:

```
import csv
import json
import os
import sys

import pandas as pd

from myproject.utils import helper
```

Bug Detection (B)

Ruff catches bugs such as the mutable default argument bug. When you use a mutable object like a list as a default argument, Python creates that object once when the function is defined—not each time the function is called. This means all calls share the same list, causing unexpected behavior where the list grows across calls.

Before (buggy):

```
def collect_values(value, results=[]):  # B006: Mutable default argument
    results.append(value)
    return results
```

After (fixed):

```
def collect_values(value, results=None):
    if results is None:
        results = []
    results.append(value)
    return results
```

Code Simplification (SIM)

Before:

```
if condition:
    return True
else:
    return False
```

After:

```
return condition
```

4.4.5. Formatting Code with ruff

In addition to linting, ruff includes a powerful code formatter. Ruff's formatter is opinionated—it makes formatting decisions for you, eliminating debates about style. The philosophy is simple: let the tool handle formatting so you can focus on the code itself.

Format your code:

```
uvx ruff format your_script.py
```

Before formatting:

```
def calculate_mean(values,skip_none=True):
    if skip_none:values=[v for v in values if v is not None]
    return sum(values)/len(values)
```

After formatting:

```
def calculate_mean(values, skip_none=True):
    if skip_none:
        values = [v for v in values if v is not None]
    return sum(values) / len(values)
```

You can use both linting and formatting together:

```
uvx ruff format your_script.py && uvx ruff check --fix your_script.py
```

This gives you a single, fast tool to automatically improve your code quality.

> Code Quality in Jupyter Notebooks
>
> Ruff can also format Jupyter notebooks:
>
> ```
> uvx ruff format analysis.ipynb
> uvx ruff check analysis.ipynb
> ```
>
> Configure per-cell ignores for exploratory code while maintaining standards for final analysis code.

4.5. Summary and Practical Guidelines

Code quality is not about perfectionism—it is about making your research more reliable, your collaboration more effective, and your future work easier. The practices covered in this chapter form the foundation of professional Python development:

- **Organization and readability**: Structure code to reveal intent clearly
- **Documentation**: Write docstrings that explain what code does and how to use it
- **Type hints**: Make data types explicit to catch errors early
- **Automated formatting**: Use ruff to maintain consistent style

For research projects, we recommend:

1. **Start simple**: Begin with basic ruff configuration, add rules gradually
2. **Format early**: Run `ruff format` regularly, not just before commits
3. **Fix what matters**: Use `ruff check --fix` to auto-fix safe issues
4. **Team consistency**: Ensure all collaborators use the same tools and configuration

These practices require some initial investment, but they pay dividends throughout your research career. Code that is well-organized, well-documented, and consistently formatted is easier to debug, easier to extend, and easier to share with collaborators and reviewers.

As you develop your empirical finance projects, make these practices habitual. Configure your tools once, integrate them into your workflow, and let automation handle the details. This frees you to focus on what

matters: designing sound research, implementing correct methodology, and drawing valid conclusions from your data.

The practices in this chapter work best alongside testing, which we will cover in the next chapter. While code quality ensures your code is readable and well-documented, testing ensures it is correct. Together, they form the foundation of reliable research software.

5. Error Handling and Testing

In empirical research, the quality and reliability of your code directly impacts the quality of your results. A small bug in your data processing pipeline or statistical calculation can invalidate months of work. Yet many researchers write code without systematic error handling or testing. The result? Papers retracted due to coding errors, results that can't be replicated, and countless hours spent debugging problems that could have been caught early.

This chapter introduces two complementary practices that will make your research code more robust: error handling and testing. Error handling is about writing code that fails gracefully and provides useful information when something goes wrong. Testing is about systematically verifying that your code does what you think it does. Together, these practices form the foundation of reliable, reproducible research.

Think of error handling as defensive driving for your code. You anticipate what might go wrong and plan for it. Testing, on the other hand, is like having a checklist before takeoff. You verify that everything works as expected before committing to your results. Both practices require a small upfront investment that pays enormous dividends in time saved and confidence gained.

5.1. Exceptions and Error Handling

When something goes wrong in a Python program, the interpreter raises an exception. An exception is Python's way of signaling that an error has occurred. If you've written any Python code, you've likely encountered exceptions: `TypeError`, `ValueError`, `KeyError`, `FileNotFoundError`, and so on. By default, an unhandled exception stops your program and prints a traceback showing where the error occurred.

While this default behavior is useful during development, it's often not what you want in production code or long-running research scripts. What if a file is missing, but you can use a default dataset instead? What if one stock in your analysis has corrupted data, but you want to continue processing the others? What if a network request fails, but you can retry it? This is where error handling comes in.

5.1.1. Understanding Exceptions

Let's start with a simple example. Suppose you're calculating portfolio returns and need to divide returns by portfolio values:

```
def calculate_return_percentage(return_dollars, portfolio_value):
    return (return_dollars / portfolio_value) * 100

# This works fine
print(calculate_return_percentage(1000, 50000))  # 2.0

# But this crashes
print(calculate_return_percentage(1000, 0))  # ZeroDivisionError!
```

When you try to divide by zero, Python raises a `ZeroDivisionError`. The program stops, and you see a traceback. While this is informative, it's not helpful if you're processing thousands of portfolios and one happens to have zero value.

5.1.2. The try-except Block

Python's `try-except` block allows you to handle exceptions gracefully. The basic structure is:

```
try:
    result = risky_operation()                    ①
except SomeException:                             ②
    result = default_value                        ③
```

① Code that might raise an exception goes in the `try` block.
② Specify which exception type(s) to catch.
③ Handle the error and provide a fallback.

Let's apply this to our portfolio example:

```
def calculate_return_percentage(return_dollars, portfolio_value):
    try:
        return (return_dollars / portfolio_value) * 100
    except ZeroDivisionError:
        # Portfolio has zero value, return None or a special value
        return None

# Now this doesn't crash
print(calculate_return_percentage(1000, 50000))  # 2.0
print(calculate_return_percentage(1000, 0))      # None
```

This is better, but we've lost information. We know the calculation failed, but we don't know which portfolio or why. In research code, you almost always want to preserve this information:

```
def calculate_return_percentage(return_dollars, portfolio_value):
    try:
        return (return_dollars / portfolio_value) * 100
    except ZeroDivisionError:
        print(f"Warning: Cannot calculate return for portfolio with zero value")
        return None
```

While `print()` statements are commonly used to log errors and warnings, there are better ways to handle logging in production code. We introduce proper logging techniques in Chapter 6.

> **When to Catch Exceptions**
>
> A common mistake is catching exceptions too broadly or too often. Don't catch exceptions just because you can. Catch them when you have a specific, sensible way to handle the error. If you can't do anything useful with the exception, it's often better to let it propagate and fail fast rather than hiding the problem.

5.1.3. Catching Multiple Exceptions

Often, several different things can go wrong, and you want to handle them differently:

```
def load_stock_data(filename):
    try:
        with open(filename, 'r') as f:
            lines = f.readlines()
        if len(lines) == 0:
            raise ValueError("File is empty")
        # Parse header and validate columns
        header = lines[0].strip().split(',')
        required_columns = ['date', 'close', 'volume']
        missing = set(required_columns) - set(header)
        if missing:
            raise ValueError(f"Missing required columns: {missing}")
        return lines
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found")
        return None
    except ValueError as e:
        print(f"Error: Invalid data format - {e}")
        return None
```

You can also catch multiple exceptions in a single `except` block if you want to handle them the same way:

```
def load_data(filename):
    try:
        with open(filename, 'r') as f:
            return f.read()
    except (FileNotFoundError, PermissionError) as e:
        print(f"Error loading '{filename}': {e}")
        return None
```

5.1.4. The else and finally Clauses

The `try-except` block can include two additional clauses: `else` and `finally`.

The `else` clause runs if no exception was raised:

```
def process_file(filename):
    try:
        with open(filename, 'r') as f:
            lines = f.readlines()
    except FileNotFoundError:
        print(f"File not found: {filename}")
        return None
    else:
        # This runs only if no exception occurred
        print(f"Successfully loaded {len(lines)} lines")
        return lines
```

The `finally` clause always runs, whether an exception occurred or not. This is useful for cleanup:

```
def analyze_large_dataset(filename):
    file_handle = None
    try:
        file_handle = open(filename, 'r')
        data = process(file_handle)
        return data
    except Exception as e:                                                    ①
        print(f"Error processing file: {e}")
        return None
    finally:                                                                  ②
        if file_handle:
            file_handle.close()
```

① The `except` block handles errors.
② The `finally` block always runs, ensuring the file is closed even if an error occurs or the function returns early.

Context Managers vs. finally

For file handling and similar resources, Python's context managers (the `with` statement) are usually cleaner than `finally`:

```
def analyze_large_dataset(filename):
    try:
        with open(filename, 'r') as file_handle:
            data = process(file_handle)
            return data
    except Exception as e:
        print(f"Error processing file: {e}")
        return None
```

The context manager automatically closes the file, even if an exception occurs.

5.1.5. Raising Exceptions

Sometimes you need to signal an error in your own code. Use the `raise` statement:

```
def calculate_sharpe_ratio(returns, risk_free_rate):
    """Calculate Sharpe ratio.

    Parameters
    ----------
    returns : array-like
        Series of returns
    risk_free_rate : float
        Risk-free rate

    Raises
    ------
    ValueError
        If returns is empty or risk_free_rate is negative
    """
    if len(returns) == 0:
        raise ValueError("Returns array cannot be empty")

    if risk_free_rate < 0:
        raise ValueError("Risk-free rate cannot be negative")

    excess_returns = returns - risk_free_rate
    return excess_returns.mean() / excess_returns.std()
```

This is much better than returning a special value like `-999` or `None` and hoping the caller checks for it. An exception forces the caller to explicitly handle the error.

5.1.6. Creating Custom Exceptions

> **Video**
>
> The following video covers similar topics to this section.
>
> - Why You Need Custom Exception Classes

For complex projects, you might want to define your own exception types. In Python, all exceptions are classes that inherit from the built-in `Exception` class (see Chapter 3 for more on classes and inheritance). This makes it easier to catch specific errors:

```
class DataQualityError(Exception):
    """Raised when data fails quality checks."""
    pass

class InsufficientDataError(Exception):
    """Raised when there's not enough data for analysis."""
    pass

def calculate_rolling_beta(stock_returns, market_returns, window=60):
    """Calculate rolling beta with data quality checks."""
    if len(stock_returns) < window:
        raise InsufficientDataError(
            f"Need at least {window} observations, got {len(stock_returns)}"
        )

    # Check for too many missing values
    missing_pct = stock_returns.isna().sum() / len(stock_returns)
    if missing_pct > 0.1:
        raise DataQualityError(
            f"Too many missing values: {missing_pct:.1%}"
        )

    # Calculate beta...
```

Now calling code can handle different errors appropriately:

```
try:
    beta = calculate_rolling_beta(stock_returns, market_returns)
except InsufficientDataError as e:
```

```
    print(f"Skipping stock: {e}")
    beta = None
except DataQualityError as e:
    print(f"Data quality issue: {e}")
    beta = None
```

> **Don't Swallow Exceptions**
>
> A common antipattern is the bare `except:` clause that catches everything:
>
> ```
> # BAD: This hides all errors, including bugs in your code
> try:
>     result = complex_calculation()
> except:
>     result = None
> ```
>
> This will catch not just the errors you expect, but also bugs in your code, keyboard interrupts, and system errors. Always catch specific exceptions, or at least use `except Exception:` which won't catch system-exiting exceptions.

5.1.7. Error Handling in Data Pipelines

In empirical research, you often process many items (stocks, firms, countries) where some might fail. Here's a pattern for handling this gracefully:

```
def process_stock(ticker, start_date, end_date):
    """Process a single stock, raising exceptions on failure."""
    # This function doesn't handle exceptions - it lets them propagate
    data = download_data(ticker, start_date, end_date)
    returns = calculate_returns(data)
    return calculate_statistics(returns)

def process_all_stocks(tickers, start_date, end_date):
    """Process multiple stocks, collecting both successes and failures."""
    results = {}
    errors = {}

    for ticker in tickers:
        try:
            results[ticker] = process_stock(ticker, start_date, end_date)
        except Exception as e:
            # Log the error but continue processing
            errors[ticker] = str(e)
```

```
            print(f"Error processing {ticker}: {e}")

    print(f"\nProcessed {len(results)} stocks successfully")
    print(f"Failed to process {len(errors)} stocks")

    return results, errors

# Usage
tickers = ['AAPL', 'MSFT', 'INVALID_TICKER', 'GOOGL']
results, errors = process_all_stocks(tickers, '2020-01-01', '2023-12-31')
```

This pattern separates the logic (in `process_stock`) from the error handling (in `process_all_stocks`). The individual function can be tested in isolation, while the batch function handles partial failures gracefully.

5.2. Unit Testing with pytest

> **Video**
>
> The following video covers similar topics to this section.
>
> - How to Write Great Unit Tests in Python

Error handling helps your code fail gracefully when things go wrong. Testing helps ensure things don't go wrong in the first place. A test is simply code that verifies other code works correctly. You write a test that calls your function with known inputs and checks that it produces the expected output.

5.2.1. Why Test?

You might think: "I'll just run my code and check the results. Why write separate tests?" Here's why testing matters:

1. **Confidence in changes**: When you modify code, tests verify you didn't break anything.
2. **Documentation**: Tests show how your code is meant to be used.
3. **Better design**: Code that's easy to test is usually better designed.
4. **Catch bugs early**: Tests find problems before they affect your results.
5. **Reproducibility**: Tests verify your code produces consistent results.
6. **Validation of AI-generated code**: Tests provide an additional layer of verification when using AI coding assistants.

In research, there's an additional benefit: tests help you understand your methods. Writing tests forces you to think clearly about what your code should do, edge cases, and assumptions. This deeper understanding often reveals problems in your research design.

AI Coding Assistants and Testing

AI coding assistants are particularly good at writing tests, making it easier to build a comprehensive test suite. However, always review AI-generated tests carefully. AI may miss important edge cases or make incorrect assumptions about expected behavior. Use AI-generated tests as a starting point, then add your own tests for edge cases and domain-specific scenarios that the AI might overlook.

5.2.2. Getting Started with pytest

pytest is Python's most popular testing framework. It's simple to use but powerful enough for complex projects. The easiest way to run pytest is using `uvx`, which runs the tool without requiring explicit installation:

```
uvx pytest test_math.py
```

If you're working on a project and want pytest available as a development dependency, add it to your project's dev group:

```
uv add --dev pytest
```

Then run it with:

```
uv run pytest
```

A pytest test is just a function whose name starts with `test_`. Here's the simplest possible test:

```
def test_simple():
    assert 1 + 1 == 2
```

Save this in a file called `test_math.py` and run:

```
uvx pytest test_math.py
```

You'll see output indicating the test passed. The `assert` statement is the heart of testing. If the expression after `assert` is `True`, the test passes. If it's `False`, the test fails.

5.2.3. Testing a Real Function

Let's test a function that calculates simple returns:

```python
# finance_utils.py
def calculate_simple_returns(prices):
    """Calculate simple returns from a price series.

    Parameters
    ----------
    prices : list
        Series of prices

    Returns
    -------
    list
        Simple returns (length is len(prices) - 1)
    """
    returns = []
    for i in range(1, len(prices)):
        ret = (prices[i] - prices[i-1]) / prices[i-1]
        returns.append(ret)
    return returns
```

Now write tests:

```python
# test_finance_utils.py
from finance_utils import calculate_simple_returns

def test_simple_returns_basic():
    """Test simple returns calculation with known values."""
    prices = [100, 110, 121]
    returns = calculate_simple_returns(prices)

    # Expected: (110-100)/100 = 0.10, (121-110)/110 = 0.10
    assert abs(returns[0] - 0.10) < 1e-10
    assert abs(returns[1] - 0.10) < 1e-10

def test_simple_returns_length():
    """Test that output length is correct."""
    prices = [100, 110, 121, 133.1]
    returns = calculate_simple_returns(prices)
    assert len(returns) == len(prices) - 1

def test_simple_returns_constant_prices():
    """Test with constant prices (zero returns)."""
    prices = [100, 100, 100]
    returns = calculate_simple_returns(prices)
    assert returns == [0, 0]
```

Run the tests:

```
uvx pytest test_finance_utils.py
```

Each test function checks a different aspect of the behavior. This is much more thorough than running the function once and eyeballing the output.

> **Test One Thing Per Test**
>
> Each test should verify one specific behavior. This makes failures easier to diagnose. When a test fails, you want to immediately know what's wrong, not spend time figuring out which of five assertions in the test failed.

5.2.4. Understanding Test Output

When a test fails, pytest provides detailed information:

```
def test_log_returns_incorrect():
    """This test will fail to demonstrate pytest output."""
    prices = [100, 110]
    returns = calculate_log_returns(prices)
    assert returns[0] == 0.1  # This is wrong - log(1.1) ≈ 0.0953
```

Running this produces:

```
test_finance_utils.py::test_log_returns_incorrect FAILED

================================== FAILURES ===================================
_________________________ test_log_returns_incorrect __________________________

    def test_log_returns_incorrect():
        prices = [100, 110]
        returns = calculate_log_returns(prices)
>       assert returns[0] == 0.1
E       assert 0.09531017980432493 == 0.1

test_finance_utils.py:8: AssertionError
```

The output shows exactly which assertion failed and what the actual value was. This makes debugging straightforward.

5.2.5. Testing with Fixtures

Often, you need the same data for multiple tests. pytest fixtures let you set up reusable test data:

```
import pytest

@pytest.fixture
def sample_prices():
    """Create sample price data for testing."""
    return [100, 105, 103, 108, 112, 110, 115]

def test_returns_are_correct(sample_prices):
    """Test returns calculation using fixture."""
    returns = calculate_simple_returns(sample_prices)
    # First return: (105-100)/100 = 0.05
    assert abs(returns[0] - 0.05) < 1e-10

def test_data_has_correct_length(sample_prices):
    """Test using the same fixture."""
    assert len(sample_prices) == 7
```

The `@pytest.fixture` decorator marks a function as a fixture. When you include the fixture name as a test function parameter, pytest automatically calls the fixture and passes its return value to your test.

Fixtures can also handle setup and teardown:

```
import pytest
import tempfile
import os

@pytest.fixture
def temp_data_file():
    """Create a temporary file with test data."""
    with tempfile.NamedTemporaryFile(mode='w', delete=False, suffix='.csv') as f:  ①
        f.write('date,close\n')
        f.write('2020-01-01,100\n')
        f.write('2020-01-02,110\n')
        temp_path = f.name

    yield temp_path  ②

    os.unlink(temp_path)  ③

def test_load_data_from_file(temp_data_file):  ④
    """Test loading data from a CSV file."""
```

```
    with open(temp_data_file) as f:
        lines = f.readlines()
    assert len(lines) == 3  # Header + 2 data rows
```

① **Setup**: Create temporary file with test data.
② **Yield**: Provide the file path to the test.
③ **Teardown**: Clean up the file after the test completes.
④ The fixture name as a parameter tells pytest to inject the fixture's return value.

The `yield` statement separates setup from teardown. Everything before `yield` runs before the test, and everything after runs after the test (even if the test fails).

5.2.6. Parametrized Tests

When you want to test the same function with multiple inputs, use parametrization:

```
import pytest


@pytest.mark.parametrize("prices,expected_length", [
    ([100, 110], 1),
    ([100, 110, 121], 2),
    ([100, 110, 121, 133], 3),
    ([100], 0),  # Edge case: single price
])
def test_returns_length_parametrized(prices, expected_length):
    """Test that returns have correct length for various inputs."""
    returns = calculate_simple_returns(prices)
    assert len(returns) == expected_length
```

This creates four separate tests, one for each parameter set. This is cleaner than writing four separate test functions and makes the pattern clear.

You can parametrize multiple arguments:

```
@pytest.mark.parametrize("initial_price,final_price,expected_return", [
    (100, 110, 0.10),   # 10% price increase
    (100, 90, -0.10),   # 10% price decrease
    (100, 100, 0),      # No change
    (50, 100, 1.0),     # 100% increase
])
def test_simple_return_calculation(initial_price, final_price, expected_return):
    """Test simple return calculation with various price changes."""
    returns = calculate_simple_returns([initial_price, final_price])
    assert abs(returns[0] - expected_return) < 1e-10
```

5.2.7. Testing for Exceptions

Sometimes you want to verify that your code raises an exception in certain situations:

```python
def calculate_mean_return(returns):
    """Calculate mean return."""
    if len(returns) == 0:
        raise ValueError("Returns list cannot be empty")

    return sum(returns) / len(returns)

def test_mean_return_empty_raises():
    """Test that empty returns raise ValueError."""
    with pytest.raises(ValueError, match="cannot be empty"):
        calculate_mean_return([])

def test_mean_return_with_data():
    """Test normal mean return calculation."""
    returns = [0.01, 0.02, -0.01, 0.03]
    mean = calculate_mean_return(returns)
    assert abs(mean - 0.0125) < 1e-10
```

The `pytest.raises()` context manager asserts that the code block raises the specified exception. The `match` parameter checks that the exception message matches a pattern (using regular expressions).

> **Testing with NumPy and pandas**
>
> Testing functions that work with NumPy arrays or pandas DataFrames sometimes requires special handling, such as using `np.allclose()` for floating-point array comparisons or `pd.testing.assert_frame_equal()` for DataFrame comparisons.

5.2.8. Organizing Tests

As your project grows, organize tests to mirror your code structure:

```
my_research_project/
├── finance_utils/
│   ├── __init__.py
│   ├── returns.py
│   ├── risk.py
│   └── portfolio.py
└── tests/
    ├── __init__.py
```

```
    ├── test_returns.py
    ├── test_risk.py
    └── test_portfolio.py
```

Run all tests with:

```
uvx pytest tests/
```

Or if pytest is installed in your project:

```
uv run pytest tests/
```

Run specific tests:

```
uvx pytest tests/test_returns.py
uvx pytest tests/test_returns.py::test_simple_returns_basic
```

Configuration with pytest.ini

Create a `pytest.ini` file in your project root to configure pytest:

```
[tool.pytest.ini_options]
testpaths = tests
python_files = test_*.py
python_functions = test_*
addopts = -v --strict-markers
```

This specifies where to find tests and how to run them.

5.3. Test-Driven Development Concepts

Video

The following video by ArjanCodes covers similar topics to this section.

- Test-Driven Development In Python // The Power of Red-Green-Refactor

Test-Driven Development (TDD) is a development approach where you write tests before writing the code they test. This might seem backwards, but it has significant benefits, especially in research.

5.3.1. The TDD Cycle

TDD follows a simple cycle:

1. **Red**: Write tests that fail (because the code doesn't exist yet), including edge cases
2. **Green**: Write just enough code to make all the tests pass
3. **Refactor**: Improve the code while keeping tests passing

Let's walk through an example. Suppose you need to calculate the maximum drawdown of a price series.

Step 1: Write failing tests, including edge cases

```
# test_risk.py
from risk import calculate_max_drawdown

def test_max_drawdown_simple():
    """Test max drawdown with simple price series."""
    prices = [100, 110, 105, 115, 90, 95]
    # Peak is 115, trough is 90, drawdown is (90-115)/115 ≈ -0.217
    assert abs(calculate_max_drawdown(prices) - (-0.217)) < 0.001

def test_max_drawdown_no_drawdown():
    """Test with monotonically increasing prices (no drawdown)."""
    prices = [100, 110, 120, 130]
    assert calculate_max_drawdown(prices) == 0

def test_max_drawdown_single_price():
    """Test with single price."""
    prices = [100]
    assert calculate_max_drawdown(prices) == 0
```

Run these tests. They will all fail because `calculate_max_drawdown` doesn't exist yet.

Step 2: Write minimal code to pass

```
# risk.py
def calculate_max_drawdown(prices):
    """Calculate maximum drawdown from a price series."""
    if len(prices) <= 1:
        return 0

    max_drawdown = 0
    peak = prices[0]

    for price in prices:
        if price > peak:
```

```
            peak = price
        drawdown = (price - peak) / peak
        if drawdown < max_drawdown:
            max_drawdown = drawdown

    return max_drawdown
```

Run the tests again. They should all pass.

Step 3: Refactor if needed

The code is clean and handles all edge cases. We can now move on, or improve our function to make it more efficient.

5.3.2. Benefits of TDD for Research

TDD might feel slow at first, but it pays off:

1. **Clarifies thinking**: Writing the test first forces you to specify exactly what you want the function to do.
2. **Prevents scope creep**: You implement only what's needed to pass tests.
3. **Documents intent**: Tests show how the function should behave.
4. **Enables refactoring**: You can improve code with confidence because tests verify behavior doesn't change.

In research, TDD is particularly valuable when implementing statistical methods or financial calculations. Write tests based on the formulas in the paper, then implement the method. The tests verify you've implemented the method correctly.

> TDD for Complex Calculations
>
> When implementing a complex statistical method from a paper:
>
> 1. Create tests using examples from the paper (if provided)
> 2. Create tests using results from R or Stata implementations
> 3. Create tests using simple cases you can verify by hand
> 4. Then implement your Python version
>
> This approach catches mistakes early and gives you confidence in your implementation.

5.3.3. When Not to Use TDD

TDD isn't always the right approach:

- **Exploratory analysis**: When you don't know what you're looking for, write code first, then add tests

- **Prototypes**: If you're just trying something to see if it works, TDD adds overhead
- **Simple scripts**: For one-off analyses, informal testing might be enough

But for any code you'll reuse or that's critical to your results, testing (whether test-first or test-after) is essential.

5.4. Testing Floating-Point Calculations

Financial calculations often involve floating-point arithmetic, which has quirks:

```
def test_floating_point_comparison():
    """Demonstrate floating-point comparison issues."""
    # This might fail due to floating-point precision
    result = 0.1 + 0.2
    # Don't do this:
    # assert result == 0.3  # Might fail!

    # Do this instead:
    assert abs(result - 0.3) < 1e-10
    # Or use pytest's approximate comparison:
    assert result == pytest.approx(0.3)
```

Always use tolerance-based comparisons for floating-point numbers. The `pytest.approx()` function is particularly nice because it chooses sensible default tolerances:

```
def test_returns_calculation():
    """Test returns calculation with approximate comparison."""
    prices = [100, 105, 110.25]
    returns = calculate_simple_returns(prices)
    expected = [0.05, 0.05]

    assert returns == pytest.approx(expected, rel=1e-6)
```

5.5. Best Practices Summary

Let's consolidate what we've learned into actionable practices:

Error Handling:

- Catch specific exceptions, not broad ones
- Provide informative error messages
- Don't hide errors unless you can handle them meaningfully
- Use custom exceptions for domain-specific errors

- Validate inputs early and explicitly

Testing:

- Write tests for any code you'll reuse
- Test edge cases, not just happy paths
- One assertion per test when possible
- Use fixtures for reusable test data
- Use parametrization to test multiple scenarios
- Run tests frequently during development

General:

- Make functions testable (pure functions with clear inputs/outputs)
- Validate assumptions with assertions
- Document expected behavior
- Use type hints to catch errors early
- Review your own code before considering it done

> **Testing in Research Workflows**
>
> In empirical research, you often have a mix of:
>
> - **Library code**: Functions you'll reuse across projects (test thoroughly)
> - **Analysis scripts**: One-off analyses (test key calculations)
> - **Exploratory code**: Trying things out (informal testing is fine)
>
> Focus your testing effort on library code and anything that affects your paper's results. A bug in a chart's formatting is annoying; a bug in your returns calculation invalidates your research.

5.6. Conclusion

Error handling and testing might feel like overhead when you start a project, but they're investments that pay enormous dividends. Code that handles errors gracefully is more robust and maintainable. Code with tests is easier to modify, debug, and trust.

In empirical research, where your code directly impacts your results and conclusions, this isn't just about software engineering best practices—it's about research integrity. A well-tested analysis pipeline gives you confidence in your results. Good error handling helps you identify data quality issues and edge cases. Together, they make your research more reproducible and reliable.

Start small. Add error handling to functions that interact with external data. Write tests for your key calculations. As these practices become habits, you'll find yourself writing better code, spending less time debugging, and having more confidence in your results.

6. Logging and Configuration

When you run a research pipeline—downloading data, cleaning it, estimating models, and generating output—things will go wrong. Files will be missing, APIs will fail, and edge cases will surface. Without proper logging, you're left guessing what happened and when. This chapter covers Python's built-in logging module and introduces Hydra, a powerful framework for managing complex research configurations.

Good logging practices are essential for reproducible research. When you run an analysis months later or share code with collaborators, logs provide a record of what the code did, what warnings occurred, and where things failed. Combined with proper configuration management, you can recreate any run exactly as it happened.

6.1. Why Logging Matters

Many researchers start by sprinkling `print()` statements throughout their code:

```
print("Starting data download...")
print(f"Downloaded {len(df)} rows")
print("WARNING: Missing values detected")
print("Error: API rate limit exceeded")
```

This approach has several problems:

1. **No severity levels**: You can't distinguish informational messages from warnings or errors
2. **No timestamps**: You don't know when events occurred
3. **No control**: You can't easily turn messages on or off or redirect them to files
4. **No context**: You don't know which module or function produced the message
5. **Cluttered output**: Everything goes to the same place, making it hard to find important messages

A proper logging system offers:

- **Severity levels**: DEBUG, INFO, WARNING, ERROR, CRITICAL—so you can filter by importance
- **Timestamps**: Know exactly when each event occurred
- **Source information**: See which module and function generated each message
- **Flexible output**: Send logs to console, files, or external services
- **Configuration**: Control logging behavior without changing code

6.2. Python's logging Module

Python's standard library includes a powerful logging module. While it has a learning curve, understanding its core concepts pays off in any serious project.

> **Video**
>
> The following video provides a good overview of Python logging.
>
> - Python Logging: How to Write Logs Like a Pro!

6.2.1. Basic Usage

The simplest way to use logging:

```
import logging

# Configure basic logging
logging.basicConfig(level=logging.INFO)

# Create a logger for this module
logger = logging.getLogger(__name__)

# Log messages at different levels
logger.debug("Detailed information for debugging")
logger.info("General information about program execution")
logger.warning("Something unexpected happened, but program continues")
logger.error("A serious problem occurred")
logger.critical("Program may not be able to continue")
```

```
INFO:__main__:General information about program execution
WARNING:__main__:Something unexpected happened, but program continues
ERROR:__main__:A serious problem occurred
CRITICAL:__main__:Program may not be able to continue
```

6.2.2. Understanding Log Levels

Log levels form a hierarchy. When you set a level, you see messages at that level and above:

Level	Numeric Value	When to Use
DEBUG	10	Detailed diagnostic information
INFO	20	Confirmation that things work as expected

Level	Numeric Value	When to Use
WARNING	30	Something unexpected but not necessarily wrong
ERROR	40	A serious problem; some functionality failed
CRITICAL	50	A very serious error; program may crash

```
import logging

# Only show WARNING and above
logging.basicConfig(level=logging.WARNING, force=True)                    ①
logger = logging.getLogger("level_demo")

logger.debug("This won't appear")
logger.info("This won't appear either")
logger.warning("This will appear")
logger.error("This will definitely appear")
```

① The `force=True` parameter is needed here because we already called `basicConfig()` earlier in this chapter. By default, `basicConfig()` only configures logging once—subsequent calls are ignored. Using `force=True` removes any existing handlers and reconfigures logging with the new settings.

```
WARNING:level_demo:This will appear
ERROR:level_demo:This will definitely appear
```

Using log levels provides two key advantages:

1. **Route messages to different outputs**: You can direct messages of different levels to different destinations—for example, send INFO messages to a file while only showing WARNING and above on the console.
2. **Control verbosity at runtime**: You can leave all log messages in your code but choose at runtime which levels to display. This means you can include detailed DEBUG messages during development that won't clutter your output in production unless you need them.

6.2.3. Configuring Log Format

The default format is minimal. For research workflows, you typically want more information:

```
import logging

# Configure with a custom format
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
```

```
    datefmt='%Y-%m-%d %H:%M:%S',
    force=True
)

logger = logging.getLogger("format_demo")
logger.info("Now you can see when this happened")
```

```
2026-01-03 10:48:29 - format_demo - INFO - Now you can see when this happened
```

Common format fields:

- `%(asctime)s`: Human-readable timestamp
- `%(name)s`: Logger name (usually module name)
- `%(levelname)s`: DEBUG, INFO, WARNING, etc.
- `%(message)s`: The actual log message
- `%(filename)s`: Source file name
- `%(lineno)d`: Line number in source file
- `%(funcName)s`: Function name

6.2.4. Logging to Files

For long-running research pipelines, you want logs saved to files:

```
import logging

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('research_pipeline.log'),
        logging.StreamHandler()  # Also print to console
    ]
)

logger = logging.getLogger(__name__)
logger.info("This goes to both the file and console")
```

6.2.5. Logging Exceptions

When catching exceptions, use `logger.exception()` to automatically include the traceback:

```
import logging

logging.basicConfig(level=logging.INFO, force=True)
logger = logging.getLogger("exception_demo")

def risky_calculation(x):
    return 1 / x

try:
    result = risky_calculation(0)
except ZeroDivisionError:
    logger.exception("Calculation failed")
    # The traceback is automatically included
```

```
ERROR:exception_demo:Calculation failed
Traceback (most recent call last):
  File "/var/folders/jr/cn9h86ld68qb5rtvs9gsb1vr0000gn/T/ipykernel_77417/1524267329.py", line 10, in <module>
    result = risky_calculation(0)
  File "/var/folders/jr/cn9h86ld68qb5rtvs9gsb1vr0000gn/T/ipykernel_77417/1524267329.py", line 7, in risky_calculation
    return 1 / x
           ~~^~~
ZeroDivisionError: division by zero
```

Including the full traceback is a tradeoff: it provides valuable debugging information, but the multi-line output can break the structure of log files, making them harder to parse or query programmatically. For production systems where logs are processed automatically, you might prefer logging just the exception message and using `logger.error()` instead.

6.2.6. Module-Level Loggers

The recommended pattern is to create a logger at the top of each module:

```
# In portfolio_analysis.py
import logging

logger = logging.getLogger(__name__)

def calculate_returns(prices):
    logger.info(f"Calculating returns for {len(prices)} observations")
    returns = prices.pct_change().dropna()

    if returns.isna().any().any():
        logger.warning("NaN values detected in returns")
```

```
    logger.debug(f"Returns shape: {returns.shape}")
    return returns
```

The `__name__` variable becomes the module's fully qualified name (e.g., `myproject.portfolio_analysis`), which helps you trace where messages came from.

6.3. Logging Best Practices

Here are key practices to follow when implementing logging in your research projects.

Use appropriate levels. Choose log levels thoughtfully:

```
# DEBUG: Detailed diagnostic info, usually only for debugging
logger.debug(f"Processing row {i}: values = {row}")

# INFO: Key milestones and confirmations
logger.info(f"Loaded {len(df)} rows from {filename}")

# WARNING: Unexpected but handled situations
logger.warning(f"Missing data for {ticker}, using interpolation")

# ERROR: Something failed, but program can continue
logger.error(f"Failed to download {ticker}: {e}")

# CRITICAL: Serious failure, program may need to stop
logger.critical("Database connection lost, cannot continue")
```

Include context in messages. Log messages should be self-explanatory:

```
# Bad: Not enough context
logger.info("Processing file")
logger.warning("Missing values")

# Good: Clear context
logger.info(f"Processing file: {filepath}")
logger.warning(f"Missing values in column '{col}': {count} rows affected")
```

Don't log sensitive information. Be careful not to log passwords, API keys, or sensitive data:

```
# Bad: Logs the API key
logger.info(f"Connecting with API key: {api_key}")

# Good: Masks sensitive information
logger.info(f"Connecting with API key: {api_key[:4]}...")
```

Use structured logging for complex data. For data that might be parsed later, consider structured formats:

```
import json
import logging

logger = logging.getLogger(__name__)

# Log structured data
metrics = {
    'ticker': 'AAPL',
    'sharpe_ratio': 1.45,
    'max_drawdown': -0.15,
    'n_observations': 252
}
logger.info(f"Performance metrics: {json.dumps(metrics)}")
```

Configure logging once at entry point. Configure logging at your application's entry point, not in library modules. Include a timestamp in the log filename so that each run generates a new file:

```
# In main.py or run_analysis.py
import logging
from datetime import datetime
from my_research import run_pipeline

def setup_logging():
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    logging.basicConfig(
        level=logging.INFO,
        format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
        handlers=[
            logging.FileHandler(f'analysis_{timestamp}.log'),
            logging.StreamHandler()
        ]
    )

if __name__ == "__main__":
    setup_logging()
    run_pipeline()
```

Library modules should only create loggers, not configure them:

```
# In my_research/analysis.py
import logging

logger = logging.getLogger(__name__)  # Just create the logger
```

```
def run_pipeline():
    logger.info("Starting pipeline")
    # ...
```

6.4. Configuration Management with Hydra

As research projects grow, managing configuration becomes a challenge. You might have:

- Different data sources (local files, APIs, databases)
- Multiple model specifications to compare
- Various output formats and destinations
- Development vs. production settings

Hardcoding these in Python leads to messy code and makes it hard to reproduce specific runs. YAML configuration files help, but you end up writing boilerplate code to load and validate them.

Hydra is a framework developed by Facebook Research (Yadan 2019) that elegantly solves these problems. It provides:

- **Hierarchical configuration**: Compose configs from multiple sources
- **Command-line overrides**: Change any parameter without editing files
- **Automatic working directories**: Each run gets its own output directory
- **Multi-run support**: Sweep over parameter combinations

> Video
>
> The following video provides a good overview of Hydra for managing project configurations.
>
> - NEVER Worry About Data Science Projects Configs Again

6.4.1. Installing Hydra

```
uv add hydra-core
```

6.4.2. Basic Hydra Application

Here's a minimal Hydra application:

```
# my_analysis.py
import hydra
from omegaconf import DictConfig

@hydra.main(version_base=None, config_path="conf", config_name="config")
def main(cfg: DictConfig) -> None:
    print(f"Processing data from: {cfg.data.source}")
    print(f"Output directory: {cfg.output.dir}")
    print(f"Model: {cfg.model.name}")

if __name__ == "__main__":
    main()
```

With a configuration file:

```
# conf/config.yaml
data:
  source: "data/returns.parquet"
  start_date: "2020-01-01"
  end_date: "2023-12-31"

model:
  name: "ols"
  robust_se: true

output:
  dir: "results"
  format: "parquet"
```

Run it:

```
python my_analysis.py
```

Override parameters from command line:

```
python my_analysis.py data.start_date=2022-01-01 model.name=fama_macbeth
```

6.4.3. Configuration Composition

Hydra's power comes from composing configurations. Organize your configs into groups:

```
conf/
├── config.yaml          # Main config with defaults
├── data/
│   ├── crsp.yaml       # CRSP data settings
│   ├── compustat.yaml  # Compustat settings
│   └── local.yaml      # Local file settings
├── model/
│   ├── ols.yaml
│   ├── fama_macbeth.yaml
│   └── panel.yaml
└── output/
    ├── paper.yaml      # Publication-ready output
    └── debug.yaml      # Quick debug output
```

The main config selects defaults:

```yaml
# conf/config.yaml
defaults:
  - data: crsp
  - model: ols
  - output: paper

experiment_name: "baseline"
```

Each group config defines its settings:

```yaml
# conf/data/crsp.yaml
source: "wrds"
database: "crsp"
table: "msf"
start_date: "1990-01-01"
end_date: "2023-12-31"
```

```yaml
# conf/model/fama_macbeth.yaml
name: "fama_macbeth"
robust_se: true
lags: 5
```

Switch configurations easily:

```
# Use Compustat data with Fama-MacBeth model
python my_analysis.py data=compustat model=fama_macbeth

# Quick debug run
python my_analysis.py output=debug data.end_date=2020-01-31
```

6.4.4. Automatic Output Directories

Hydra automatically creates a unique output directory for each run:

```
outputs/
└── 2024-01-15/
    └── 14-30-22/
        ├── .hydra/
        │   ├── config.yaml      # Full resolved config
        │   ├── hydra.yaml       # Hydra settings
        │   └── overrides.yaml   # Command-line overrides
        ├── my_analysis.log      # Automatic logging
        └── results/             # Your output files
```

This makes every run reproducible—you can see exactly what configuration was used.

6.4.5. Using Hydra for Research Pipelines

Here's a more complete example for an empirical finance pipeline:

```python
# run_analysis.py
import logging
from pathlib import Path

import hydra
from omegaconf import DictConfig, OmegaConf
import pandas as pd

logger = logging.getLogger(__name__)


def load_data(cfg: DictConfig) -> pd.DataFrame:
    """Load data according to configuration."""
    logger.info(f"Loading data from {cfg.data.source}")

    if cfg.data.source == "local":
        df = pd.read_parquet(cfg.data.path)
    elif cfg.data.source == "wrds":
        # WRDS loading logic
        pass

    # Apply date filters
    df = df[(df['date'] >= cfg.data.start_date) &
            (df['date'] <= cfg.data.end_date)]
```

```
    logger.info(f"Loaded {len(df)} observations")
    return df


def run_model(df: pd.DataFrame, cfg: DictConfig) -> dict:
    """Run the specified model."""
    logger.info(f"Running {cfg.model.name} model")

    # Model logic here
    results = {"coefficients": {}, "stats": {}}

    return results


def save_results(results: dict, cfg: DictConfig) -> None:
    """Save results according to configuration."""
    output_dir = Path(cfg.output.dir)
    output_dir.mkdir(parents=True, exist_ok=True)

    # Save based on format
    if cfg.output.format == "parquet":
        # Save as parquet
        pass
    elif cfg.output.format == "latex":
        # Generate LaTeX tables
        pass

    logger.info(f"Results saved to {output_dir}")


@hydra.main(version_base=None, config_path="conf", config_name="config")
def main(cfg: DictConfig) -> None:
    # Log the full configuration
    logger.info("Configuration:\n" + OmegaConf.to_yaml(cfg))

    # Run pipeline
    df = load_data(cfg)
    results = run_model(df, cfg)
    save_results(results, cfg)

    logger.info("Pipeline completed successfully")


if __name__ == "__main__":
```

```
    main()
```

6.4.6. Multi-Run for Parameter Sweeps

Hydra can automatically run your code with multiple parameter combinations:

```
# Run with multiple date ranges
python run_analysis.py -m data.start_date=2010-01-01,2015-01-01,2020-01-01

# Sweep over models
python run_analysis.py -m model=ols,fama_macbeth,panel
```

Each combination gets its own output directory with full configuration tracking.

This feature is particularly useful for testing the sensitivity of your analysis to empirical choices. For example, you can run your analysis with multiple winsorization levels, different sample periods, or alternative variable definitions to ensure your results are robust to these choices.

6.4.7. Hydra with Logging

Hydra automatically configures Python's logging module. Your log messages go to both the console and a file in the output directory:

```
import logging
import hydra
from omegaconf import DictConfig

logger = logging.getLogger(__name__)

@hydra.main(version_base=None, config_path="conf", config_name="config")
def main(cfg: DictConfig) -> None:
    logger.info("Starting analysis")  # Automatically logged to file
    logger.debug("Debug info")  # Also captured

    # Your code here
```

You can customize logging in `conf/hydra/job_logging.yaml` or in your main config.

6.5. Summary

Proper logging and configuration management are essential for reproducible research:

- **Use Python's logging module** instead of print statements for production code
- **Choose appropriate log levels** to distinguish routine information from warnings and errors
- **Include context in log messages** so they're meaningful when read later
- **Configure logging at the entry point**, not in library modules
- **Use Hydra for configuration management** in complex research pipelines
- **Leverage Hydra's automatic output directories** to make every run reproducible

These practices might seem like overhead for small scripts, but they pay dividends as projects grow. When you need to debug a failed run from last month or share code with collaborators, good logging and configuration management make the difference between hours of frustration and quickly finding the answer.

7. Development Environment and Tools

This chapter covers setting up Visual Studio Code (VS Code) for Python data science work. A well-configured development environment makes coding more pleasant and efficient. We'll cover fonts, themes, extensions for formatting and linting, data exploration tools, and productivity enhancements.

> Videos
>
> This chapter follows my VS Code configuration video for data science.
>
> - My VS Code Config for Data Science

7.1. Fonts for Coding

Before opening VS Code, choose a good programming font. The right font makes code more readable and reduces eye strain during long sessions. Programming fonts have specific characteristics:

- **Monospace**: Every character has the same width, essential for code alignment
- **Distinguishable characters**: Clear differences between similar characters like `1`, `l`, `I` and `0`, `O`
- **Ligatures** (optional): Special combined glyphs for common character sequences

Ligatures are combined symbols that appear when certain characters follow each other. For example, `->` becomes a proper arrow →, `!=` becomes a not-equal symbol ≠, and `>=` becomes a greater-than-or-equal symbol ≥. The underlying characters don't change—your code still contains `->`. Ligatures simply render more elegantly on screen. Whether to use ligatures is a personal choice; some developers love them, others find them distracting.

Here are some recommended programming fonts:

- FiraCode: A popular programming font with excellent readability and comprehensive ligature support. It's the font used throughout this book.
- JetBrains Mono: Designed specifically for developers by JetBrains (makers of PyCharm), featuring increased letter height for better readability and distinctive character shapes.
- Monaspace: A family of programming fonts published by GitHub with several variants optimized for different use cases.
- Victor Mono: Particularly nice-looking in terminal applications, with a distinctive style that some developers prefer.

All of these fonts are also available as Nerd Fonts versions. Nerd Fonts are patched versions that include additional icons and symbols. While they don't add anything when coding in an editor, they make many terminal-based tools look nicer by enabling icons for file types, git status, and other visual indicators.

Pick the font that feels right to you, install it on your system, and we'll configure it in VS Code below.

7.2. VS Code Setup

In this section, I present how I configure VS Code for research and data science work. These are my personal preferences—feel free to adapt them to your own workflow.

7.2.1. Theme and colors

A good color theme makes your editor pleasant to look at during long coding sessions. My favorite is **Catppuccin**, which offers four variants: one light and three dark.

To install:

1. Open the Extensions panel (`Ctrl+Shift+X` or `Cmd+Shift+X` on Mac)
2. Search for "Catppuccin for VSCode"
3. Install the extension
4. Choose your preferred variant (I use Mocha, the darkest)

Also install **Catppuccin Icons for VS Code** to get matching file icons in the explorer.

> Consistent theming
>
> Catppuccin is available for many applications beyond VS Code. If you like the color scheme, check out their website to theme your terminal, browser, and other tools consistently.

7.2.2. Configuring your font

To set your chosen font in VS Code:

1. Open Settings (`Ctrl+,` or `Cmd+,` on Mac)
2. Search for "font"
3. In "Editor: Font Family", enter your font name (e.g., `Fira Code`)

To enable ligatures, you need to edit the settings JSON directly:

1. In Settings, search for "Font Ligatures"
2. Click "Edit in settings.json"
3. Add or modify:

```
{
    "editor.fontFamily": "Fira Code",
    "editor.fontLigatures": true
}
```

Set `fontLigatures` to `false` if you prefer not to use them.

7.2.3. Disabling the minimap

The minimap (the code preview on the right side of the editor) takes up space without adding much value for most workflows. To disable it:

1. Open Settings
2. Search for "minimap"
3. Uncheck "Editor: Minimap: Enabled"

7.2.4. Adding a ruler

Ruff (which we'll install next) limits lines to 88 characters by default. Adding a visual ruler helps you see this limit:

1. Open Settings and search for "rulers"
2. Click "Edit in settings.json"
3. Add:

```
{
    "editor.rulers": [88]
}
```

This displays a vertical line at column 88, making it easy to arrange multiple files side by side.

7.3. Extensions

7.3.1. Essential extensions

Python extension

The Python extension is fundamental for Python development in VS Code. It provides:

- IntelliSense (smart code completion)
- Linting and error detection
- Debugging support

- Jupyter notebook integration
- Test runner integration

Install it from the Extensions panel by searching for “Python” (by Microsoft).

Ruff for formatting and linting

Ruff is a fast Python linter and formatter. Install the Ruff extension from the Extensions panel.

To configure automatic formatting on save, add to your settings.json:

```
{
    "[python]": {
        "editor.formatOnSave": true,
        "editor.defaultFormatter": "charliermarsh.ruff",
        "editor.codeActionsOnSave": {
            "source.organizeImports": "explicit"
        }
    }
}
```

This configuration:

- Formats your Python code automatically when you save
- Uses Ruff as the default formatter
- Organizes imports (alphabetically, grouped by standard library, third-party, and local imports)
- Removes unused imports

To enable formatting for Jupyter notebooks as well:

```
{
    "notebook.formatOnSave.enabled": true
}
```

> **What Ruff does**
>
> Ruff doesn’t change your code’s behavior—it reformats it to follow consistent conventions. For example, it fixes spacing, line breaks, and quote styles. It also removes unused imports and organizes your import statements.

Markdown

If you write Markdown documents (notes, README files, documentation), these extensions are useful:

- **markdownlint**: Catches formatting issues and enforces consistent style.
- **Markdown All-in-One**: Adds productivity features like keyboard shortcuts, table of contents generation, and list editing.
- **Markdown Preview Mermaid Support**: Previews Mermaid diagrams in Markdown files. Mermaid is a standard diagramming format supported by GitHub and many documentation tools.

7.3.2. Data science extensions

- **Rainbow CSV**: Colorizes CSV files when you open them in VS Code, making it easier to distinguish columns in raw data files.
- **Data Wrangler**: A powerful extension for exploring and cleaning pandas DataFrames interactively. When you load data with pandas, you can click "View Data" to open an interactive data explorer, filter and sort data visually, and generate Python code that replicates your data transformations. This extension is particularly valuable for data cleaning and initial data exploration.

7.3.3. Git extensions

VS Code has built-in Git support, but these extensions enhance it:

- **Git Graph**: Provides a visual representation of your repository's branch structure and commit history. It's essential for understanding complex branching scenarios.
- **GitLens**: By GitKraken, adds inline blame annotations showing who changed each line, detailed commit information, and file and line history. The free version includes many useful features; premium features require a subscription.

7.3.4. AI coding assistant

GitHub Copilot provides AI-powered code suggestions as you type. Install the GitHub Copilot extension and sign in with your GitHub account. A free tier is available with some limitations. Students and academics can apply for GitHub Education, which includes Copilot access.

> **AI tools require verification**
>
> While AI assistants can boost productivity, always verify the code they generate. AI models can produce plausible-looking but incorrect code, especially for domain-specific tasks like financial calculations.

7.3.5. Productivity extensions

- **TODO Tree**: Scans your project for comments containing `TODO` or `FIXME` and displays them in a sidebar panel. This helps you track tasks and issues scattered throughout your codebase.
- **Even Better TOML**: Adds syntax highlighting and validation for TOML files. TOML is the standard format for Python configuration files (`pyproject.toml`), so this extension is useful for any Python project.
- **Vim** (VSCodeVim): For keyboard-driven editing without touching the mouse. Vim bindings have a steep learning curve but dramatically increase editing speed once mastered. If you're new to Vim, also install **Learn Vim**, which provides an interactive tutorial.

7.3.6. Dev containers

The **Dev Containers** extension lets you develop inside containerized environments, isolating your project dependencies and keeping your system clean. See the Dev Containers chapter for details on setting this up.

7.4. Complete Settings Reference

Here's a complete `settings.json` incorporating the configurations discussed:

```
{
    "editor.fontFamily": "Fira Code",
    "editor.fontLigatures": true,
    "editor.minimap.enabled": false,
    "editor.rulers": [88],
    "[python]": {
        "editor.formatOnSave": true,
        "editor.defaultFormatter": "charliermarsh.ruff",
        "editor.codeActionsOnSave": {
            "source.organizeImports": "explicit"
        }
    },
    "notebook.formatOnSave.enabled": true
}
```

You can access your settings file by opening the Command Palette (`Ctrl+Shift+P` or `Cmd+Shift+P`) and selecting "Preferences: Open User Settings (JSON)".

8. Version Control using Git and GitHub

Video

Parts of this chapter are also available as a video tutorial on YouTube.

- Use Github For Academic Research Projects: Track Changes Like a Pro

8.1. What is version control?

Version control, also known as source control, is a system that records changes to a file or set of files over time so that you can recall specific versions later. It's one of the most important tools in the toolkit of any developer or data scientist. It's also very useful for researchers, especially those working with code, but in practice, it is underused in academia. The idea behind version control is quite simple: it allows you to track and manage changes to your projects. Think of it as the "track changes" feature in Microsoft Word, but for all your files and turbocharged with features that make it easy to collaborate with others.

Imagine you're working on a research paper and decide to delete a section. A few days later, you realize that section was crucial. Without version control, you'd have to rewrite that entire section. With version control, you can simply look at your previous versions, find the one that includes the section you need, and restore it. Most of us have some kind of version control in our lives. For example, when you write a paper, you might save different versions of the document as you work on it. This way, if you make a mistake or delete something important, you can go back to a previous version. However, this approach has limitations, and there are better ways to manage versions of your work.

In the context of coding, version control is even more important. As you add new features to your code or fix bugs, it's essential to be able to track these changes. If something breaks, you need to know what was changed so you can figure out what went wrong and how to fix it. Additionally, version control systems allow multiple people to work on the same project simultaneously, making collaboration easier and more efficient while keeping a detailed record of who made what changes and when.

8.2. What is Git?

Git is the most widely used version control system in the world. It was created in 2005 by Linus Torvalds, the creator of the Linux operating system. Torvalds wanted a version control system that was fast, efficient, and capable of handling small to very large projects with ease. Unlike its predecessors, Git was designed to be

decentralized, allowing multiple developers to work on the same project simultaneously without stepping on each other's toes. Like Linux, Git is free and distributed under an open-source license.

At its core, Git allows users to keep a complete history of their project, noting every change made to every file. This feature is akin to having a detailed logbook that captures the evolution of a project over time. With Git, users can branch off from the main project to experiment or work on new features without disrupting the core project. Later, these branches can be merged back into the main project seamlessly. This ability to branch and merge is particularly powerful, preventing conflicts and maintaining the integrity of the original project. Git is also incredibly robust in managing project history, enabling users to revert to previous versions if needed, offering a safety net against errors or unintended consequences of new changes.

> **Not only for code**
>
> Git is a great tool for version control of any kind of file, especially text files. It turns out that if you mainly use LaTeX or Markdown for writing and presentations, you can use Git to track changes in your documents and collaborate with others. Gone are the days of sending around files with names like `paper_v1_final_final_really_final.tex` and `paper_v1_final_final_really_final_revised.tex`!

8.3. What is GitHub?

GitHub, launched in 2008 and acquired by Microsoft in 2018, quickly rose to become the de facto online platform for code management and collaboration. While Git is the engine, GitHub can be thought of as the sleek, user-friendly vehicle that houses this engine. It takes the core functionalities of Git and provides a web-based graphical interface that is intuitive and accessible. GitHub's rise is not just due to its user-friendly nature but also because it functions like a social network for developers and researchers. Users can host their Git repositories, share their work with others, collaborate on projects, and even contribute to others' projects.

> **Not the only game in town.**
>
> While GitHub is the most popular platform for code management and collaboration, it is not the only one. Two other popular platforms are GitLab and Bitbucket. Cloud providers like AWS and Azure also offer Git hosting services. Gitea is an open-source platform that can be self-hosted for free.

8.4. Why use Git and GitHub for research?

For finance researchers, Git and GitHub offer a multitude of benefits. Git is an excellent tool for managing complex research projects. It allows researchers to track changes in their data analysis scripts, models, and even research papers, ensuring a clear audit trail of how the analysis was conducted and conclusions were reached. This level of transparency is crucial not just for personal record-keeping but also for collaborative projects where multiple researchers contribute to a single body of work. In a field where reputation is everything, Git can help researchers maintain a high level of integrity and accountability. The pull request system of

GitHub is particularly beneficial for collaborative projects. It enables researchers to propose, discuss, and review changes before they are integrated into the main project. This not only ensures that every change is scrutinized for accuracy and relevance but also fosters a culture of peer review and collective improvement among collaborators as the project progresses. Furthermore, GitHub's issue-tracking and project management features help researchers organize their tasks, track bugs, and manage project progress transparently.

8.5. Git workflow

8.5.1. Understanding the Core Concepts of Git

Git's power lies in its ability to manage and track changes in your projects, and this is achieved through a set of core functionalities. Let's demystify these key terms:

1. Repository (Repo): The heart of any Git project, a repository is like a project folder but with superpowers. It contains all of your project files along with each file's revision history. You can have local repositories on your computer and synchronize them with remote repositories on GitHub to share and collaborate.

2. Staging: Think of staging as a prep area. When you make changes to files, they don't automatically get saved into your repository. Instead, you selectively add these changes to the staging area, indicating that you've marked these modifications for your next commit.

3. Commits: Committing is the act of saving your staged changes to the project's history. A commit is like a snapshot of your repository at a particular point in time. Each commit has a unique ID and includes a message describing the changes, aiding future you or collaborators in understanding what was modified and why.

4. Push and Pull: These are the methods by which you interact with a remote repository. When you *push*, you are sending your committed changes to a remote repo. Conversely, when you *pull*, you are fetching the latest changes from the remote repo to your local machine.

5. Branching: Branching allows you to diverge from the main line of development and work independently without affecting the main project, often referred to as the `main` branch.[1] It's perfect for developing new features or experimenting.

6. Merging: After you've finished working in a branch, you merge those changes back into the main project. Merging combines the changes in your branch with those in the main branch, creating a single, unified history.

7. Conflicts: Sometimes, when merging branches, Git encounters conflicts - changes that contradict each other. This can happen when two people make changes to the same file. These conflicts need to be manually resolved before completing the merge process.

We will explore these concepts in more detail in the next sections.

[1] Historically, this branch was called `master`, but GitHub has recently changed the default branch name to `main` to avoid the racially charged connotations of the word `master`.

8.5.2. Creating a repository

A repository (or *repo*) is where all the magic happens – it's where your code, documentation, and all other project-related files reside. To create a new repo, simply log into your GitHub account, click on the `+` icon in the top right corner, and select `New repository`.

Naming and Describing Your Repository

Choose a name that succinctly reflects your project. Keep in mind that this name will be part of the URL for your repository, and that it will be used as the default name for the folder when you clone the repository (make a local copy of the repository on your computer). The description field is an opportunity to briefly outline your project's objective. This helps others understand the purpose of your repo at a glance.

Selecting a License

You can also define the license for your project. It is not necessary if you don't intend on sharing this code publicly, but it is a good practice to include a license. When it comes to research code, transparency and accessibility are key. I recommend opting for a permissive license, like the MIT License. This license allows others to freely use, modify, and distribute your work – perfect for fostering open-source collaboration in the research community. GitHub makes it easy to include a license; just select the MIT License from the dropdown menu when creating your repository. Other permissive licenses include the BSD License and the Apache License.

Adding a `.gitignore` file

Before you start adding files to your repo, consider setting up a `.gitignore` file. This file tells Git which files or folders to ignore in a project. Typically, you'll want to exclude certain files from being tracked – like temporary files, local configuration files, files containing sensitive information, or large data files. GitHub offers templates for `.gitignore` files tailored to various programming languages and frameworks, which can be a great starting point. It is available as a dropdown menu option when creating the repository. gitignore.io is another useful resource for generating `.gitignore` files.

Adding a README file

Finally, you'll want to add a `README.md` file to your repository. This file is the first thing visitors will see when they visit your repository on GitHub. It's an essential component of your project, acting as the introduction and guide. Use the README to explain what your project does, how to set it up, and how to use it. This is important even if your project is not public, as it will help you remember how to use your project in the future and facilitate onboarding new collaborators. This file can be written in plain text or formatted using Markdown, a lightweight markup language that is easy to learn and use. GitHub automatically renders Markdown files, making them easy to read and navigate. You can also include images, links, and code snippets in Markdown files. GitHub offers a handy guide to help you get started with Markdown.

8.5.3. Cloning a repository

Cloning a repository creates a local copy of the remote repository on your computer. This allows you to work on the project locally and push your changes to the remote repository when you're ready to share them with others. To clone a repository, you'll need the URL of the remote repository. You can find this by clicking on the green `Code` button on the repository's homepage. If you are using GitHub Desktop, you can clone the repository by selecting `Open with GitHub Desktop`, which will open the repository in GitHub Desktop. You can then select the location where you want to store the repository on your computer and click `Clone`.

If you are not using GitHub Desktop, you can clone the repository using the command line. First, copy the URL of the repository from the repository's homepage by clicking on the green `Code` button, then copying the URL by clicking on the clipboard icon next to the URL. To clone the repository, open the terminal and navigate to the directory where you want to store the repository. Then, run the following command:

```
git clone <url>
```

This will create a new directory with the same name as the repository and download all the files from the remote repository into this directory. You can then open this directory in VS Code and start working on the project. From GitHub Desktop, you can open the repository in VS Code by selecting `Open in Visual Studio Code` from the `Repository` menu.

8.5.4. Tracking changes

Once you have cloned the repository, you can start making changes to the files in the repository. You can create new files, edit existing files, or delete files. You can also move files around or rename them. You can see all your changes in the `Source Control` tab in VS Code. Files will be listed under `Changes` with a `U` if they are new (untracked), a `M` if they have been modified, or a `D` if they have been deleted. You can also see the changes you have made to each file by clicking on the file name.

When you create a new file, it will not be tracked by Git until you add it to the *staging area*. To add a file to the staging area, you use the `Stage Changes` button in the `Source Control` tab in VS Code (the little `+` sign next to a file when you hover over it). You need to do this not only for new files, but for all files that you have modified or deleted since the last commit. Files in the staging area be included in the next commit.

Once you have added one or many changed files to the staging area, you can commit those changes to the repository. To commit changes, you need to enter a commit message describing the changes you have made and then click on the `Commit` button in the `Source Control` tab in VS Code (the checkmark icon). You can also use the keyboard shortcut (`Command+Enter` on Mac or `Ctrl+Enter` on Windows or Linux) to commit your changes. This will create a new commit, i.e., a new snapshot, with the changes you have staged.

You can see all your commits in the `Source Control` tab under `Commits`. You can click on a commit to see the changes that were made in that commit. You can also right-click on the commit to access the commit details, including the commit message, the author, and the date and time of the commit.

8.5.5. Syncing with the remote repository

After committing your changes locally in Visual Studio Code, the next step is to synchronize these changes with your remote repository on GitHub. This process involves two main actions: pulling changes from the remote repository and pushing your local changes to the remote.

Pulling changes from the remote repository

Before you push your changes, it's a good practice to pull any updates that others might have made to the remote repository. This ensures that your local repository is up-to-date. In VS Code, you can pull changes by clicking on the ... (more actions) button in the `Source Control` tab and selecting `Pull`. Alternatively, you can use the keyboard shortcut (`Command+Shift+P` on Mac or `Ctrl+Shift+P` on Windows/Linux) and type `Git: Pull` in the command palette. Pulling changes will merge updates from the remote repository into your local branch. If there are no conflicts, the merge will happen automatically.

Pushing changes to the remote repository

Once your local branch is up-to-date and you've committed your changes, you're ready to push these changes to the remote repository. In the `Source Control` tab, click on the ... button and select `Push`. This will upload your commits to the remote repository on GitHub. You can also use the keyboard shortcut (`Command+Shift+P` on Mac or `Ctrl+Shift+P` on Windows/Linux) and type `Git: Push` in the command palette. If you're pushing to a branch that doesn't exist on the remote, VS Code will automatically create this branch in the remote repository.

Resolving merge conflicts

Occasionally, when you pull changes from the remote repository, you may encounter merge conflicts. These occur when changes in the remote repository overlap with your local changes in a way that Git can't automatically resolve. VS Code provides tools to help resolve these conflicts. Conflicted files will be marked in the `Source Control` tab. You can open these files and choose which changes to keep. After resolving conflicts, you'll need to stage and commit the merged files before pushing.

Regularly pulling and pushing changes will keep your local and remote repositories synchronized. This is crucial in collaborative projects to ensure everyone is working with the most current version of the project.

8.5.6. Branching and merging

Before using Git, whenever I wanted to try something new in my code, I would make a copy of the entire project folder and work on that copy. This was a tedious process, and it was easy to lose track of which version was the most recent. With Git, branching makes this process much easier. Branching allows you to create a copy of your project, called a branch, and work on that branch without affecting the main project. Once you're satisfied with the changes you've made in your branch, you can merge those changes back into the main project. This process is much more efficient and less error-prone than manually copying and pasting files.

Creating a New Branch

In VS Code, you can create a new branch by clicking on the branch name in the bottom left corner, then selecting `Create new branch...`. Give your branch a descriptive name that reflects its purpose. You can switch between branches by clicking on the branch name in the bottom left corner and selecting the branch you want to work on.

After creating and switching to your new branch, any changes you make are confined to that branch. You can stage and commit changes in this branch as you would in the main branch.

You can also choose to publish your branch to the remote repository. This will create a copy of your branch on GitHub. This is useful if you want to collaborate with others on this branch, or to use GitHub to backup the branch. Note that once the branch is published, others who have access to the repository will be able to see that branch. To publish your branch, click on the `...` button in the `Source Control` tab and select `Publish Branch...`.

Merging Branches

Once you've completed the work in your branch and you're satisfied with the changes, you'll want to merge these changes back into the main branch. Before merging, ensure your branch is up-to-date with the main branch. You can do this by checking out the main branch and pulling the latest changes, then switching back to your branch and merging the main branch into it. After that, you are ready to merge your branch into the main branch. After merging, you can delete your branch if you no longer need it. This avoids cluttering the repository with branches that are no longer needed.

In the `Source Control` tab, click on the `...` button, select `Merge Branch...`, and choose the branch you want to merge into your current branch. If there are no conflicts, VS Code will complete the merge. VS Code will also ask you if you want to delete the merged branch.

Merge conflicts happen when the same lines of code have been changed differently in both branches. VS Code will notify you if there are conflicts that need resolution. The first time, Git will also need you to confiure how you want to handle merge conflicts by entering one of the following commands in the terminal:

1. `git config pull.rebase false`: This command sets the pull behavior to merge. When you pull from a remote repository, Git will merge any incoming commits with your current branch. *This is the one I usually use.*
2. `git config pull.rebase true`: This command sets the pull behavior to rebase. Instead of merging incoming commits, Git will reapply your local commits on top of the incoming commits, creating a linear commit history.
3. `git config pull.ff only`: This command sets the pull behavior to fast-forward only. Git will only update your branch if it can fast-forward, meaning the main branch has not changed since you created your branch. If the main branch has new commits, Git will not pull the changes and you'll need to manually merge or rebase.

Conflicted files will be marked in the `Source Control` tab. Open these files, and VS Code will highlight the conflicting changes. Choose which changes to keep, then save the file, stage, and commit the resolved files. Once all conflicts are resolved and changes are committed, the branches are successfully merged. If you've

merged into your local main branch, don't forget to push these changes to the remote repository to keep everything synchronized.

8.5.7. Pull requests and code reviews

A pull request (PR) is a method in GitHub to propose changes from one branch to another, typically from a feature into the main branch. It's a request to *pull in* your changes. The name is a bit misleading because it's not related to the `pull` command in Git. You can think of it as a "merge request" instead. When you create a PR, you're initiating a discussion about your proposed changes. Your collaborators can review the code, leave comments, request changes, or approve the PR.

Creating a Pull Request in GitHub

Once you have pushed your branch to the remote repository, you can create a PR. Navigate to the repository on GitHub.com. GitHub often shows a prompt to create a PR for recent branches. If not, go to the `Pull Requests` tab and click `New pull request`. Select your branch and the branch you want to merge into (usually the `main` branch). When creating a PR, include a clear title and a detailed description of the changes. This helps reviewers understand the context and purpose of the changes. You can also assign reviewers to the PR, add labels, and set a milestone. Once you're satisfied with the PR, click `Create pull request`. Any assigned reviewers will be notified of the PR and can begin reviewing it.

Code Reviews

Collaborators can review the changes in a PR by navigating to the `Files changed` tab within the PR. Reviewers can leave comments on specific lines of code, general comments on the PR, and suggest changes. They can also pull the branch locally and test the changes themselves. Once the review is complete, the reviewer can approve the PR, request changes, or leave a comment. If changes are requested, the PR author can make the requested changes and push them to the branch. The PR will be automatically updated with the new changes. Once the PR is approved, it can be merged into the target branch. Based on the feedback, you might need to make additional commits to your branch. These updates will automatically appear in the PR. This back-and-forth can continue until the changes are satisfactory.

Merging the Pull Request

Once the PR is approved and any conflicts are resolved, you can merge it into the target branch. This is typically done via the 'Merge pull request' button on GitHub. After merging, it's a good practice to delete the feature branch from the remote repository to keep the branch list tidy.

Best Practices for Pull Requests and Code Reviews

- **Small, Focused Changes**: Aim for smaller, manageable PRs that focus on a specific feature or fix. This makes code reviews more efficient and less overwhelming.
- **Clear Communication**: Use clear, descriptive messages in both your PRs and commits. This helps reviewers understand your thought process and the changes made.
- **Constructive Feedback**: When reviewing, offer constructive and respectful feedback. Code reviews are not just about finding mistakes but also about sharing knowledge and improving the codebase collaboratively.

Pull requests and code reviews are vital for maintaining high-quality code and fostering collaboration in your finance research projects. While not yet commonly used in academia, I have found them the perfect tools for collaborating on research projects. They ensure that every change is scrutinized and understood by all collaborators, and they foster a culture of peer review and collective improvement as the project progresses.

GitHub offers many other features that can be useful for research projects. I list them at the end of this post and will cover them in a future post.

8.6. GitHub for research code

In empirical finance research, the ability to reproduce results is more important now than ever, especially that most top journals require authors to share their code and data. In this section, I will discuss some best practices I have adopted for using GitHub to manage research code, with an emphasis on reproducibility, documentation, and effective use of GitHub's features.

The first thing to consider after creating a new repository is the structure of your project. A well-organized project is easier to navigate and understand, and it makes it easier for others to reproduce your work. There is no one-size-fits-all approach to organizing a project, but the following project structure is a good starting point:

```
project
├── data/
├── docs/
├── output/
    ├── figures/
    └── tables/
├── src/
├── .gitignore
├── .env
├── .env-example
├── conf.yaml
├── LICENSE
├── uv.lock
├── pyproject.toml
├── README.md
```

```
└── requirements.txt
```

So, which files should you commit to your repository? Here are some guidelines:

You should include:

- **Configuration files**: Files like `.json`, `.yml`, or `.ini` are crucial for ensuring that your project can be set up and run by others with the exact same parameters you used. In my example the `conf.yaml` file contains the configuration parameters for the project and should be included in the repository.
- **Source code**: Include all scripts and code files that are essential for your analysis or model. In my example, the `src` directory contains all the Python scripts used in the project.
- **Documentation**: Any files that help explain your project, especially markdown files with notes. In my example, the `docs` directory contains the documentation for the project and should be included in the repository. If your documentation is generated from source files, such as Markdown or Latex, then you should include the source files in the repository, not the generated files. Your repository should also include a `README` file at the root of the repository that provides an overview of your project, its purpose, and how to use it.
- **Dependencies**: For projects in languages like Python, a file listing the dependencies is essential. This file lists all the external libraries and their specific versions needed for your project. This ensures that anyone cloning your repository can easily install the necessary dependencies and run your code in an environment identical to yours. In my example, I use uv to manage dependencies, so I include the `pyproject.toml` and `uv.lock` files. The lock file records the exact versions of all dependencies, ensuring reproducible environments across machines. I also include a `requirements.txt` file for users who prefer to install dependencies using `pip` instead of uv.
- **`.gitignore` file**: This file tells Git which files or folders to ignore in a project. Typically, you'll want to exclude certain files from being tracked – like temporary files, local configuration files, files containing sensitive information, or large data files. GitHub offer templates, but they seem to be missing a few things. For example, if you are on Mac you will want to add `.DS_Store` to your ignore file. gitignore.io is another useful resource for generating `.gitignore` files that are much more comprehensive. Make sure to also add the `.gitignore` file to your repository.

Finally, make sure that you include in the `.gitignore` file all the files and directories that you should not include in the repository.

You should not include:

- **Data files**: While large datasets might not be feasible to store on GitHub, even small datasets can be problematic if they are updated often. Instead, consider including sample datasets or scripts that automatically fetch or generate data, or sharing your data among collaborators using a cloud storage service like Dropbox or Google Drive. There exist tools like DVC that can help you manage large datasets with version control, but I have not used them myself.
- **Sensitive data and local configuration files**: Do not include sensitive information like passwords or API keys, or computer-specific configuration parameters such as local paths. Instead, you should include an example file with the expected parameters that need to be set in the configuration file. In my example, I use a `.env` file to store sensitive and local information, and I include a `.env-example` file that contains the name of the environment variable that needs to be set in the `.env` file. I would then include

the `.env-example` file in the repository, but not the `.env` file. I also include the `.env` file in the `.gitignore` file so that it is not included in the repository.
- **Output**: You should not include output files in the repository. Instead, you should include the code that generates the output files. Every collaborator should be able to generate the results in his environment. In my example, the `output` directory contains the figures and tables generated by the code in the `src` directory.

8.7. Git and Jupyter notebooks

Jupyter Notebooks are a popular tool for data analysis and visualization. They allow users to combine code, text, and visuals in one document, making it easy to share and collaborate on data science projects. However, Jupyter Notebooks have many shortcomings when it comes to replicability and using them with Git can be challenging.

8.7.1. Challenges with Git and Jupyter notebooks

Jupyter Notebooks, while an excellent tool for data analysis and visualization, present unique challenges when used with Git. The core issue lies in their format: Notebooks save both the input (code) and the output (results, graphs, etc.) in a single JSON file. This means that even small changes in the code can lead to large changes in the file, making it difficult for Git to handle diffs and merges effectively. The output sections, especially those with visual content, can create "noise" in version control. When different users run the same notebook, slight differences in output can appear, leading to unnecessary conflicts. Because Git keeps track of the full history of the notebook, the size of the repository can grow quickly, especially if the notebook contains large outputs such as images. This can make it difficult to share and collaborate on notebooks.

8.7.2. VS Code Notebook Diff Viewer

Recognizing these challenges, tools like Visual Studio Code have introduced features to help. The VS Code diff viewer (the tool that shows differences in files due to changes) supports Jupyter notebooks, allowing users to compare and understand changes between notebook versions more easily. This tool provides a clearer visualization of differences in the code, reducing the complexity involved in tracking changes in notebooks in Git.

8.7.3. Using Notebooks with Online Platforms

Despite these challenges, Jupyter Notebooks remain a popular and powerful tool for data analysis and research. Their interactive nature and the ability to combine code, text, and visuals in one document make them invaluable.

Platforms like Binder and Google Colab integrate well with Jupyter Notebooks hosted on GitHub. These platforms can automatically create interactive, shareable environments from notebooks, making them more accessible for collaborative work and education. By using these platforms, researchers can share their notebooks

in a more user-friendly and interactive format, ensuring that others can easily replicate and experiment with their findings.

8.8. GitHub for writing

GitHub is not just for code; it's also an excellent platform for tracking your writing, especially if you are using formats based on plain-text files such as Markdown (like the Quarto publishing system) and LaTeX.

The same principles for organizing code projects apply to writing projects. You should include all the files that are essential to generate the output of your project, such as the source (e.g. `.md`, `.tex`, and `.bib`) files, configuration files, and tables and figures. You should avoid including the output files, such as PDFs, or HTML files.

8.9. Tagging releases for milestones

There are times when you want to create a snapshot of your project, including the output, at a specific point in time. For example, when you submit a paper to a journal, you want to create a snapshot of the project at that point in time. This allows you to keep track of the changes made in between revisions. GitHub provides a way to do this using tags and releases.

When you reach a significant milestone in your writing – such as the completion of a draft, submission to a journal, or final revisions – you can create a tag and a corresponding release.

To create a tag and release, head to the repository on GitHub.com and click on `Create a new release` under `Releases` in the right sidebar. Enter a tag version number and a title for the release. You can also add release notes summarizing the changes or updates in this version. Finally, attach the output files (e.g. PDFs) to the release. Click `Publish release` to create the release. You can then download the release files or share the release link with others.

8.10. Publishing your code on GitHub

In empirical finance academic research, sharing your code has become increasingly important. Publishing your code enhances the transparency and reproducibility of your research. It allows peers to review, replicate, and build upon your work, contributing to the collective knowledge of the field. Making your code available can also increase the citation and impact of your research, as it provides tangible artifacts that others can use and reference. Finally, it is also a requirement for publishing in many journals, including the top ones.

Journals will publish your code alongside your paper, so why should you also publish it on GitHub?

For me, the main reason is to keep control over my code. By publishing your code on GitHub, you retain control over it. You can continue to make changes to it, and update it as needed. Other researchers who visit your GitHub repository can also be exposed to your other work, increasing the visibility of your research.

Finally, GitHub offers a platform for collaboration and feedback, allowing others to flag issues, contribute to your work, and build upon it.

To publish your code on GitHub, all you need to do is set the visibility of your repository to public. If you don't want to share the full history of your code, you can create a new repository and upload the latest version of your code.

Make sure to include a README file that explains what your project does, how to set it up, and how to use it. Documentation is key to making your code accessible to others (and to reducing the number of questions you get about your code). You can also include instructions on how to cite your code in the README file. Finally, you should also include a LICENSE file to clearly state how others can use your code.

Once you have completed these steps, your code is published! If you want a DOI for your repository, you can use Zenodo, which allows you to mint a DOI for your GitHub repository.

8.11. Other GitHub features for academic researchers

In addition to the core features of GitHub, many other tools and functionalities can be useful for academic researchers. I plan on covering most of them in future posts. Here are the ones that I use the most:

8.11.1. Project Management Tools

GitHub offers several tools to help you manage your projects, including *Projects*, *Issues*, *Discussions*, and *Wikis*. These tools can be used to organize your work, track tasks, and collaborate with others.

8.11.2. GitHub Copilot

GitHub Copilot is an AI coding assistant. It can do code completion, suggest functions, and even generate code based on comments. There is also a Copilot Chat powered by GPT-4 that can answer questions about code while being aware of the context. When you allow it, it can consult your private repositories to provide more relevant suggestions.

Seriously, if you haven't tried it yet, you should. It's a game-changer, and new features are being added all the time. And it will work for text too if you write your Markdown or LaTeX files in VS Code.

8.11.3. GitHub Pages

GitHub Pages is a free service that allows you to host static websites directly from GitHub. This can be useful for hosting project websites, blogs, or personal websites. My personal website and this blog are both hosted on GitHub Pages.

8.11.4. GitHub Classroom

GitHub Classroom simplifies the use of GitHub in classroom settings. It's a toolset that automates the repetitive tasks involved in grading and feedback, making it easier to use GitHub for coursework and assignments in a research or academic context. I have been using it for three years and it has been a game-changer for me. While it's not bug-free, it has saved me countless hours of grading and feedback. Automated grading has a monthly limit after which you need to pay, but the cost is minimal and well worth it.

8.11.5. GitHub Actions

GitHub Actions is a powerful tool that allows you to automate workflows. You can set up CI/CD pipelines[2] to automate testing, building, and deploying your applications or research code. GitHub Actions are small scripts that run in response to events in your repository, such as commit or pull requests. For researchers, it can be used to automate the testing of code or even automate routine data processing tasks.

8.11.6. GitHub Codespaces

GitHub Codespaces provide a fully featured cloud development environment accessible directly from GitHub. Your code lives in a remote server, and you get a complete VS Code environment in your browser. This can be particularly useful for researchers who want to quickly experiment with code or collaborate without the need to set up a local development environment. It is also great to ensure maximum replicability of the code you distribute, as the environment is identical for everyone.

[2] Continuous integration and continuous development

9. Using AI for Coding in Empirical Research

Artificial intelligence tools have transformed how programmers write code. Large language models can explain syntax, generate functions from natural language descriptions, and debug error messages in seconds. For students learning Python for empirical finance research, these tools offer an appealing shortcut: why struggle with cryptic error messages when an AI can explain them? Why write boilerplate code by hand when an AI can generate it instantly?

This chapter addresses a fundamental tension. AI coding tools can genuinely accelerate your work and reduce frustration, but they can also undermine the skill formation that makes you a capable researcher. The difference between these outcomes depends entirely on how you use these tools. Used wisely, AI becomes a force multiplier for your growing expertise. Used carelessly, it becomes a crutch that prevents you from developing the understanding you need.

We begin with the core trade-offs, then cover the practical landscape of AI coding tools, and conclude with a workflow designed to help you benefit from AI assistance while still building genuine programming competence.

9.1. Why This Chapter Exists

The premise of this chapter is straightforward: AI can accelerate learning, but it cannot replace it. If you cannot read, understand, and debug the code that AI produces, you are flying blind. Learning Python remains non-negotiable for credible empirical research. AI shifts where your effort goes, not whether effort is required.

Consider what happens when an AI writes code for you. The tool produces syntactically correct Python that might even run without errors on your first attempt. But what happens when you need to modify that code for a different dataset? What happens when it produces incorrect results that look plausible? What happens when a collaborator asks you to explain your methodology? If you cannot answer these questions, the AI-generated code is a liability rather than an asset.

The goal of this chapter is not to discourage you from using AI tools. They are genuinely useful, and most professional programmers now use them in some capacity. The goal is to help you use these tools in ways that build rather than erode your capabilities. This requires understanding what AI tools can and cannot do, recognizing when to rely on them and when to step back, and developing habits that keep you in control of your own code.

AI does not understand your research

AI coding assistants have no conception of what makes empirical research valid. They cannot distinguish between code that runs and code that correctly implements your methodology. They cannot verify that your variable definitions match your research design. They cannot ensure that your sample selection avoids look-ahead bias. These judgments require domain expertise that only you can provide.

9.2. AI, Skill Formation, and Career Constraints

The most important reason to develop genuine Python skills, even when AI tools are available, is that over-reliance on AI slows long-run skill acquisition. Programming proficiency comes from struggling with problems, making mistakes, and building mental models of how code works. When AI removes that struggle, it also removes the learning.

Think about the difference between using a calculator for arithmetic and understanding how arithmetic works. Calculators are faster and more reliable for computation, but if you never learned arithmetic, you cannot estimate whether an answer is reasonable. You cannot catch errors. You cannot extend the calculation in ways the calculator does not support. The same dynamic applies to AI and coding. AI can produce code faster than you can type it, but if you never developed the underlying understanding, you cannot evaluate, modify, or debug what it produces.

9.2.1. The industry reality

Many financial institutions restrict or prohibit external AI tools. Banks, asset managers, and hedge funds handle sensitive data and proprietary strategies. Sending code snippets or error messages to external AI services creates compliance and security risks that many firms will not accept. Some institutions run air-gapped systems where internet access is simply unavailable. Others have policies that prohibit sharing any code or data with third parties, including AI providers.

If you are targeting industry roles after your degree, this matters. Code you write with AI assistance still needs to be maintained without it. When you join a firm with restrictive policies, you need to be able to read, understand, and modify code on your own. The skills you develop now determine whether you will be effective in those environments.

Even in academic settings, AI dependence creates problems. Universities and journals increasingly have policies that restrict or require disclosure of AI use in research. Peer reviewers may ask you to explain or modify your methodology. Collaborators may need to extend your code. You may need to debug issues years after the original analysis. In all these cases, you need genuine understanding of what your code does and why.

9.2.2. Building transferable skills

The bottom line is that AI is a supplement, not a crutch. The goal is to develop skills that remain valuable regardless of what tools are available. This means understanding Python syntax, data structures, and control flow well enough to write code from scratch when necessary. It means being able to read and understand code you did not write. It means having mental models of how your analysis works that go beyond the specific implementation.

When you use AI tools, pay attention to whether you are learning or just producing output. If you find yourself repeatedly asking the AI to solve similar problems, that is a signal to step back and build understanding. If you cannot explain what the AI-generated code does, you should not use it.

9.3. Ethical, Academic, and Scientific Constraints

Before discussing how to use AI tools effectively, we need to address the constraints that govern their use in academic and professional settings. These are not optional guidelines. They are requirements that determine whether your work is acceptable.

9.3.1. Academic integrity and authorship

The fundamental principle is simple: responsibility for correctness always lies with the researcher. When you submit code as part of a research project or assignment, you are asserting that the code correctly implements your methodology and that you understand what it does. AI assistance does not change this responsibility. If AI-generated code contains errors that invalidate your results, you bear the consequences.

Different institutions and courses have different policies about AI use. Some prohibit it entirely for certain assignments. Others permit it with disclosure requirements. The specific rules vary, but the underlying principle is consistent: you must be able to explain and defend every line of code you submit. If you cannot, the work is not genuinely yours, regardless of how it was produced.

9.3.2. Reproducibility and auditability

Empirical research must be reproducible. Other researchers should be able to examine your methodology, run your code, and verify your results. This requires that your code be inspectable and explainable. AI-generated code that you do not understand fails this requirement, even if it produces the correct output.

When reviewers or collaborators ask about your implementation, you need to be able to explain why you made specific choices. Why did you winsorize at the 1st and 99th percentiles rather than the 5th and 95th? Why did you cluster standard errors at the firm level rather than the industry level? These questions require understanding that goes beyond the code itself to the research design it implements.

9.3.3. Data confidentiality

Never paste proprietary data, credentials, or restricted information into AI tools. This applies to actual data values, but also to code that reveals the structure of proprietary datasets or the logic of trading strategies. Most AI tools send your input to external servers for processing. Even tools that claim privacy protections may retain data for training or analysis.

In practice, this means being careful about what you share. Error messages that include data values should be sanitized before sharing. Code that implements proprietary methodology should be described in general terms rather than pasted directly. When in doubt, assume that anything you share with an AI tool could become public.

> A simple rule for submissions
>
> If you would not submit work without AI, you should not submit it with AI. AI can help you produce better work faster, but it cannot make you capable of work you could not otherwise do. When you submit code, you are asserting that you could have written it yourself given sufficient time. If that is not true, the submission is not appropriate.

9.4. Mental Models for Using AI Effectively

The key to using AI tools effectively is having the right mental model for what they are and what they can do. This section provides frameworks for thinking about AI that will help you use it productively, especially as a beginner.

9.4.1. AI as a junior assistant

The most useful mental model is to think of AI as a junior assistant that writes drafts quickly and carelessly. Like a junior employee, AI can produce output fast, but that output requires review and correction. The assistant has broad knowledge but limited judgment. They will confidently produce work that contains subtle errors. They will follow instructions literally even when those instructions are ambiguous or misguided.

This mental model has several implications. First, you should expect to review and edit everything the AI produces. Just as you would not submit a junior employee's first draft, you should not use AI-generated code without careful examination. Second, the quality of your instructions matters enormously. Vague requests produce vague outputs. Specific, well-structured prompts produce better results. Third, you remain responsible for the final product. The assistant helps, but you make the decisions.

9.4.2. Why precise prompts matter

Precise prompts matter more than model choice. The difference between a good prompt and a bad prompt often exceeds the difference between AI models. A well-crafted prompt includes context about what you are

trying to accomplish, specific requirements for the output, and examples of what success looks like. A poor prompt leaves the AI guessing about your intentions.

Consider the difference between these prompts:

- Poor: "Write code to calculate returns"
- Better: "Write a Python function that calculates simple returns from a list of prices. The function should take a list of floats representing daily closing prices and return a list of floats representing daily returns. Handle the edge case of the first day, which has no previous price, by returning None for that position."

The second prompt specifies the input format, output format, programming language, and edge case handling. It gives the AI enough information to produce something useful. The first prompt leaves all these decisions to the AI, which may or may not match your needs.

9.4.3. The iteration loop

Effective AI use follows an iteration loop: prompt, inspect, test, revise. You start with a prompt describing what you need. You inspect the output to understand what the AI produced. You test the code to verify it works correctly. You revise either the prompt or the code based on what you learned.

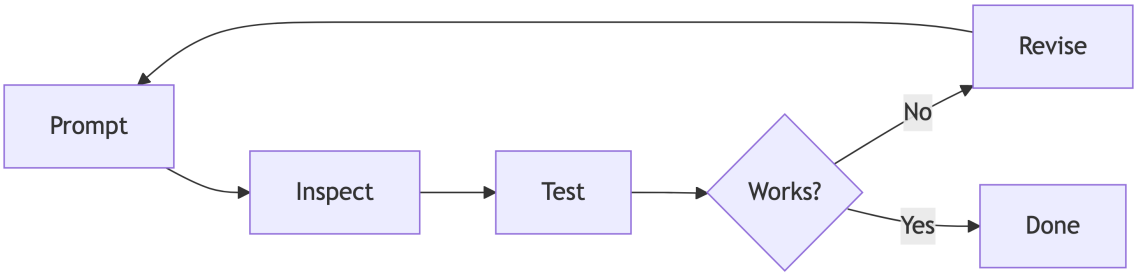


Figure 9.1.: The AI-assisted coding iteration loop

This loop emphasizes that AI assistance is not a one-shot process. You should expect multiple iterations, especially for complex tasks. Each iteration teaches you something about both the problem and the AI's capabilities. Over time, you learn to write better prompts and to anticipate where AI is likely to make mistakes.

9.4.4. Treat every response as a hypothesis

Perhaps the most important mental model is to treat every AI response as a hypothesis, not an answer. The AI is proposing a solution that might be correct. Your job is to test that hypothesis by reading the code, understanding what it does, and verifying that it behaves correctly.

This mindset prevents the most dangerous failure mode: accepting AI output without verification. When you treat AI responses as hypotheses, you naturally engage your critical faculties. You ask questions. You check assumptions. You test edge cases. This engagement is precisely what builds understanding.

9.5. General-Purpose Chatbots for Research Coding

General-purpose AI chatbots are the most familiar category of AI coding tools. These are web-based interfaces where you type questions in natural language and receive conversational responses. The major tools in this category include OpenAI ChatGPT, Anthropic Claude, Google Gemini, and Microsoft Copilot in chat mode. Each has different strengths and pricing, but for most research coding tasks, the differences are less important than how you use them. For research coding, they serve as explanation-first tools: they excel at helping you understand rather than at producing production-ready code.

9.5.1. What they are good at

Chatbots excel at explaining syntax and error messages. When you encounter a Python error that you do not understand, a chatbot can explain what went wrong in plain language. It can describe what the error message means, why it occurred, and how to fix it. This is often faster and more helpful than searching documentation or Stack Overflow.

They are also good at translating ideas into rough code sketches. If you have a conceptual understanding of what you want to do but are unsure how to express it in Python, a chatbot can provide a starting point. The code may need refinement, but it gives you something concrete to work with and learn from.

Clarifying library usage is another strength. Python has thousands of packages with varying documentation quality. A chatbot can explain how to use a specific function, what parameters it accepts, and what it returns. It can compare alternatives and explain trade-offs. This is particularly helpful when learning new libraries.

9.5.2. What they are bad at

Chatbots are poor at designing correct empirical workflows. They do not understand your research question, your data structure, or the methodological requirements of your field. The most recent models, especially those with reasoning capabilities, are improving in this area, but not yet to the point where you can trust them to be reliable without verification. They can produce code that runs but implements the wrong analysis. Detecting these errors requires domain expertise that the chatbot lacks.

They also struggle with understanding your project context by default. Each conversation with a chatbot starts fresh. The AI does not know what code you have already written, what data you are working with, or what you have tried before. You need to provide this context explicitly, which is time-consuming and error-prone.

9.5.3. Typical research uses

For empirical finance research, chatbots are most useful for:

- **Debugging tracebacks**: Paste an error message and ask for an explanation. The chatbot can often identify the problem and suggest fixes.
- **Refactoring simple logic**: Describe what your current code does and ask for a cleaner implementation. Review the suggestion carefully before adopting it.

- **Learning new concepts**: Ask for explanations of Python features, statistical methods, or library functions. Use the chatbot as an interactive tutorial.
- **Generating boilerplate**: Request template code for common patterns like reading CSV files or setting up matplotlib figures. Customize for your specific needs.

Using chatbots for learning

Chatbots make excellent tutors. When you do not understand a piece of code, ask the AI to explain it line by line. When you are confused about a concept, ask for multiple examples at different difficulty levels. When you get stuck on a problem, ask for hints rather than solutions. This approach builds understanding rather than dependence.

9.6. In-Editor Code Completion and Inline Assistance

In-editor completion tools provide a different kind of AI assistance. Rather than conversational interaction, they offer continuous, low-friction suggestions as you type. The major tools in this category are GitHub Copilot, which integrates with Visual Studio Code and other editors, and Cursor, an editor with deep AI integration built in. Cursor is a fork of VS Code, so it offers a familiar interface and supports most VS Code extensions, excluding those published by Microsoft. These tools analyze your current file and context to predict what you are likely to write next, then offer to complete it for you.

If you have used auto-complete on your phone, you already understand the basic experience. The AI predicts what you are likely to type next and offers to complete it for you. Most of the time the suggestions are helpful and save keystrokes. But occasionally the AI makes silly mistakes, inserting words that are grammatically correct but contextually wrong. Code completion works the same way: usually helpful, occasionally nonsensical, and always requiring your judgment about whether to accept.

9.6.1. How completion differs from chat

The interaction model is fundamentally different from chatbots. With a chatbot, you formulate a request, submit it, wait for a response, then evaluate and possibly revise. With completion tools, suggestions appear automatically as you type. You accept with a keystroke or ignore by continuing to type. The feedback loop is measured in seconds rather than minutes.

This difference has important implications. Completion tools are embedded in your normal workflow. You do not need to context-switch to another application. The AI sees your entire file and often your entire project, so it has much more context than a chatbot conversation. Suggestions are smaller and more incremental, completing lines or functions rather than generating entire programs.

9.6.2. Why this is often safer for beginners

In-editor completion is often safer for beginners because it produces smaller changes that are easier to evaluate. When a chatbot generates a 50-line function, understanding and verifying it requires significant effort. When

a completion tool suggests finishing the line you are already writing, you can evaluate it immediately. The suggestion is within the context you already understand.

The immediate visual context also helps. You see the suggestion alongside your existing code. You can compare it to what you were planning to write. Obvious errors are more apparent because they conflict with the surrounding code that you wrote and understand.

9.6.3. Common risks

The primary risk with completion tools is silent logical errors. The AI might suggest syntactically correct code that does not do what you intend. Because suggestions appear quickly and accepting them is easy, you might accept code without fully processing what it does. This is especially dangerous for statistical operations where the code runs without errors but produces incorrect results.

Over-engineered suggestions are another issue. The AI might suggest more complex solutions than necessary, introducing abstractions or edge case handling that you do not need. This complexity makes the code harder to understand and maintain. Simpler code that you fully understand is usually better than sophisticated code that you do not.

9.6.4. Improving suggestions with context

Completion tools work better when they have more context about your code. The AI analyzes your file and project to understand what you are trying to do, and richer context leads to better suggestions.

Type hints and docstrings, discussed in Chapter 4, serve double duty here. They make your code more readable for humans and more understandable for AI. When you declare that a function takes a `pd.DataFrame` and returns a `float`, the AI can suggest code that correctly uses DataFrame methods and returns an appropriate value. When you write a docstring explaining that a function calculates annualized volatility from daily returns, the AI understands the domain and can suggest relevant implementations.

This creates a virtuous cycle. Writing type hints and docstrings is good practice regardless of AI tools. But if you use completion tools, that investment pays additional dividends through better suggestions. The same practices that make your code maintainable also make it more AI-friendly.

Conversely, poorly documented code with no type hints gives the AI little to work with. Suggestions will be more generic and less likely to match your intent. If you find that completion suggestions are consistently unhelpful, consider whether adding context to your code might improve them.

9.6.5. When to accept a suggestion

A useful rule of thumb: accept a suggestion when you could have written it yourself, just more slowly. This standard ensures that you understand what you are accepting. The AI is saving you keystrokes, not doing your thinking.

If a suggestion surprises you, pause before accepting. Either you are learning something new (good), or the suggestion is wrong (bad). Either way, the surprise is a signal that you should engage more deeply rather than just accepting. Take time to understand why the AI suggested what it did.

9.7. Coding Agents

Coding agents represent the most powerful and most dangerous category of AI coding tools. Unlike completion tools that suggest small changes, agents can make extensive modifications across multiple files. Unlike chatbots that produce isolated code snippets, agents can execute code, observe results, and iterate. The major tools in this category include GitHub Copilot in agent mode, Anthropic's Claude Code, and OpenAI Codex. Both Claude Code and OpenAI Codex are available as command-line tools and as VS Code extensions. Cursor also includes deep agent integration. These tools evolve rapidly, so specific recommendations become outdated quickly, but at the time of writing, Claude Code with the Opus 4.5 model is my tool of choice.

For all AI coding tools, but especially agents, reading the product documentation is essential to achieve the best results. Each tool behaves differently, with its own strengths, limitations, and recommended workflows. Adapting your approach to the specific tool you use will greatly improve the quality of the output you get. Time spent learning the tool's features and best practices pays off quickly.

Video

In this video, I go through an example use case of Claude Code with a financial research application.

- AI-Powered Data Analysis with Claude Code

9.7.1. What coding agents actually do

Agents operate with a significant degree of autonomy. You describe a goal, and the agent takes a series of actions to achieve it: writing code, running tests, reading error messages, and revising its approach. Some agents can browse documentation, search codebases, and even interact with external services. They do in minutes what might take you hours.

This capability is transformative for experienced developers who can evaluate agent output quickly. An agent can implement a feature, write tests, and handle edge cases while the developer reviews and directs. The human remains in control but operates at a higher level of abstraction.

9.7.2. Why they are dangerous for novice programmers

For novice programmers, agents are dangerous precisely because of their power. An agent might modify your code in ways you do not understand. It might introduce bugs that are difficult to detect because you do not know what was changed. It might solve problems in ways that work but that you cannot maintain or extend.

The core issue is verification. When an agent makes changes across multiple files, verifying correctness requires understanding all those files. A novice lacks this understanding. They may accept changes that seem

to work but contain subtle errors. They may lose track of what their code does and why. They may find themselves unable to make further modifications without agent assistance.

> ⚠ The danger of "fix everything" requests
>
> Never ask an agent to "fix everything" or "make it work" on code you do not understand. This request gives the agent maximum autonomy with minimum oversight. You have no way to evaluate whether the resulting changes are correct. Even if the code runs, it may not implement what you intended. Even if it produces plausible output, that output may be wrong.

9.7.3. Appropriate research use cases

Agents are appropriate when you already understand the code being modified. Useful applications include:

- **Refactoring code you already understand**: If you know what the code does and want to improve its structure, an agent can help. You can verify that the refactored version preserves the original behavior.
- **Generating tests or documentation drafts**: Agents can produce initial test cases or docstrings that you then review and refine. The human provides the specification; the agent provides the implementation.
- **Applying consistent changes across files**: When you need to rename a variable, update a function signature, or apply a similar change in many places, agents can do this quickly and reliably.
- **Rapid prototyping**: Agents can quickly produce working code to test whether an idea is feasible or to explore how a library works. Treat this as throw-away code for learning and experimentation, not as the foundation for your actual analysis.

9.7.4. Inappropriate use cases

Agents are inappropriate when you cannot verify their output:

- **Writing core empirical logic**: The code that implements your research methodology must be code you understand. Agents should not write this code for you.
- **Modifying unfamiliar code**: If you do not understand code before the agent modifies it, you definitely will not understand it after.
- **Debugging without understanding**: When code does not work, the solution is understanding, not automation. An agent might fix the symptom while leaving the underlying problem.

9.7.5. Version control is essential

Using version control with Git is not optional when working with coding agents. Agents make extensive changes across multiple files, and you need the ability to review those changes before accepting them and to revert if something goes wrong. Git's diff tools let you see exactly what the agent modified, line by line. Without version control, you have no systematic way to understand or undo what the agent did.

Commit frequently when working with agents. Before asking the agent to make changes, ensure your working directory is clean. After the agent finishes, review the diff carefully before committing. If the changes are wrong, you can discard them and try a different approach.

For advanced workflows, Git's `worktree` feature lets you maintain multiple working directories from the same repository, each on a different branch. This enables running multiple agents in parallel on separate tasks without conflicts. Each agent works in its own worktree, and you merge the results when ready.

Video

For a quick introduction to git worktrees, see this video.

- Learn git worktrees in under 5 minutes

9.7.6. Security and permissions

Coding agents require significant permissions to do their work. They need to read your files, write new code, and often execute commands. This power creates security risks that you should take seriously.

Be cautious about what you allow agents to access. An agent with permission to execute arbitrary shell commands could potentially damage your system, exfiltrate data, or install malicious software. Most agents have safety measures, but these are not foolproof. Never run agents with elevated privileges, and be wary of granting access to sensitive directories or credentials.

Development containers, discussed in Chapter 10, provide an effective safety layer. By running agents inside a container, you isolate them from your main system. Even if something goes wrong, the damage is contained. The container can be reset to a clean state, and your host system remains protected. For any serious agent use, especially with less-established tools, running in a devcontainer is a sensible precaution.

9.8. AI-Assisted Workflow for Research

This section presents a workflow designed to help you benefit from AI assistance while maintaining understanding and control. The principles apply regardless of which specific tools you use.

9.8.1. Start from human-written pseudocode

Before asking AI for help, write pseudocode describing what you want to accomplish. This forces you to think through the logic before delegating to AI. The pseudocode becomes both your specification for the AI and your reference for evaluating its output.

Pseudocode does not need to be syntactically correct Python. It should describe the steps of your computation in terms you understand:

```
# Calculate portfolio returns for each month
# For each month:
#   Get the weights at the start of the month
#   Get the stock returns during the month
#   Multiply each weight by its corresponding return
#   Sum to get portfolio return
# Return the list of monthly portfolio returns
```

With this pseudocode, you can ask AI to help implement specific steps. You have a clear reference for whether the implementation matches your intention.

9.8.2. Use AI to reduce syntax friction, not to invent logic

AI should help you express ideas you already have, not generate ideas for you. If you do not know what analysis to run, AI cannot tell you. If you know what analysis to run but are unsure of the Python syntax, AI can help.

This distinction keeps you in the role of researcher rather than consumer of AI output. You make the methodological decisions. AI helps you implement those decisions efficiently. The resulting code reflects your understanding because it implements your logic.

9.8.3. Run and test after every AI-assisted change

Never accumulate multiple AI-generated changes before testing. After each change, run the code and verify it behaves correctly. Check edge cases. Compare results against manual calculations or known benchmarks. This immediate feedback catches errors early when they are easy to diagnose.

If you accept several AI suggestions before testing, and something goes wrong, you do not know which change caused the problem. You may need to undo everything and start over. Testing incrementally avoids this.

> ⚠ AI-generated tests require careful review
>
> AI tools are good at writing tests, which can accelerate your workflow. However, when AI writes both the code and the tests, you lose an important check on correctness. Review AI-generated tests carefully to ensure they actually test the right behavior and cover edge cases. A test that passes is only meaningful if the test itself is correct.
> Be especially careful with coding agents. When tests fail, agents have a tendency to modify the tests to make them pass rather than fixing the underlying code. This defeats the purpose of testing entirely. Always review diffs carefully to ensure the agent fixed the function, not the test.

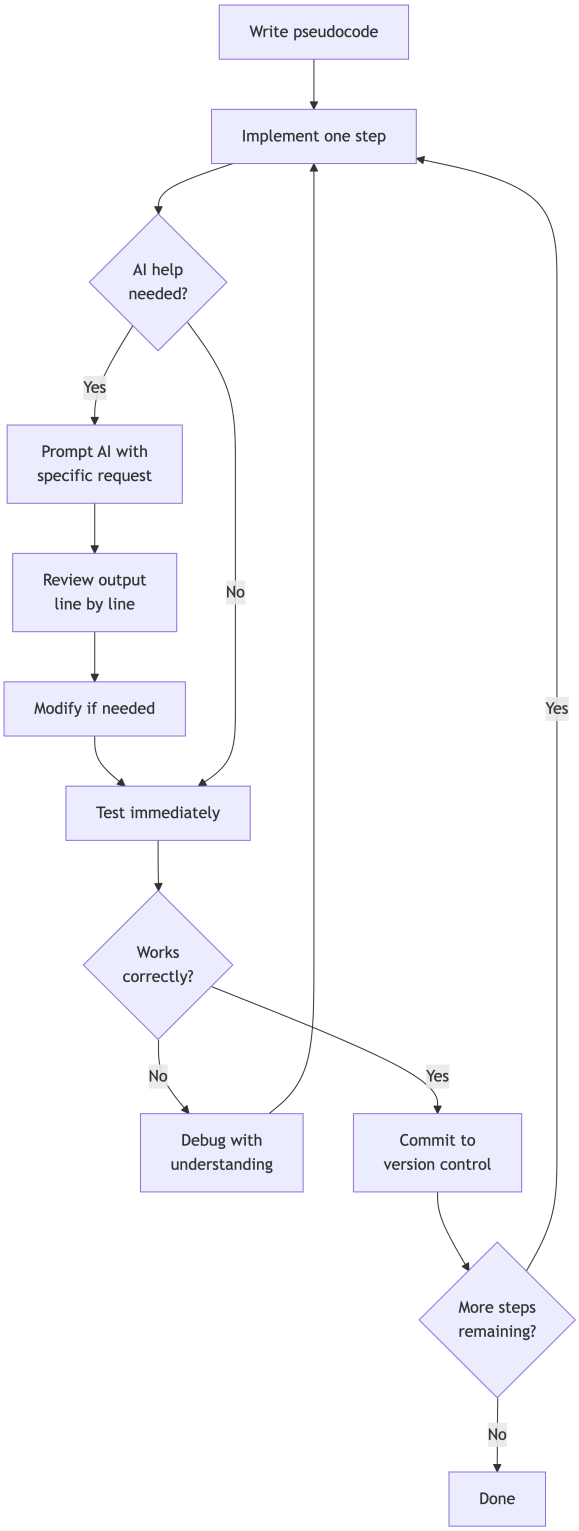


Figure 9.2.: AI-assisted workflow for research coding

9.8.4. Use version control aggressively between iterations

Commit working code to version control before making AI-assisted changes. If the changes break something, you can easily revert. This safety net encourages experimentation while protecting against mistakes.

Good commit messages document what you changed and why. When you later review your project history, you can see the evolution of your analysis. This is valuable for debugging, for collaboration, and for your own understanding. AI coding assistants are good at writing detailed commit messages that summarize the changes made. Just make sure you review the message before submitting the commit to ensure it accurately describes the work.

9.8.5. Never merge AI-generated code you cannot explain line by line

The final check is simple: can you explain every line of the code? If someone asked why a particular line exists, could you answer? If the code stopped working, would you know where to look?

If you cannot answer these questions, you should not use the code. Take time to understand it, or ask the AI to explain it, or write it yourself. The short-term cost of slower progress is worth the long-term benefit of genuine understanding.

9.9. Local and Open Models

Cloud-based AI tools are convenient but not always appropriate. This section covers alternatives that run on your own hardware, avoiding the need to send data to external servers.

Local models address several concerns that matter for research. Your prompts and code never leave your machine, eliminating concerns about data retention, training on your inputs, or security breaches at the provider. Organizations that prohibit external AI tools may permit local ones since the code stays within your controlled environment. Local inference has no per-token charges, so after the initial setup, usage costs only electricity. And local models work without internet access, which matters for air-gapped systems or unreliable connections.

Running AI models locally requires significant computing resources. Effective local models require Apple Silicon Macs or PCs with modern GPUs; older hardware often cannot run useful models at reasonable speed. Memory is usually the binding constraint, with larger models requiring more of it. On Apple computers, the system uses unified memory where RAM is shared between the CPU and GPU, so a machine with 32GB or more of RAM can run substantial models. On PCs, the relevant constraint is typically VRAM on the GPU rather than system RAM, since the model must fit in GPU memory to run efficiently. Model files themselves are large, often multiple gigabytes each, so keeping several models available requires substantial storage. Local AI also requires more technical setup than cloud services: you need to install software, download models, and configure integration with your editor.

Two popular options for running AI models locally are Ollama and LM Studio. Ollama is a command-line tool that makes running local models straightforward, handling model downloads, memory management, and inference. Many editor integrations support Ollama as a backend. LM Studio provides a graphical interface for

downloading and running local models, making it more accessible for users uncomfortable with command-line tools, and includes a built-in chat interface.

Several tools support local model backends for coding assistance. Some configurations of GitHub Copilot allow using local models instead of cloud APIs, though capabilities may be reduced. Open-source alternatives like Continue provide Copilot-like functionality with local model support. Command-line agents like Claude Code can also be configured with local model providers, though this requires additional setup.

Local models currently trail cloud models in capability. The most capable models require hardware that exceeds what most individuals own, so local models are typically smaller and less capable. Cloud providers invest heavily in tooling that local solutions cannot match, and you are responsible for updates, troubleshooting, and configuration. These trade-offs may be acceptable when privacy or institutional requirements demand local operation, but for learning purposes, cloud tools are usually more practical.

9.10. Summary: Rules You Should Actually Follow

This chapter has covered many considerations, but effective AI use comes down to a few key principles:

If you cannot read the code, you should not use the code. This is the fundamental rule. AI can help you write code faster, but you must understand what you are using. If you cannot explain it, you cannot debug it, maintain it, or defend it.

AI accelerates feedback, not understanding. The AI can produce code quickly and identify errors quickly. But understanding comes from your engagement with the code. Use AI to speed up the feedback loop, not to skip the learning.

In research and industry, credibility beats convenience. Your reputation depends on the quality and correctness of your work. AI-generated code that contains errors damages your credibility. Code you understand and can explain builds it.

Start with what you know. Write pseudocode before asking for help. Use AI to implement your ideas, not to have ideas for you. The research decisions are yours; AI helps with the implementation.

Test relentlessly. Every AI-generated change should be followed by testing. Catch errors early, when they are easy to diagnose. Never assume that code is correct because AI produced it.

Maintain version control. Commit working code before making changes. Create a safety net that lets you experiment confidently and recover from mistakes.

Treat AI responses as hypotheses. The AI is proposing solutions, not providing answers. Your job is to evaluate those proposals critically, accept what works, and reject what does not.

These principles will serve you regardless of how AI tools evolve. The specific tools will change. The specific capabilities will improve. But the fundamental dynamic remains: AI is a tool that amplifies your capabilities, for better or worse. Used wisely, it makes you more effective. Used carelessly, it prevents you from becoming effective at all.

The choice is yours.

10. Stay Safe with Devcontainers

In empirical finance research, the tools and methodologies we employ play a crucial role in ensuring the integrity and reproducibility of our findings. One such powerful tool is containerization, which allows you to encapsulate your code and its dependencies into a standardized unit. Development containers, or devcontainers, provide a convenient way to create isolated environments for your data analysis tasks, ensuring that your code runs consistently across different systems.

Containers are essentially lightweight, portable environments that package up code and all its dependencies, ensuring that the software runs consistently across different computing environments. This consistency is particularly valuable in research settings where the reproducibility of results is paramount. By containerizing their code, researchers can avoid the "it works on my machine" problem, ensuring that their analyses can be replicated by others, regardless of the underlying system configurations. You might be familiar with the use of tools like uv or pyenv for managing Python environments to separate dependencies for different projects. Containers take this concept a step further by encapsulating the entire environment, including the operating system, runtime, libraries, and configurations. This ensures that the code runs identically on any machine, making it easier for researchers to share their work and for others to verify their findings.

This encapsulation also provides an added level of safety and security that is essential in research settings. By isolating the code and its dependencies from the host system, containers protect the environment from potential security vulnerabilities or malware disguised as python librairies. This isolation ensures that the code runs in a controlled environment, reducing the risk of unintended interactions with the host system. For additional security, you can use third-party services like Snyk to scan your dependencies for known vulnerabilities, or use curated package repositories like Anaconda or PyX that vet packages before making them available.

Finally, containers facilitate collaboration among researchers. When a project is containerized, collaborators can easily set up their environment by simply running the container, eliminating the often cumbersome process of manually installing and configuring dependencies. This ease of setup promotes a more efficient workflow and reduces the likelihood of errors, making collaborative research more streamlined and productive. Even if you work solo, containers improve the resiliency of your research workflows, making it easy to recover from a broken or stolen computer by reinstalling your project on a new computer without worrying about compatibility issues. You can even run your code in the cloud with services like GitHub Codespaces, ensuring that your analyses are not tied to a specific machine or operating system.

In this chapter, I provide a step-by-step guide on setting up devcontainers in VS Code, with a focus on supporting Python uv and mounting local directories for file storage.

All the code and configurations used in this tutorial are available in the GitHub repository.

Video

This chapter is also available as a video tutorial on YouTube (note that the video uses poetry instead of uv).

- Use Dev Containers in VS Code for Safe and Replicable Data Analysis in Python

10.1. Setting Up Devcontainers in VS Code

Development containers are a feature in VS Code that leverages Docker[1] to create and manage containers specifically for development purposes. This allows developers and researchers to work within a consistent environment, which is crucial for complex data analysis tasks. Setting up devcontainers in VS Code is straightforward, but there are a few prerequisites you need to have in place: Docker, Visual Studio Code, and the Dev Containers extension. If you're on macOS, you can install Docker using Homebrew:

```
brew install --cask docker
```

To begin, you'll need to create a `.devcontainer` folder in your project directory. Inside this folder, you should create a `devcontainer.json` file, which will define the configuration for your development container. This file specifies the base image for the container, any additional tools or libraries that need to be installed, and other settings related to the development environment. By configuring this file, you can tailor the container to meet the specific needs of your data analysis tasks. This can be done using the Dev Containers extension in VS Code, which provides a user-friendly interface for creating and managing devcontainers. To get started, simply invoke `Dev Containers: Add Development Container Configuration Files...` from the command palette in VS Code and follow the prompts to create your `devcontainer.json` file.

This will prompt you to select a base image for your container, configure any additional tools or libraries, and set up other environment settings. As default options, I like to use the following:

- Setup location: in workspace
- Base image: Python 3.12 image from Microsoft
- Features: Any additional tools you need (e.g., Quarto, GitHub CLI)

Python version

While the current latest version is 3.14, a the time of writing the latest Python base image from Microsoft is 3.12. This is not much of an issue because uv will install its own Python version when creating your environment based on your specification in `pyproject.toml`.

Once you have configured your `devcontainer.json` file, you should get a prompt to reopen the project in the container. This will build the container based on the specified configuration and open your project within

[1] Other alternatives such as Podman and Colima can be used, but the official documentation is centered around Docker. See the VS Code documentation for supported alternatives.

the containerized environment. You can verify that the container is running by checking the status bar in VS Code, which should indicate that you are working in a containerized environment.

You can always rebuild the container by invoking the `Dev Containers: Rebuild and Reopen in Container` command from the command palette. This will recreate the container based on the latest configuration settings, ensuring that your development environment is up-to-date and consistent with your project requirements.

Here is what a simple `devcontainer.json` file (minus the comments) looks like:

```
{
  "name": "Python 3",
  "image": "mcr.microsoft.com/devcontainers/python:1-3.12-bullseye",
  "postCreateCommand": "./.devcontainer/postCreateCommand.sh"
}
```

This configuration references a shell script that will be executed after the container is created. This script handles installing uv and setting up the project dependencies (more on this later).

10.2. Images

Images are a crucial aspect of containerization, as they define the base environment for your development container. When setting up a devcontainer in VS Code, you can choose from a variety of pre-built images that provide different programming languages, tools, and libraries. These images serve as the foundation for your development environment, ensuring that the necessary dependencies are available for your data analysis tasks. By tying your devcontainer to a specific image, you can guarantee that your code runs consistently across different systems, making it easier to share and replicate your work.

Note: Most images are based on Linux distributions, so you may need to adjust your code or configurations if you are used to working on Windows or macOS. However, the differences are usually minimal and can be easily managed within the container.

Note 2: Most images specify the environment (i.e. Linux version, Python version, etc.), but not the architecture, which makes them compatible with most systems. For example, most PCs and older Macs use Intel or AMD CPUs with the `x86_64` architecture, while newer Macs with Apple Silicon have the `arm64` architecture. The images are usually compatible with both architectures, but that means that the environment will not be 100% identical if you run the container on different architectures. This is usually not a problem for data analysis tasks, but it's something to keep in mind if you are having issues with your code running differently on different systems.

10.3. Features

Features are additional tools or libraries that can be installed in your development container to enhance its functionality. These features can include language-specific tools, package managers, or development environments that are tailored to your project requirements. By specifying features in your `devcontainer.json`

file, you can extend the capabilities of your container and ensure that it is well-suited for your data analysis tasks. You can find a list of available features at containers.dev/features. For example, you can add Quarto support with the `ghcr.io/rocker-org/devcontainer-features/quarto-cli:1` feature, or add the GitHub CLI with `ghcr.io/devcontainers/features/github-cli:1`.

10.4. uv

uv is an extremely fast Python package and project manager written in Rust, designed to replace tools like pip, pip-tools, pipx, poetry, and more. It simplifies the process of creating, managing, and sharing Python projects by providing a unified interface for dependency management, packaging, and virtual environment handling. uv uses a `pyproject.toml` file to define project dependencies, scripts, and configurations, making it easy to manage project settings and requirements. By integrating uv into your devcontainer setup, you can streamline your development workflow and ensure that your Python projects are well-organized and reproducible.

For example here is a `pyproject.toml` file that defines the dependencies for a Python project:

```
[project]
name = "my-project"
version = "0.1.0"
description = "My research project"
readme = "README.md"
requires-python = ">=3.13,<4.0"
dependencies = [
    "pandas>=2.2.2",
    "numpy>=2.0.0",
]

[dependency-groups]
dev = [
    "pytest>=7.2.0",
    "ruff>=0.11.5",
]
```

With uv, project dependencies are defined in the standard `[project]` section following PEP 621, rather than in a tool-specific section. This makes your `pyproject.toml` file more portable across different Python tools. You can also define development dependencies in a separate `[dependency-groups]` section, which allows you to install them only when needed.

To set up uv in your devcontainer, create a `postCreateCommand.sh` script in your `.devcontainer` folder:

```
#!/usr/bin/env bash

# Install uv
```

```
curl -LsSf https://astral.sh/uv/install.sh | sh

# Install project dependencies
uv sync
```

This script installs uv and then runs `uv sync` to install all project dependencies into a virtual environment (`.venv`). The `uv sync` command reads your `pyproject.toml` file and creates a reproducible environment with all specified dependencies.

To use this script, reference it in your `devcontainer.json` file:

```
  "postCreateCommand": "./.devcontainer/postCreateCommand.sh"
```

Make sure the script is executable by running `chmod +x .devcontainer/postCreateCommand.sh` before committing it to your repository.

10.5. Mounting Local Directories

By default, the development container is isolated from the host system, except for the workspace directory, which is mounted into the container. This ensures that your project files are accessible within the container, allowing you to work on your code seamlessly. However, there are cases where you may need to access files or directories outside the workspace, such as large data files. To achieve this, you can mount local directories into the development container, making them available within the container environment.

To mount a local directory, you can add the `mounts` property to your `devcontainer.json` file, specifying the `source` path on the host and the `target` path in the container.

Here is an example configuration for mounting a local directory:

```
  "mounts": ["source=/path/to/local/directory,target=/workspace/data,type=bind,consistency=cached"]
```

This setup ensures that the directory `/path/to/local/directory` on your host machine is accessible within the container at `/workspace/data`. This approach provides the flexibility to work with local files while benefiting from the isolated environment of the container, except for the mounted directories.

Note: This mounting process limits the flexibility of the container, as the source directory must be available on the host system. If you are working on a shared project or need to access files from different locations, you may need to consider alternative approaches, such as using a networked file system or cloud storage. As far as I know, there is no way to specify the source directory using an environment variable, so each collaborator would need to update the `devcontainer.json` file with their local path.

10.6. VS Code Extensions

You can also use the devcontainer.json file to configure default VS Code seettings and install VS Code extensions in your development container. When working with devcontainers, you will notice that not all the extensions you have installed on your host system are available in the container environment. Specifically, extensions that are mostly used for the host system (think UI), such as themes or language packs, will still be available. However, extensions that require access to the container environment, such as language servers or debuggers, will need to be installed. For example, the python image will install the Python extension, but you may need to install additional extensions for specific tasks, such as the Jupyter extension for working with Jupyter notebooks or with the interactive window. It can also be useful to make sure that all collaborators use the same formatting tools, such as Ruff.

To address this, you can specify the extensions and settings you want to install in the `devcontainer.json` file. For example, this will install Jupyter and Ruff, configure Ruff as the default formatter, and point VS Code to the virtual environment created by uv:

```
    "customizations": {
        "vscode": {
            "extensions": [
                "ms-toolsai.jupyter",
                "charliermarsh.ruff"
            ],
            "settings": {
                "[python]": {
                    "editor.defaultFormatter": "charliermarsh.ruff",
                    "editor.codeActionsOnSave": {
                        "source.fixAll": "explicit",
                        "source.organizeImports": "explicit"
                    }
                },
                "python.defaultInterpreterPath": "${workspaceFolder}/.venv/bin/python",
                "editor.formatOnSave": true
            }
        }
    }
```

Note the `python.defaultInterpreterPath` setting, which tells VS Code to use the Python interpreter in the `.venv` virtual environment created by `uv sync`. This ensures that VS Code uses the correct Python environment with all your project dependencies.

You can find the extension IDs by looking at the extension in VS Code, then clicking on the gear icon and selecting “Copy Extension ID”.

10.7. Limitations

Devcontainers are a powerful tool for creating isolated development environments, but they do have some limitations. One of the main drawbacks is the overhead associated with running containers, which can slow down the development process, especially for large projects or resource-intensive tasks. It's not an issue that I have found to be significant, but it's something to keep in mind if you are working on a particularly demanding project or have limited system resources.

Additionally, the isolation provided by containers can make it challenging to use some resources from the host system, such as GPUs or hardware peripherals. While it is possible to pass through devices to the container using Docker, this process can be complex and may not be suitable for all use cases. For example, on Apple Silicon Macs, even if the Linux container could access the GPU, there are no Linux drivers available for the GPU, so it would not be able to use it.

Part II.

Working with Data

This part covers the essential tools and techniques for working with data in Python. We introduce three powerful libraries—pandas, Polars, and DuckDB—and demonstrate how to use them for common data manipulation tasks in empirical finance.

The chapters in this part progressively build your data manipulation skills:

- **Introduction to DataFrames**: A high-level overview of pandas, Polars, and DuckDB
- **Data Input and File Formats**: Loading data from CSV, Parquet, and Excel files
- **Data Cleaning**: Handling duplicates, missing data, and validation
- **Data Structuring and Aggregation**: Grouping, aggregation, and working with keys
- **Reshaping Data**: Converting between long and wide formats
- **Joins and Merges**: Combining datasets correctly

Most concepts are demonstrated with examples in pandas, Polars, and DuckDB, allowing you to choose the right tool for your needs and understand how to translate between them.

11. Introduction to DataFrames

11.1. Overview

In empirical finance, we work mostly with structured data: stock prices organized by date and ticker, financial statements arranged by company and quarter, portfolio returns indexed by time and strategy. The DataFrame abstraction is the fundamental tool for handling this kind of tabular data in Python.

This chapter introduces three major DataFrame libraries in the Python ecosystem: pandas (the established standard), Polars (a modern, high-performance alternative), and DuckDB (which brings SQL to DataFrames). Understanding when and how to use each tool will significantly impact your productivity and the performance of your empirical work.

11.2. The DataFrame Abstraction

11.2.1. What is a DataFrame?

A DataFrame is a two-dimensional labeled data structure with columns of potentially different types. Think of it as a spreadsheet in code, a database table with an API, or a generalization of a matrix that allows mixed types.

But a DataFrame is more than just a container. It's an abstraction that encapsulates:

1. **Data storage**: Efficient memory layout for columnar data
2. **Indexing**: Labels for rows and columns enabling semantic access
3. **Operations**: A rich API for transformations, aggregations, and joins
4. **Type system**: Handling heterogeneous data types within a single structure

11.2.2. Why DataFrames Matter in Finance

In empirical finance research, DataFrames solve several critical problems:

Data Alignment: When you merge stock returns with company characteristics, the DataFrame automatically handles date and identifier alignment. No manual loops checking if dates match.

Missing Data: Financial datasets are riddled with missing values. DataFrames provide principled methods for handling missing values that respect the underlying data structure.

Performance: Modern DataFrame libraries use columnar storage and vectorized operations, making them orders of magnitude faster than row-by-row processing.

Expressiveness: Complex operations like "calculate rolling volatility by stock, then merge with quarterly earnings" can be expressed in a few lines of readable code.

> **The Columnar Advantage**
>
> DataFrames store data column-by-column rather than row-by-row. This means:
>
> - Operations on a single column (like calculating returns) are blazingly fast
> - Memory access patterns are optimized for modern CPUs
> - Compression works better when similar data is stored together
> - Aggregations can skip irrelevant columns entirely
>
> This is why DataFrames can process millions of rows efficiently while naive Python loops struggle.

11.2.3. Core Concepts

All DataFrame libraries share several fundamental concepts:

Columns and Rows: The basic 2D structure. Columns typically represent variables (price, volume, return), while rows represent observations (a specific date, company, or transaction).

Index: A special column (or set of columns) that labels rows. In finance, this is often a date or a combination of date and identifier.

Schema: The data type of each column. Strongly-typed schemas enable optimization and catch errors early.

Operations: Transformations that produce new DataFrames. These include filtering, selecting, aggregating, and joining.

> **Beyond two dimensions**
>
> If you need more than two dimensions for your data, the xarray library in Python can be helpful. It provides labeled, multi-dimensional arrays that extend the DataFrame concept to higher dimensions. While less widespread in finance than pandas or Polars, xarray is useful for working with panel data or other multi-dimensional structures.

11.3. pandas Overview

pandas is the original DataFrame library for Python and remains the most widely used. Created by Wes McKinney in 2008 for financial data analysis, it has become the de facto standard for data manipulation in Python. While pandas is no longer my go-to library for data processing (I now mostly use Polars for that),

pandas remains the standard and is unavoidable because many additional libraries for plotting and statistics rely on pandas DataFrames. If you only learn one DataFrame library, it should be this one.

11.3.1. Basic Operations

Let's start with creating and manipulating a simple DataFrame:

```
import pandas as pd                                                    ①

df = pd.DataFrame({                                                    ②
    'ticker': ['AAPL', 'AAPL', 'MSFT', 'MSFT', 'GOOG', 'GOOG'],
    'date': pd.date_range('2024-01-01', periods=6, freq='D')[:6],      ③
    'price': [150.5, 152.3, 380.2, 378.9, 140.1, 142.5],
    'volume': [1000000, 1050000, 800000, 820000, 950000, 980000]
})

df                                                                     ④
```

① The standard convention is to import pandas as `pd`, allowing you to reference it with this short alias throughout your code.
② Create a DataFrame from a dictionary where keys become column names.
③ `pd.date_range()` generates a sequence of dates with the specified frequency (`'D'` for daily).
④ In Jupyter notebooks and Quarto, simply placing a DataFrame at the end of a cell displays it with nice formatting.

	ticker	date	price	volume
0	AAPL	2024-01-01	150.5	1000000
1	AAPL	2024-01-02	152.3	1050000
2	MSFT	2024-01-03	380.2	800000
3	MSFT	2024-01-04	378.9	820000
4	GOOG	2024-01-05	140.1	950000
5	GOOG	2024-01-06	142.5	980000

Key characteristics of this DataFrame:

- **Index**: Automatically created as integers (0-5)
- **Columns**: Four columns with mixed types (object, datetime64, float64, int64)
- **Shape**: 6 rows × 4 columns

11.3.2. Indexing and Selection

pandas provides multiple ways to select data, which can be confusing at first but powerful once mastered. pandas is somewhat unique in this regard: Polars and DuckDB don't rely on indexing the same way pandas does—they basically treat all columns the same. In pandas, the index column (or columns, if there's a multi-index) has a special status that affects how the DataFrame behaves.

Column Selection

```
# Select a single column (returns a Series)
prices = df['price']
prices
```

```
0    150.5
1    152.3
2    380.2
3    378.9
4    140.1
5    142.5
Name: price, dtype: float64
```

```
# Select multiple columns (returns a DataFrame)
subset = df[['ticker', 'price']]
subset
```

	ticker	price
0	AAPL	150.5
1	AAPL	152.3
2	MSFT	380.2
3	MSFT	378.9
4	GOOG	140.1
5	GOOG	142.5

Notice that the double brackets in the second example aren't a different operator—we're still using the same square bracket indexing, but instead of passing a single column name, we're passing a list of column names. When you pass a list, pandas returns a DataFrame; when you pass a single column name, it returns a Series.

Row Selection with .loc and .iloc

pandas distinguishes between label-based indexing (`.loc`) and position-based indexing (`.iloc`):

```
# Select rows 0-2 by position
df.iloc[0:3]
```

	ticker	date	price	volume
0	AAPL	2024-01-01	150.5	1000000
1	AAPL	2024-01-02	152.3	1050000
2	MSFT	2024-01-03	380.2	800000

```
# If we set an index, we can use .loc with labels
df_indexed = df.set_index('date')
df_indexed.loc['2024-01-01':'2024-01-03']
```

	ticker	price	volume
date			
2024-01-01	AAPL	150.5	1000000
2024-01-02	AAPL	152.3	1050000
2024-01-03	MSFT	380.2	800000

.loc vs .iloc vs []

- Use `.loc[row_labels, column_labels]` for label-based indexing
- Use `.iloc[row_positions, column_positions]` for integer position-based indexing
- Use `[]` for simple column selection or boolean masking

The distinction prevents ambiguity: `df[0]` is an error (is it column 0 or row 0?), but `df.iloc[0]` and `df['column_0']` are clear.

Boolean Indexing

Filtering data is a core operation in empirical finance:

```
# Select rows where price > 150
high_price = df[df['price'] > 150]
high_price
```

	ticker	date	price	volume
0	AAPL	2024-01-01	150.5	1000000
1	AAPL	2024-01-02	152.3	1050000
2	MSFT	2024-01-03	380.2	800000
3	MSFT	2024-01-04	378.9	820000

The expression inside the square brackets (`df['price'] > 150`) is itself a Series of Boolean values. Let's look at what it produces:

```
# The condition produces a Boolean Series
df['price'] > 150
```

```
0     True
1     True
2     True
3     True
4    False
5    False
Name: price, dtype: bool
```

The rows that get selected are those where the Boolean value is `True`.

```
# Multiple conditions: AAPL stocks with price > 150
aapl_high = df[(df['ticker'] == 'AAPL') & (df['price'] > 150)]
aapl_high
```

	ticker	date	price	volume
0	AAPL	2024-01-01	150.5	1000000
1	AAPL	2024-01-02	152.3	1050000

```
# Using query method for more readable syntax
result = df.query('ticker == "AAPL" and price > 150')
result
```

	ticker	date	price	volume
0	AAPL	2024-01-01	150.5	1000000
1	AAPL	2024-01-02	152.3	1050000

11.3.3. Creating New Columns

A common operation is creating new columns from existing ones. When you perform arithmetic operations between columns, pandas aligns them by index and applies the operation element by element:

```
# Create a new column by multiplying two existing columns
df['dollar_volume'] = df['price'] * df['volume']                    ①
df
```

① Element-wise multiplication: each row's price is multiplied by that row's volume.

	ticker	date	price	volume	dollar_volume
0	AAPL	2024-01-01	150.5	1000000	150500000.0
1	AAPL	2024-01-02	152.3	1050000	159915000.0
2	MSFT	2024-01-03	380.2	800000	304160000.0
3	MSFT	2024-01-04	378.9	820000	310698000.0
4	GOOG	2024-01-05	140.1	950000	133095000.0
5	GOOG	2024-01-06	142.5	980000	139650000.0

When you multiply or add a scalar, it applies to all elements in the column:

```
# Multiply a column by a scalar
df['price_cents'] = df['price'] * 100                               ①
df
```

① The scalar 100 is applied to every element in the `price` column.

	ticker	date	price	volume	dollar_volume	price_cents
0	AAPL	2024-01-01	150.5	1000000	150500000.0	15050.0
1	AAPL	2024-01-02	152.3	1050000	159915000.0	15230.0
2	MSFT	2024-01-03	380.2	800000	304160000.0	38020.0
3	MSFT	2024-01-04	378.9	820000	310698000.0	37890.0
4	GOOG	2024-01-05	140.1	950000	133095000.0	14010.0
5	GOOG	2024-01-06	142.5	980000	139650000.0	14250.0

You can also combine columns with different operations:

```
# Combining columns with arithmetic
df['avg_price_per_share'] = df['dollar_volume'] / df['volume']
df[['ticker', 'price', 'avg_price_per_share']]
```

	ticker	price	avg_price_per_share
0	AAPL	150.5	150.5
1	AAPL	152.3	152.3
2	MSFT	380.2	380.2
3	MSFT	378.9	378.9
4	GOOG	140.1	140.1
5	GOOG	142.5	142.5

i Index alignment in operations

When you perform operations between two columns (or two DataFrames), pandas automatically aligns them by their index. This is powerful but can also be a source of subtle bugs if your indices don't match as expected. We'll cover sorting, grouping, and aggregation operations in a later chapter.

11.3.4. Strengths and Limitations

Strengths:

- Mature ecosystem with extensive documentation
- Rich functionality covering nearly every data manipulation need
- Excellent integration with other libraries (scikit-learn, statsmodels, matplotlib)
- Flexible indexing system enabling time-series analysis
- Wide adoption means abundant examples, Stack Overflow answers, and AI coding assistance

Limitations:

- Performance can degrade with multi-gigabyte datasets, depending on available RAM
- Memory usage is higher than necessary due to row-based implementation details
- API inconsistencies accumulated over 15+ years of development
- Single-threaded execution for most operations
- String operations are particularly slow

11.4. Polars Overview

Polars is a modern DataFrame library written in Rust, designed from the ground up for performance. It addresses many of pandas' limitations while maintaining a familiar API for common operations.

Polars differs from pandas in several important ways:

Lazy evaluation allows you to write your data processing code without immediately executing it. Polars builds a query plan and defers execution until you explicitly collect the results. This enables the query optimizer to analyze the entire pipeline and find efficiencies—such as eliminating unnecessary computations, reordering operations, or pushing filters earlier in the pipeline.

Parallel execution means Polars can automatically use multiple CPU cores for operations that support it. Unlike pandas, which is largely single-threaded, Polars can process different parts of your data concurrently, leading to significant speedups on modern multi-core machines.

Apache Arrow memory representation is an efficient columnar format that Polars uses internally. Arrow improves interoperability with other libraries that support the format (pandas includes Arrow as an optional backend), and generally makes operations more efficient. Importantly, Arrow explicitly supports missing values as a distinct type rather than relying on NaN, which is important for accurate calculations and proper handling of missing data. Arrow also enables efficient reading and writing to the optimized Parquet file format.

The expression API provides a composable syntax for data transformations that is not identical to pandas. Users coming from the R tidyverse may find it feels more familiar than pandas' approach. Expressions can be combined and optimized together, leading to more efficient execution.

11.4.1. Basic Operations

```
import polars as pl                                                    ①

df_pl = pl.DataFrame({
    'ticker': ['AAPL', 'AAPL', 'MSFT', 'MSFT', 'GOOG', 'GOOG'],
    'date': pd.date_range('2024-01-01', periods=6, freq='D'),
    'price': [150.5, 152.3, 380.2, 378.9, 140.1, 142.5],
    'volume': [1000000, 1050000, 800000, 820000, 950000, 980000]
})

df_pl
```

① The standard convention is to import Polars as `pl`.

ticker str	date datetime[ns]	price f64	volume i64
"AAPL"	2024-01-01 00:00:00	150.5	1000000
"AAPL"	2024-01-02 00:00:00	152.3	1050000
"MSFT"	2024-01-03 00:00:00	380.2	800000
"MSFT"	2024-01-04 00:00:00	378.9	820000
"GOOG"	2024-01-05 00:00:00	140.1	950000
"GOOG"	2024-01-06 00:00:00	142.5	980000

11.4.2. The Expression API

Polars introduces an expression-based API that is more composable than pandas:

```
# Select and transform columns
result = df_pl.select([
    pl.col('ticker'),                                                    ①
    pl.col('price'),
    (pl.col('price') * pl.col('volume')).alias('dollar_volume')          ②
])
result
```

① `pl.col()` references a column by name.
② `.alias()` names the resulting column.

ticker str	price f64	dollar_volume f64
”AAPL”	150.5	1.505e8
”AAPL”	152.3	1.59915e8
”MSFT”	380.2	3.0416e8
”MSFT”	378.9	3.10698e8
”GOOG”	140.1	1.33095e8
”GOOG”	142.5	1.3965e8

```
# Filtering with expressions
high_price_pl = df_pl.filter(pl.col('price') > 150)
high_price_pl
```

ticker str	date datetime[ns]	price f64	volume i64
”AAPL”	2024-01-01 00:00:00	150.5	1000000
”AAPL”	2024-01-02 00:00:00	152.3	1050000
”MSFT”	2024-01-03 00:00:00	380.2	800000
”MSFT”	2024-01-04 00:00:00	378.9	820000

```
# Chaining operations
result = (
    df_pl
    .filter(pl.col('ticker') == 'AAPL')
    .with_columns((pl.col('price') * 1.1).alias('price_plus_10pct'))     ①
    .select(['ticker', 'date', 'price', 'price_plus_10pct'])
)
result
```

① `.with_columns()` adds or replaces columns in the DataFrame.

ticker str	date datetime[ns]	price f64	price_plus_10pct f64
”AAPL”	2024-01-01 00:00:00	150.5	165.55
”AAPL”	2024-01-02 00:00:00	152.3	167.53

11.4.3. Lazy Evaluation

Polars’ lazy API builds a query plan and optimizes it before execution:

```
lazy_df = df_pl.lazy()                                                    ①

# Build a query (nothing executes yet)
lazy_query = (
    lazy_df
    .filter(pl.col('volume') > 850000)
    .group_by('ticker')
    .agg([
        pl.col('price').mean().alias('avg_price'),
        pl.col('volume').sum().alias('total_volume')
    ])
    .sort('avg_price', descending=True)
)

result = lazy_query.collect()                                             ②
result
```

① `.lazy()` converts an eager DataFrame to a lazy frame.
② `.collect()` executes the optimized query plan and returns the result.

ticker str	avg_price f64	total_volume i64
”AAPL”	151.4	2050000
”GOOG”	141.3	1930000

! When to Use Lazy Evaluation

Lazy evaluation shines when:

- You have a complex chain of operations
- Your data is large enough that optimization matters
- You’re reading from files (Polars can push down filters and projections)

For small datasets and simple operations, the overhead of optimization may not be worth it. Use eager

mode (regular DataFrames) for interactive exploration.

11.4.4. Grouping and Aggregation

```
# Group by with multiple aggregations
summary_pl = df_pl.group_by('ticker').agg([
    pl.col('price').mean().alias('avg_price'),
    pl.col('price').std().alias('std_price'),
    pl.col('volume').sum().alias('total_volume')
])
summary_pl
```

ticker str	avg_price f64	std_price f64	total_volume i64
”GOOG”	141.3	1.697056	1930000
”MSFT”	379.55	0.919239	1620000
”AAPL”	151.4	1.272792	2050000

```
# Window functions for returns calculation
df_pl_sorted = df_pl.sort(['ticker', 'date'])
df_with_returns = df_pl_sorted.with_columns(
    pl.col('price').pct_change().over('ticker').alias('return')
)
df_with_returns
```

ticker str	date datetime[ns]	price f64	volume i64	return f64
”AAPL”	2024-01-01 00:00:00	150.5	1000000	null
”AAPL”	2024-01-02 00:00:00	152.3	1050000	0.01196
”GOOG”	2024-01-05 00:00:00	140.1	950000	null
”GOOG”	2024-01-06 00:00:00	142.5	980000	0.017131
”MSFT”	2024-01-03 00:00:00	380.2	800000	null
”MSFT”	2024-01-04 00:00:00	378.9	820000	-0.003419

11.4.5. Strengths and Limitations

Strengths:

- Significantly faster than pandas, especially for large datasets

- Lower memory usage thanks to Arrow format
- Automatic parallelization
- Lazy evaluation enables query optimization
- More consistent API design
- Excellent handling of string and categorical data

Limitations:

- Smaller ecosystem (fewer third-party integrations)
- Less mature documentation and community resources
- Some advanced pandas features not yet implemented
- Different mental model requires learning curve
- Index-less design may require adaptation for time-series workflows

11.5. DuckDB and SQL for DataFrames

DuckDB is an in-process SQL database optimized for analytical queries. While not strictly a DataFrame library, it provides a powerful alternative interface for DataFrame operations through SQL.

While this book focuses mostly on Python, SQL is also a very important language for financial data analytics because it is the language most databases use. Many third-party databases can be accessed through SQL from Python and return DataFrames, but DuckDB presents a nice alternative that lets you write SQL directly within your Python environment. DuckDB doesn't need to run inside Python—it can also operate as a standalone tool—but using it within Python makes it easy to take partial results and transfer them to pandas or Polars for further processing. Like Polars, DuckDB uses Arrow for its memory representation, enabling efficient data exchange between tools.

11.5.1. Why SQL for DataFrames?

SQL has several advantages:

1. **Declarative**: Describe what you want, not how to get it
2. **Optimized**: Query optimizers can find efficient execution plans
3. **Familiar**: Many finance professionals already know SQL
4. **Standard**: SQL skills transfer across tools and platforms

11.5.2. Basic Usage

```
import duckdb

result = duckdb.sql("""
    SELECT ticker,
           AVG(price) as avg_price,
```

```
           SUM(volume) as total_volume
    FROM df
    GROUP BY ticker
    ORDER BY avg_price DESC
""").df()                                                                    ①

result
```

① `.df()` converts the DuckDB result to a pandas DataFrame.

	ticker	avg_price	total_volume
0	MSFT	379.55	1620000.0
1	AAPL	151.40	2050000.0
2	GOOG	141.30	1930000.0

Notice that we didn't need to import the DataFrame into DuckDB explicitly. DuckDB automatically detects pandas DataFrames in the scope and makes them queryable.

11.5.3. Advanced SQL Operations

```
# Window functions for returns
returns_sql = duckdb.sql("""
    SELECT ticker,
           date,
           price,
           price / LAG(price) OVER (PARTITION BY ticker ORDER BY date) - 1 as return
    FROM df
    ORDER BY ticker, date
""").df()

returns_sql
```

	ticker	date	price	return
0	AAPL	2024-01-01	150.5	NaN
1	AAPL	2024-01-02	152.3	0.011960
2	GOOG	2024-01-05	140.1	NaN
3	GOOG	2024-01-06	142.5	0.017131
4	MSFT	2024-01-03	380.2	NaN
5	MSFT	2024-01-04	378.9	-0.003419

```
# Complex filtering and aggregation
analysis = duckdb.sql("""
    WITH high_volume AS (
        SELECT *
        FROM df
        WHERE volume > 900000
    )
    SELECT
        ticker,
        COUNT(*) as n_high_volume_days,
        AVG(price) as avg_price_on_high_volume
    FROM high_volume
    GROUP BY ticker
""").df()

analysis
```

	ticker	n_high_volume_days	avg_price_on_high_volume
0	GOOG	2	141.3
1	AAPL	2	151.4

11.5.4. Integration with Multiple Sources

DuckDB can query multiple DataFrame types simultaneously:

```
# Query both pandas and Polars DataFrames in one query
combined = duckdb.sql("""
    SELECT
        p.ticker,
        p.avg_price as pandas_avg,
        pl.avg_price as polars_avg
    FROM (SELECT ticker, AVG(price) as avg_price FROM df GROUP BY ticker) p
    JOIN (SELECT ticker, AVG(price) as avg_price FROM df_pl GROUP BY ticker) pl
        ON p.ticker = pl.ticker
""").df()

combined
```

	ticker	pandas_avg	polars_avg
0	GOOG	141.30	141.30
1	MSFT	379.55	379.55

	ticker	pandas_avg	polars_avg
2	AAPL	151.40	151.40

11.5.5. DuckDB Relations API

For a more programmatic interface, DuckDB provides a relational API:

```
rel = duckdb.sql("SELECT * FROM df")                                    ①

# Chain operations
result = (
    rel
    .filter("volume > 900000")
    .aggregate("ticker, AVG(price) as avg_price")
    .order("avg_price DESC")
)

result.df()
```

① Create a relation from a DataFrame that can be queried programmatically.

	ticker	avg_price
0	AAPL	151.4
1	GOOG	141.3

11.5.6. Strengths and Limitations

Strengths:

- Extremely fast for analytical queries
- Optimized query plans can outperform hand-written DataFrame code
- SQL is a standard, transferable skill
- Seamless integration with pandas and Polars
- Can query files directly without loading into memory
- Excellent for joins and complex aggregations

Limitations:

- SQL syntax can be verbose for simple operations
- Less suitable for row-wise operations or complex custom logic
- Debugging SQL queries can be harder than Python code
- Some operations are more natural in a DataFrame API
- Embedded database means limited concurrency

11.6. Choosing Between pandas, Polars, and DuckDB

The right tool depends on your specific use case. Here's a decision framework:

11.6.1. When to Use pandas

Choose pandas when:

- You're working with small to medium datasets (<1GB in memory)
- You need integration with scikit-learn, statsmodels, or other pandas-centric libraries
- You're learning and want maximum community support
- Your code needs to be maintained by others who know pandas
- You need pandas' advanced time-series functionality (business day calendars, time zone handling)

Example use case: Exploratory analysis of a few thousand stocks with daily data for factor model estimation.

11.6.2. When to Use Polars

Choose Polars when:

- You're working with large datasets (>1GB)
- Performance is critical (e.g., production pipelines, real-time systems)
- You want cleaner, more maintainable code through the expression API
- You need parallel processing without explicit threading code
- String manipulation performance matters

Example use case: Processing tick-by-tick trade data for thousands of stocks to compute intraday liquidity measures.

11.6.3. When to Use DuckDB

Choose DuckDB when:

- You're comfortable with SQL and prefer declarative queries
- You need to join multiple large datasets
- Your data lives in files (CSV, Parquet) and you want to query without loading
- You need maximum analytical query performance
- You're aggregating or filtering large datasets

Example use case: Joining quarterly earnings data with daily stock returns across thousands of companies and years.

11.6.4. Hybrid Approaches

You don't need to choose just one. Modern workflows often combine tools:

```
# Read large file with DuckDB
duckdb_result = duckdb.sql("""
    SELECT ticker, date, price, volume
    FROM 'large_dataset.parquet'
    WHERE date >= '2020-01-01'
    AND ticker IN ('AAPL', 'MSFT', 'GOOG')
""")

# Convert to Polars for further processing
df_polars = pl.from_arrow(duckdb_result.arrow())

# Do complex feature engineering in Polars
features = df_polars.with_columns([
    pl.col('price').pct_change().over('ticker').alias('return'),
    pl.col('volume').rolling_mean(20).over('ticker').alias('avg_volume_20d')
])

# Convert to pandas for sklearn
df_pandas = features.to_pandas()

# Use in machine learning pipeline
# from sklearn.ensemble import RandomForestRegressor
# model = RandomForestRegressor()
# ...
```

Practical Recommendation

For MSc-level empirical finance work, I recommend:

1. **Learn pandas first**: It's the standard, and you'll encounter it everywhere
2. **Add Polars for performance**: When pandas becomes slow, rewrite critical sections in Polars
3. **Use DuckDB for data wrangling**: ETL pipelines and complex joins are often cleaner in SQL
4. **Profile your code**: Don't optimize prematurely—measure where time is actually spent

The best practitioners know all three and choose the right tool for each task.

11.7. Summary

DataFrames are the fundamental abstraction for tabular data in empirical finance. This chapter introduced three powerful tools:

pandas: The established standard with broad ecosystem support. Best for general-purpose data analysis, learning, and integration with statistical libraries.

Polars: A modern, high-performance alternative with lazy evaluation and automatic parallelization. Best for large datasets and performance-critical applications.

DuckDB: SQL interface for analytical queries on DataFrames and files. Best for complex joins, aggregations, and data pipeline work.

The key insight is that these tools are complementary. Understanding when to use each—and how to combine them—will make you significantly more productive in empirical finance research.

In the next chapters, we'll dive deeper into specific operations: data cleaning, merging and joining, time-series operations, and efficient computation with large datasets.

11.8. Further Reading

- pandas documentation: Comprehensive guide to pandas
- Polars User Guide: Official Polars documentation
- DuckDB documentation: Complete DuckDB reference
- McKinney, W. (2022). *Python for Data Analysis, 3rd Edition.* O'Reilly. (by pandas creator)
- VanderPlas, J. (2016). *Python Data Science Handbook.* O'Reilly.

12. Data Input and File Formats

The first step in any empirical analysis is getting your data into a form where you can work with it. In the world of finance, data comes in many shapes and sizes: CSV files exported from Bloomberg, Excel spreadsheets from analysts, proprietary formats from data vendors, and increasingly, large datasets stored in efficient binary formats. Understanding how to work with different file formats efficiently is crucial for productive data analysis.

The choice of file format matters more than you might think. A format that works well for a small dataset of a few hundred stocks might become painfully slow when you're dealing with millions of trades. Similarly, while Excel is ubiquitous in finance, it has limitations when working with large datasets or when you need to ensure reproducibility. In this chapter, we'll explore the most common file formats you'll encounter in empirical finance and learn how to work with them using both pandas and polars, two powerful Python libraries for data manipulation.

Pandas has long been the standard for data analysis in Python, offering a rich API and extensive functionality. Polars is a newer library built in Rust that offers significantly faster performance, especially for larger datasets, while maintaining a similar conceptual model. Throughout this chapter, we'll show examples using both libraries so you can choose the right tool for your specific needs.

12.1. CSV Files

CSV (Comma-Separated Values) is perhaps the most universal data format. It's simple, human-readable, and supported by virtually every data tool and programming language. In finance, you'll frequently encounter CSV files: historical price data from exchanges, bulk downloads from data vendors, exports from Bloomberg terminals, and outputs from other analysis tools.

The beauty of CSV lies in its simplicity. Each line represents a row of data, with values separated by commas (or sometimes other delimiters like tabs or semicolons). The first line typically contains column names. Here's what a simple CSV file of stock prices might look like:

```
date,ticker,price,volume
2024-01-02,AAPL,185.64,52000000
2024-01-02,MSFT,374.58,28000000
2024-01-03,AAPL,184.25,48000000
2024-01-03,MSFT,371.32,25000000
```

However, this simplicity comes with some drawbacks. CSV files store everything as text, so data types must be inferred when reading. They don't compress particularly well compared to binary formats. And for very large files, parsing text can be slow compared to reading binary data directly.

12.1.1. Reading CSV Files with Pandas

> **pandas supports many file formats**
>
> Pandas supports reading and writing a wide variety of file formats beyond CSV, including Excel spreadsheets, SAS data sets, Stata data sets, and many others. However, not all functionality is implemented in pandas itself—it relies on third-party libraries for some formats.
> For CSV files specifically, pandas includes multiple parsing engines. The Python engine is the most flexible but also the slowest. You can also use the C engine (the default) or the PyArrow engine for better performance. Whenever you need to handle a nonstandard CSV file or work with a different file format, consult the pandas I/O documentation for guidance on which engine or approach to use.

Reading a CSV file with pandas is straightforward using the `read_csv()` function:

```
import pandas as pd

# Basic CSV reading
df = pd.read_csv("stock_prices.csv")

# View the first few rows
print(df.head())
```

The `read_csv()` function is remarkably flexible and can handle many variations and edge cases you'll encounter with real-world data:

```
# Specify date columns to parse
df = pd.read_csv(
    "stock_prices.csv",
    parse_dates=["date"],                                    ①
    index_col="date"                                         ②
)

# Handle different delimiters (e.g., semicolon-separated)
df = pd.read_csv("data.csv", sep=";")                        ③

# Skip rows or specify column names
df = pd.read_csv(
    "data.csv",
    skiprows=2,                                              ④
    names=["date", "price", "vol"],                          ⑤
    header=None                                              ⑥
)

# Handle missing values
```

```
df = pd.read_csv(
    "data.csv",
    na_values=[".", "NA", "n/a"]                                    ⑦
)

# Read only specific columns
df = pd.read_csv(
    "large_file.csv",
    usecols=["date", "ticker", "price"]                             ⑧
)
```

① Convert date column to datetime during parsing
② Use the date column as the DataFrame index
③ Use semicolon as delimiter instead of comma
④ Skip the first 2 rows of the file
⑤ Provide custom column names
⑥ Indicate the file has no header row
⑦ Treat additional strings as missing values (NaN)
⑧ Read only the specified columns, ignoring others

> **Memory considerations with large CSV files**
>
> When reading very large CSV files, pandas loads the entire file into memory. If your CSV file is larger than your available RAM, you have several options:
>
> 1. Read the file in chunks using the `chunksize` parameter
> 2. Use `usecols` to read only the columns you need
> 3. Use polars with lazy evaluation (discussed below)
> 4. Convert the file to a more efficient format like Parquet

For truly large files, you can process them in chunks:

```
# Process CSV in chunks
chunk_size = 100000
results = []

for chunk in pd.read_csv("large_file.csv", chunksize=chunk_size):   ①
    # Process each chunk
    filtered = chunk[chunk["volume"] > 1000000]                     ②
    results.append(filtered)                                        ③

# Combine all results
df = pd.concat(results, ignore_index=True)                          ④
```

① Read the file in chunks of 100,000 rows at a time

② Filter each chunk to keep only rows with high volume
③ Append filtered results to a list
④ Combine all filtered chunks into a single DataFrame

Pandas also supports reading compressed CSV files directly. If you have a CSV file compressed as `.gz`, `.bz2`, `.zip`, or `.xz`, pandas will automatically decompress it:

```
# Read a gzip-compressed CSV file
df = pd.read_csv("stock_prices.csv.gz")

# Read from a ZIP archive containing a single CSV file
df = pd.read_csv("data.zip")
```

12.1.2. Writing CSV Files with Pandas

Writing data to CSV is just as easy using the `to_csv()` method:

```
# Basic CSV writing
df.to_csv("output.csv")

# Customize the output
df.to_csv(
    "output.csv",
    index=False,           # Don't write row indices
    float_format="%.4f",   # Format floats to 4 decimal places
    encoding="utf-8"       # Specify encoding
)

# Write to a compressed file
df.to_csv("output.csv.gz", compression="gzip")
```

12.1.3. Reading CSV Files with Polars

Polars offers two ways to read CSV files: eager reading with `read_csv()` and lazy reading with `scan_csv()`. Eager reading loads the entire file into memory immediately, while lazy reading creates a query plan that can be optimized before execution.

```
import polars as pl

# Eager reading - similar to pandas
df = pl.read_csv("stock_prices.csv")

# Specify data types for better performance
df = pl.read_csv(
```

```
    "stock_prices.csv",
    dtypes={"ticker": pl.Categorical, "volume": pl.Int64},  ①
    try_parse_dates=True  ②
)

# Lazy reading - more efficient for large files
lazy_df = pl.scan_csv("large_file.csv")  ③

# Build up a query (not executed yet)
result = (
    lazy_df
    .filter(pl.col("volume") > 1000000)  ④
    .select(["date", "ticker", "price"])  ⑤
    .group_by("ticker")
    .agg(pl.col("price").mean().alias("avg_price"))  ⑥
)

# Execute the optimized query
final_df = result.collect()  ⑦
```

① Explicitly specify data types for columns
② Automatically parse date-like columns
③ Create a lazy frame that defers reading until needed
④ Filter rows (not executed yet)
⑤ Select only the columns we need (not executed yet)
⑥ Calculate average price by ticker (not executed yet)
⑦ Execute the optimized query plan and return the result

The lazy approach is particularly powerful because polars can optimize the entire query before executing it. For example, if you're filtering rows and then selecting specific columns, polars can read only those columns for only the rows that pass the filter, dramatically reducing I/O.

> **Lazy evaluation for large datasets**
>
> When working with CSV files that are large but still fit in memory after filtering, using `scan_csv()` followed by filtering operations can be much faster than `read_csv()`. Polars will optimize the query to avoid reading unnecessary data.
> This is especially valuable when your CSV file has many columns but you only need a few, or when you'll be filtering most rows out early in your analysis.

12.1.4. Writing CSV Files with Polars

Writing CSV files with polars is similarly straightforward:

```
# Write a DataFrame to CSV
df.write_csv("output.csv")

# You can also write to a file-like object
with open("output.csv", "w") as f:
    df.write_csv(f)
```

12.2. Parquet and Apache Arrow

While CSV files are universal and human-readable, they're not the most efficient format for large datasets. This is where Parquet comes in. Parquet is a columnar storage format originally developed for use in the Hadoop ecosystem but now widely used across the data science world. It offers several key advantages over CSV:

1. **Compression**: Parquet files are typically 5-10x smaller than equivalent CSV files
2. **Speed**: Reading Parquet is much faster because data is stored in a binary format optimized for quick access
3. **Type preservation**: Data types are stored in the file, so there's no need for type inference
4. **Complex data types**: Parquet handles complex types like datetimes with timezone information natively, encoding them exactly as-is with no loss of information or conversion errors
5. **Column selection**: You can read only specific columns without parsing the entire file
6. **Predicate pushdown**: Filters can be applied during reading, avoiding loading unnecessary data

The columnar storage format is key to Parquet's efficiency. Rather than storing data row by row (like CSV), Parquet stores each column's data together. This means if you want to read just a few columns, Parquet only needs to read those column chunks, not the entire file. It also means values in the same column can be compressed very efficiently since similar values are stored together.

Parquet is built on Apache Arrow, a cross-language development platform for in-memory data. Arrow defines a standardized columnar memory format that is highly efficient for analytical operations. Both pandas and polars can work with Arrow, and polars is built on Arrow from the ground up.

When to use Parquet

Parquet is an excellent choice for:

- Large datasets that you'll read multiple times
- Datasets where you often need only a subset of columns
- Sharing data between different tools and languages
- Long-term data storage with efficient compression

Stick with CSV for:

- Small datasets where file size doesn't matter
- Data that needs to be human-readable

- Sharing with tools that don't support Parquet
- Quick exports for inspection in Excel or text editors

When I receive a dataset larger than a few megabytes, my first step is usually to run a script that reads it and converts it to Parquet. All subsequent processing then benefits from Parquet's faster read times and smaller file size.

12.2.1. Reading and Writing Parquet with Pandas

Pandas makes working with Parquet files very simple:

```
import pandas as pd

# Read a Parquet file
df = pd.read_parquet("stock_prices.parquet")

# Read only specific columns
df = pd.read_parquet(
    "large_file.parquet",
    columns=["date", "ticker", "price"]
)

# Read using PyArrow as both the engine and in-memory backend
df = pd.read_parquet(
    "stock_prices.parquet",
    engine="pyarrow",
    dtype_backend="pyarrow"
)

# Write to Parquet
df.to_parquet("output.parquet")

# Write without the index
df.to_parquet("output.parquet", index=False)
```

Using `dtype_backend="pyarrow"` keeps the data in pandas using the PyArrow memory format, which is the same format that polars uses internally. This can provide significant performance improvements for certain operations, though not all pandas features are fully supported with this backend yet.

12.2.2. Reading and Writing Parquet with Polars

Polars has excellent support for Parquet and, like with CSV, offers both eager and lazy reading:

```python
import polars as pl

# Eager reading
df = pl.read_parquet("stock_prices.parquet")

# Read only specific columns
df = pl.read_parquet(
    "large_file.parquet",
    columns=["date", "ticker", "price"]  ①
)

# Lazy reading with scan_parquet
lazy_df = pl.scan_parquet("large_file.parquet")  ②

# Build an optimized query
result = (
    lazy_df
    .filter(pl.col("date") >= "2024-01-01")  ③
    .select(["ticker", "price", "volume"])
    .group_by("ticker")
    .agg([
        pl.col("price").mean().alias("avg_price"),
        pl.col("volume").sum().alias("total_volume")
    ])
    .collect()  ④
)

# Write to Parquet
df.write_parquet("output.parquet")
```

① Only read the specified columns from the Parquet file
② Create a lazy frame that defers execution
③ Filters are pushed down to the Parquet reader when possible
④ Execute the query and return the result

The combination of Parquet's columnar format and polars' query optimization can be extremely powerful. When you use `scan_parquet()`, polars can push filters down to the Parquet reader, meaning only relevant data is loaded into memory.

12.2.3. Practical Example: CSV to Parquet Conversion

A common workflow is to receive data as CSV (perhaps from a vendor or Bloomberg) and immediately convert it to Parquet for more efficient analysis:

```
import pandas as pd

# Read CSV with appropriate data types
df = pd.read_csv(
    "stock_data.csv",
    parse_dates=["date"],
    dtype={
        "ticker": "category",  # Use categorical for string columns with few unique values
        "volume": "int64"
    }
)

# Save as Parquet
df.to_parquet("stock_data.parquet", compression="snappy")

# Future reads will be much faster
df_fast = pd.read_parquet("stock_data.parquet")
```

For very large CSV files, you might do this conversion with polars:

```
import polars as pl

# Lazy read CSV, optimize, and write to Parquet
(
    pl.scan_csv("large_stock_data.csv")                                    ①
    .select([
        pl.col("date").str.to_date(),                                      ②
        pl.col("ticker").cast(pl.Categorical),                             ③
        pl.col("price").cast(pl.Float64),
        pl.col("volume").cast(pl.Int64)
    ])
    .sink_parquet("stock_data.parquet")                                    ④
)
```

① Create a lazy frame from the CSV file
② Parse the date string column to a proper date type
③ Convert ticker to categorical for memory efficiency
④ Stream results directly to Parquet without loading into memory

The `sink_parquet()` method is particularly efficient because it streams the data to disk without loading everything into memory first.

12.3. Excel Files

Despite the advantages of formats like CSV and Parquet, Excel remains ubiquitous in finance. Analysts send Excel files, regulatory filings include Excel attachments, and many financial models live in Excel workbooks. As a result, being able to read from and write to Excel is an essential skill for empirical finance work.

Excel files (.xlsx) are actually compressed archives containing XML files that describe the workbook structure, data, and formatting. This makes them more complex than simple text formats but also allows them to store multiple sheets, formulas, formatting, and metadata in a single file.

12.3.1. Reading Excel Files with Pandas

Pandas relies on third-party libraries for Excel support. These are optional dependencies that are not installed by default when you install pandas. If a required library is missing, pandas will notify you when you try to use the related function and indicate which package to install. You can then install the missing dependency with `uv add`.

Pandas provides robust Excel support through the `read_excel()` function:

```
import pandas as pd

# Read the first sheet
df = pd.read_excel("financial_data.xlsx")

# Read a specific sheet by name
df = pd.read_excel("financial_data.xlsx", sheet_name="Returns")  ①

# Read a specific sheet by index (0-indexed)
df = pd.read_excel("financial_data.xlsx", sheet_name=1)  ②

# Read multiple sheets at once
sheets_dict = pd.read_excel(
    "financial_data.xlsx",
    sheet_name=["Returns", "Fundamentals"]  ③
)
# Returns a dictionary: {"Returns": df1, "Fundamentals": df2}

# Read all sheets
all_sheets = pd.read_excel("financial_data.xlsx", sheet_name=None)  ④
```

① Specify a sheet by its name
② Specify a sheet by its position (0-indexed)
③ Read multiple sheets by passing a list of names
④ Use `None` to read all sheets as a dictionary

Excel files often have headers on multiple rows, merged cells, or data that doesn't start in cell A1. Pandas provides parameters to handle these cases:

```
# Skip rows at the beginning
df = pd.read_excel(
    "report.xlsx",
    skiprows=3,  # Skip first 3 rows
    sheet_name="Data"
)

# Specify header row
df = pd.read_excel(
    "report.xlsx",
    header=2  # Use row 2 (0-indexed) as column names
)

# Specify which columns to read
df = pd.read_excel(
    "data.xlsx",
    usecols="A:D"  # Read only columns A through D
)

# Or specify by column names
df = pd.read_excel(
    "data.xlsx",
    usecols=["Date", "Price", "Volume"]
)

# Handle dates
df = pd.read_excel(
    "data.xlsx",
    parse_dates=["date_column"]
)
```

> ⚠ Excel file size and performance
>
> Reading Excel files is significantly slower than reading CSV or Parquet files, especially for large datasets. If you're repeatedly reading the same Excel file, consider converting it to Parquet for better performance. Additionally, Excel has a row limit of 1,048,576 rows. If you're working with larger datasets, you'll need to use a different format.

12.3.2. Writing Excel Files with Pandas

Writing to Excel is similarly straightforward:

```
# Write to a single sheet
df.to_excel("output.xlsx", sheet_name="Data", index=False)

# Write multiple DataFrames to different sheets
with pd.ExcelWriter("output.xlsx") as writer:
    df_returns.to_excel(writer, sheet_name="Returns", index=False)
    df_fundamentals.to_excel(writer, sheet_name="Fundamentals", index=False)
    df_summary.to_excel(writer, sheet_name="Summary", index=False)

# Specify float formatting
df.to_excel(
    "output.xlsx",
    sheet_name="Data",
    index=False,
    float_format="%.4f"
)
```

You can also use the ExcelWriter context manager to add formatting:

```
# Create an Excel file with formatting
with pd.ExcelWriter(
    "formatted_output.xlsx",
    engine="xlsxwriter"
) as writer:
    df.to_excel(writer, sheet_name="Data", index=False)

    # Get the workbook and worksheet objects
    workbook = writer.book
    worksheet = writer.sheets["Data"]

    # Add a format
    format1 = workbook.add_format({"num_format": "$#,##0.00"})

    # Apply formatting to a column (e.g., column B)
    worksheet.set_column("B:B", 12, format1)
```

12.3.3. Reading Excel Files with Polars

Polars also supports reading Excel files, though with a focus on data extraction rather than preserving formatting:

```
import polars as pl

# Read the first sheet
```

```
df = pl.read_excel("financial_data.xlsx")

# Read a specific sheet
df = pl.read_excel(
    "financial_data.xlsx",
    sheet_name="Returns"
)

# Read with specific sheet index
df = pl.read_excel("financial_data.xlsx", sheet_id=2)
```

Polars uses the `fastexcel` engine by default, which is optimized for speed. For more complex Excel files, you might need to install additional engines like `openpyxl` or `xlsx2csv`.

12.3.4. Writing Excel Files with Polars

Polars can write DataFrames to Excel files, though it requires the `xlsxwriter` library:

```
import polars as pl

# Write to Excel (requires xlsxwriter)
df.write_excel("output.xlsx")

# Write to a specific worksheet
df.write_excel("output.xlsx", worksheet="MyData")
```

Excel as a data exchange format

While Excel is convenient for sharing data with non-technical stakeholders, it's not ideal for:

- Large datasets (performance and row limits)
- Preserving exact data types (dates can be problematic)
- Version control (binary format is hard to diff)
- Reproducible research (formulas and manual edits aren't tracked)

Consider using Excel for final outputs and presentation, but use CSV or Parquet for data storage and analysis pipelines.

12.4. Choosing the Right Format

We've covered three common file formats: CSV, Parquet, and Excel. Each has its place in a data analysis workflow. Here's a decision guide:

Use CSV when:

- Working with small to medium datasets (under 1 GB)
- You need human-readable data
- Sharing data with tools that don't support other formats
- You value simplicity and universality over performance
- Version controlling data (text files work well with git)

Use Parquet when:

- Working with large datasets (multiple GB or more)
- You'll read the data multiple times
- You often need only a subset of columns
- Performance and storage efficiency matter
- Sharing data between different tools and languages
- Building data pipelines and ETL processes

Use Excel when:

- Sharing data with business users
- You need multiple related tables in one file (multiple sheets)
- The recipient expects Excel format
- Working with small datasets where performance doesn't matter
- You need to include formatting, formulas, or charts

In practice, many empirical finance workflows involve all three: receiving data in Excel or CSV, converting to Parquet for efficient analysis, and exporting results back to Excel for presentation.

12.5. Best Practices for Data I/O

As you work with different file formats, keep these best practices in mind:

1. Specify data types explicitly

When reading data, especially CSV files, explicitly specifying data types can improve both performance and correctness:

```
# Pandas
df = pd.read_csv(
    "data.csv",
    dtype={
        "ticker": "category",
        "volume": "int64",
        "price": "float64"
    },
    parse_dates=["date"]
```

```
)

# Polars
df = pl.read_csv(
    "data.csv",
    dtypes={
        "ticker": pl.Categorical,
        "volume": pl.Int64,
        "price": pl.Float64
    },
    try_parse_dates=True
)
```

2. Use compression for large files

Both CSV and Parquet support compression. For CSV files you'll store long-term, use gzip compression:

```
# Pandas - automatic compression based on filename
df.to_csv("data.csv.gz")

# Explicit compression
df.to_parquet("data.parquet", compression="snappy")
```

3. Consider storage vs. memory tradeoffs

Parquet files with strong compression (like gzip or zstd) are smaller on disk but take longer to read. For files you'll read frequently, snappy compression offers a good balance. For archival storage, use stronger compression.

4. Validate data after reading

After reading data, especially from external sources, validate that it matches your expectations:

```
import pandas as pd
import numpy as np

# Sample data with some issues to detect
df = pd.DataFrame({
    "date": pd.to_datetime(["2024-01-02", "2024-01-02", "2024-01-03", "2024-01-03"]),
    "ticker": ["AAPL", "MSFT", "AAPL", "AAPL"],
    "price": [185.64, 374.58, None, 184.25],
    "volume": [52000000, 28000000, 48000000, 48000000]
})
```

```
# Check for missing values
df.isnull().sum()
```

```
date      0
ticker    0
price     1
volume    0
dtype: int64
```

```
# Check data types
df.dtypes
```

```
date      datetime64[ns]
ticker            object
price            float64
volume             int64
dtype: object
```

```
# Basic statistics
df.describe()
```

	date	price	volume
count	4	3.000000	4.000000e+00
mean	2024-01-02 12:00:00	248.156667	4.400000e+07
min	2024-01-02 00:00:00	184.250000	2.800000e+07
25%	2024-01-02 00:00:00	184.945000	4.300000e+07
50%	2024-01-02 12:00:00	185.640000	4.800000e+07
75%	2024-01-03 00:00:00	280.110000	4.900000e+07
max	2024-01-03 00:00:00	374.580000	5.200000e+07
std	NaN	109.488024	1.083205e+07

```
# Check for duplicates
df.duplicated().sum()
```

```
np.int64(0)
```

5. Document your data sources

Keep a record of where your data came from, when it was downloaded, and any transformations applied. This is crucial for reproducibility:

```
# Add metadata when saving
metadata = {
    "source": "Bloomberg Terminal",
    "download_date": "2024-01-15",
    "description": "Daily stock prices for S&P 500 constituents"
}

# Parquet supports metadata
df.to_parquet("data.parquet", metadata=metadata)
```

Working with data files is the foundation of empirical finance research. By understanding the strengths and limitations of different file formats, and by using the right tools for each format, you can build efficient, reproducible data analysis pipelines. In the next chapter, we'll explore data cleaning techniques to prepare your data for analysis.

13. Data Cleaning

Data cleaning is one of the most critical and time-consuming steps in any empirical analysis. In finance, working with real-world data means confronting issues like duplicate records, missing observations, data entry errors, and outliers. These problems are not merely technical annoyances—they can lead to incorrect conclusions, flawed trading strategies, or misleading research findings if not handled properly.

The goal of data cleaning is not to make data "perfect" but to make it suitable for analysis. This means understanding the nature of data quality issues, their potential causes, and the trade-offs involved in different cleaning approaches. In this chapter, we will cover the essential techniques for detecting and handling common data quality problems using Python's main data manipulation libraries: pandas and Polars.

Before we begin, let's understand an important principle: data cleaning decisions should be **documented** and **reproducible**. Every choice you make—whether to drop duplicates, impute missing values, or remove outliers—affects your results. Your code should clearly show what cleaning steps were taken and why, so that others (including your future self) can understand and validate your approach.

> Data Wrangler in VS Code
>
> Microsoft provides a VS Code extension called "Data Wrangler" that provides a nice UI for filtering, cleaning, and transforming data in pandas DataFrames. When done, you can export the Python code to replicate your steps.
> See: Explore Data Like a Pro in VS Code with Data Wrangler, Pandas and Python

13.1. Detecting and Handling Duplicates

Duplicate records are a common problem in financial datasets. They can arise from multiple sources: data feed errors that transmit the same trade twice, mistakes in data aggregation where the same record appears in multiple files, or simple human error in manual data entry. Duplicates are particularly problematic because they can distort statistical analyses, inflate trading volumes, and create artificial patterns in the data.

13.1.1. Identifying duplicates

The first step in handling duplicates is identifying them. A duplicate can be defined in different ways depending on your data structure and business context.

Types of duplicates

1. **Complete duplicates**: All columns have identical values
2. **Partial duplicates**: Only certain key columns are identical (e.g., same date and ticker, but different price)
3. **Near duplicates**: Values are very similar but not exactly identical (often due to floating-point precision)

The appropriate definition depends on your data and use case.

Let's start with a simple example using stock price data:

```
import pandas as pd
import polars as pl
from datetime import datetime, timedelta

# Create sample stock price data with duplicates
dates = pd.date_range('2024-01-01', periods=5, freq='D')
data = {
    'date': list(dates) + [dates[2]],  # Duplicate date                    ①
    'ticker': ['AAPL', 'AAPL', 'AAPL', 'AAPL', 'AAPL', 'AAPL'],
    'price': [150.0, 152.5, 155.0, 153.0, 157.0, 155.0],
    'volume': [1000000, 1100000, 1200000, 950000, 1300000, 1200000]
}

df_pd = pd.DataFrame(data)
df_pd
```

① Adding a duplicate of the third date to create a duplicate record.

	date	ticker	price	volume
0	2024-01-01	AAPL	150.0	1000000
1	2024-01-02	AAPL	152.5	1100000
2	2024-01-03	AAPL	155.0	1200000
3	2024-01-04	AAPL	153.0	950000
4	2024-01-05	AAPL	157.0	1300000
5	2024-01-03	AAPL	155.0	1200000

Now let's detect duplicates. The `duplicated()` method returns a boolean Series indicating which rows are duplicates:

```
# Check for complete duplicates
df_pd.duplicated()
```

```
0    False
1    False
2    False
3    False
4    False
5     True
dtype: bool
```

We can also check for duplicates based on specific columns:

```
# Check for duplicates based on specific columns (date and ticker)
df_pd.duplicated(subset=['date', 'ticker'])
```

```
0    False
1    False
2    False
3    False
4    False
5     True
dtype: bool
```

To see which rows are duplicates, we can filter the DataFrame:

```
# See which rows are duplicates (keep=False marks all duplicates)
df_pd[df_pd.duplicated(subset=['date', 'ticker'], keep=False)]
```

	date	ticker	price	volume
2	2024-01-03	AAPL	155.0	1200000
5	2024-01-03	AAPL	155.0	1200000

The `keep` parameter controls which duplicate to mark:

- `keep='first'` (default): Mark all duplicates as `True` except the first occurrence
- `keep='last'`: Mark all duplicates as `True` except the last occurrence
- `keep=False`: Mark all duplicates as `True`, including the first occurrence

Now let's see the same operations in Polars:

```
df_pl = pl.DataFrame(data)
df_pl
```

date datetime[μs]	ticker str	price f64	volume i64
2024-01-01 00:00:00	"AAPL"	150.0	1000000
2024-01-02 00:00:00	"AAPL"	152.5	1100000
2024-01-03 00:00:00	"AAPL"	155.0	1200000
2024-01-04 00:00:00	"AAPL"	153.0	950000
2024-01-05 00:00:00	"AAPL"	157.0	1300000
2024-01-03 00:00:00	"AAPL"	155.0	1200000

To find duplicates in Polars, we use `is_duplicated()`:

```
# Check for duplicates based on date and ticker
df_pl.filter(pl.struct(['date', 'ticker']).is_duplicated())
```

date datetime[μs]	ticker str	price f64	volume i64
2024-01-03 00:00:00	"AAPL"	155.0	1200000
2024-01-03 00:00:00	"AAPL"	155.0	1200000

13.1.2. Removing duplicates

Once we've identified duplicates, we need to decide how to handle them. The most common approaches are:

1. **Drop all duplicates**: Keep only the first (or last) occurrence
2. **Aggregate duplicates**: Combine duplicate rows using aggregation (e.g., average prices)
3. **Manual investigation**: For small numbers of duplicates, inspect each case individually

Let's see how to implement these approaches. In pandas, we use `drop_duplicates()`:

```
# pandas: Drop duplicates, keeping first occurrence
df_pd_clean = df_pd.drop_duplicates(subset=['date', 'ticker'], keep='first')
df_pd_clean
```

	date	ticker	price	volume
0	2024-01-01	AAPL	150.0	1000000
1	2024-01-02	AAPL	152.5	1100000
2	2024-01-03	AAPL	155.0	1200000
3	2024-01-04	AAPL	153.0	950000
4	2024-01-05	AAPL	157.0	1300000

In Polars, we use the `unique()` method:

```
# Polars: Drop duplicates
df_pl_clean = df_pl.unique(subset=['date', 'ticker'], keep='first')
df_pl_clean
```

date datetime[μs]	ticker str	price f64	volume i64
2024-01-02 00:00:00	”AAPL”	152.5	1100000
2024-01-01 00:00:00	”AAPL”	150.0	1000000
2024-01-04 00:00:00	”AAPL”	153.0	950000
2024-01-05 00:00:00	”AAPL”	157.0	1300000
2024-01-03 00:00:00	”AAPL”	155.0	1200000

For the aggregation approach, suppose we want to take the average price and total volume when duplicates exist:

```
# pandas: Aggregate duplicates
df_pd_agg = df_pd.groupby(['date', 'ticker'], as_index=False).agg({
    'price': 'mean',
    'volume': 'sum'
})
df_pd_agg
```

	date	ticker	price	volume
0	2024-01-01	AAPL	150.0	1000000
1	2024-01-02	AAPL	152.5	1100000
2	2024-01-03	AAPL	155.0	2400000
3	2024-01-04	AAPL	153.0	950000
4	2024-01-05	AAPL	157.0	1300000

```
# Polars: Aggregate duplicates
df_pl_agg = df_pl.group_by(['date', 'ticker']).agg([
    pl.col('price').mean(),
    pl.col('volume').sum()
])
df_pl_agg
```

date datetime[μs]	ticker str	price f64	volume i64
2024-01-01 00:00:00	”AAPL”	150.0	1000000
2024-01-04 00:00:00	”AAPL”	153.0	950000

date datetime[μs]	ticker str	price f64	volume i64
2024-01-05 00:00:00	"AAPL"	157.0	1300000
2024-01-02 00:00:00	"AAPL"	152.5	1100000
2024-01-03 00:00:00	"AAPL"	155.0	2400000

⚠ Common pitfall: implicit duplicates

Be careful about duplicates that arise from data structure decisions. For example, if you merge two datasets and accidentally create a many-to-many join, you may generate duplicates without realizing it. Always check the row count before and after joins to ensure you haven't inadvertently created duplicates. We discuss joins in detail in Chapter 16.

13.1.3. Best practices for duplicate handling

1. **Always investigate before removing**: Look at a sample of duplicates to understand why they exist
2. **Document your decisions**: Add comments explaining which columns define a duplicate and why
3. **Keep track of removals**: Log how many duplicates you removed and their characteristics
4. **Consider the source**: Different data sources may have different duplicate patterns

```
# Example: Comprehensive duplicate handling with logging
def handle_duplicates(df, subset_cols, keep='first', verbose=True):
    """
    Remove duplicates with logging.

    Parameters
    ----------
    df : pd.DataFrame
        Input dataframe
    subset_cols : list
        Columns that define a duplicate
    keep : str
        Which duplicate to keep ('first', 'last', or False)
    verbose : bool
        Whether to print diagnostic information

    Returns
    -------
    pd.DataFrame
        Cleaned dataframe
    """
    initial_rows = len(df)
```

```
    duplicated_mask = df.duplicated(subset=subset_cols, keep=keep)
    n_duplicates = duplicated_mask.sum()

    if verbose:
        print(f"Initial rows: {initial_rows}")
        print(f"Duplicates found: {n_duplicates}")
        if n_duplicates > 0:
            print(f"Duplicate rows (sample):")
            print(df[duplicated_mask].head())

    df_clean = df[~duplicated_mask].copy()

    if verbose:
        print(f"Final rows: {len(df_clean)}")
        print(f"Rows removed: {initial_rows - len(df_clean)}")

    return df_clean

# Use the function
df_cleaned = handle_duplicates(df_pd, subset_cols=['date', 'ticker'])
```

```
Initial rows: 6
Duplicates found: 1
Duplicate rows (sample):
        date ticker  price   volume
5 2024-01-03   AAPL  155.0  1200000
Final rows: 5
Rows removed: 1
```

13.2. Missing Data: Detection and Imputation

Missing data is ubiquitous in financial datasets. Stock markets close on holidays, companies report earnings quarterly but not daily, economic indicators are released with delays, and data collection systems occasionally fail. How you handle missing data can significantly impact your analysis, and there is rarely a single "correct" approach.

13.2.1. Understanding missing data mechanisms

Before deciding how to handle missing data, it's important to understand why data might be missing. Statisticians classify missing data into three categories:

1. **Missing Completely at Random (MCAR)**: The probability that a value is missing is unrelated to any observed or unobserved data. For example, if a data feed randomly drops 1% of all observations due to network issues, the missing data is MCAR.

2. **Missing at Random (MAR)**: The probability that a value is missing depends on observed data but not on the missing value itself. For example, small-cap stocks might have more missing volume data than large-cap stocks, but conditional on market cap, the missingness is random.

3. **Missing Not at Random (MNAR)**: The probability that a value is missing depends on the missing value itself. For example, companies might be less likely to report earnings when performance is poor, or traders might fail to report large losses.

> **!** Why this matters
>
> The type of missingness affects which imputation methods are valid. MCAR is the easiest to handle (you can often just drop missing observations), while MNAR is the most problematic because missing data itself carries information. Unfortunately, in real financial data, you often cannot definitively determine which mechanism is operating, so you need to think carefully about the context.

13.2.2. Detecting missing data

Python uses different representations for missing data depending on the data type:

- **`NaN`** (Not a Number): Used for missing floating-point values
- **`None`**: Python's built-in null value
- **`NaT`** (Not a Time): Used for missing datetime values
- **`pd.NA`**: pandas' new missing indicator for all dtypes

> **i** Historical context: missing values in pandas
>
> Initially, pandas used NumPy as its backend and could not represent missing values explicitly. Instead, it used `NaN` for floats and `None` (a Python object) for strings. This approach had important limitations:
>
> 1. For integers, pandas would first convert all values in the column to float and use `NaN` to represent missing values. This meant you couldn't have a truly integer column with missing values.
>
> 2. `NaN` has a specific meaning in floating-point arithmetic: it represents a value that is "not a number," such as the result of an undefined operation (e.g., `log(-1)`). Reusing `NaN` to mean "missing" conflates two distinct concepts and complicates handling of true `NaN` results.
>
> Polars and newer backends for pandas (such as PyArrow) explicitly support missing data indicators for all column types, avoiding these limitations.

Let's create a dataset with missing values:

```
import numpy as np

data_missing = {
    'date': pd.date_range('2024-01-01', periods=10, freq='D'),
    'ticker': ['AAPL'] * 10,
    'price': [150.0, 152.5, np.nan, 153.0, 157.0, np.nan, 159.0, 158.5, np.nan, 161.0],
    'volume': [1000000, np.nan, 1200000, 950000, np.nan, 1100000, 1300000, np.nan, 1250000, 1400000],
    'dividend': [0.0, 0.0, 0.0, 0.25, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
}

df_missing = pd.DataFrame(data_missing)
df_missing
```

	date	ticker	price	volume	dividend
0	2024-01-01	AAPL	150.0	1000000.0	0.00
1	2024-01-02	AAPL	152.5	NaN	0.00
2	2024-01-03	AAPL	NaN	1200000.0	0.00
3	2024-01-04	AAPL	153.0	950000.0	0.25
4	2024-01-05	AAPL	157.0	NaN	0.00
5	2024-01-06	AAPL	NaN	1100000.0	0.00
6	2024-01-07	AAPL	159.0	1300000.0	0.00
7	2024-01-08	AAPL	158.5	NaN	0.00
8	2024-01-09	AAPL	NaN	1250000.0	0.00
9	2024-01-10	AAPL	161.0	1400000.0	0.00

Now let's detect missing values. The `isna()` method returns a boolean DataFrame indicating which values are missing:

```
df_missing.isna()
```

	date	ticker	price	volume	dividend
0	False	False	False	False	False
1	False	False	False	True	False
2	False	False	True	False	False
3	False	False	False	False	False
4	False	False	False	True	False
5	False	False	True	False	False
6	False	False	False	False	False
7	False	False	False	True	False
8	False	False	True	False	False
9	False	False	False	False	False

We can count missing values per column:

```
df_missing.isna().sum()
```

```
date        0
ticker      0
price       3
volume      3
dividend    0
dtype: int64
```

Or calculate the percentage of missing values:

```
df_missing.isna().mean() * 100
```

```
date         0.0
ticker       0.0
price       30.0
volume      30.0
dividend     0.0
dtype: float64
```

To see rows with any missing values:

```
df_missing[df_missing.isna().any(axis=1)]
```

	date	ticker	price	volume	dividend
1	2024-01-02	AAPL	152.5	NaN	0.0
2	2024-01-03	AAPL	NaN	1200000.0	0.0
4	2024-01-05	AAPL	157.0	NaN	0.0
5	2024-01-06	AAPL	NaN	1100000.0	0.0
7	2024-01-08	AAPL	158.5	NaN	0.0
8	2024-01-09	AAPL	NaN	1250000.0	0.0

In Polars, we use `null_count()` to count missing values:

```
# Polars version - convert from pandas DataFrame
# (Polars uses None/null for missing values, not np.nan)
df_missing_pl = pl.from_pandas(df_missing)

df_missing_pl.null_count()
```

date u32	ticker u32	price u32	volume u32	dividend u32
0	0	3	3	0

We can filter rows with missing values in a specific column:

```
df_missing_pl.filter(pl.col('price').is_null())
```

date datetime[ns]	ticker str	price f64	volume f64	dividend f64
2024-01-03 00:00:00	”AAPL”	null	1.2e6	0.0
2024-01-06 00:00:00	”AAPL”	null	1.1e6	0.0
2024-01-09 00:00:00	”AAPL”	null	1.25e6	0.0

13.2.3. Handling missing data: deletion

The simplest approach to missing data is to delete it. There are two main strategies:

1. **Listwise deletion** (complete case analysis): Rows with any missing values are completely dropped from the sample, so they are not used in any calculation or analysis. This approach is simple and ensures consistency across analyses, but can significantly reduce sample size if missingness is widespread.
2. **Pairwise deletion**: For each calculation or analysis (e.g., for each regression), use all observations for which we have the required variables. This approach maximizes the use of available data but can lead to different sample sizes for different analyses, making comparisons difficult.

Listwise deletion removes any row with missing values:

```
df_complete = df_missing.dropna()
df_complete
```

	date	ticker	price	volume	dividend
0	2024-01-01	AAPL	150.0	1000000.0	0.00
3	2024-01-04	AAPL	153.0	950000.0	0.25
6	2024-01-07	AAPL	159.0	1300000.0	0.00
9	2024-01-10	AAPL	161.0	1400000.0	0.00

```
print(f"Rows removed: {len(df_missing) - len(df_complete)}")
```

```
Rows removed: 6
```

We can also drop rows only if specific columns are missing:

```
df_price_complete = df_missing.dropna(subset=['price'])
df_price_complete
```

	date	ticker	price	volume	dividend
0	2024-01-01	AAPL	150.0	1000000.0	0.00
1	2024-01-02	AAPL	152.5	NaN	0.00
3	2024-01-04	AAPL	153.0	950000.0	0.25
4	2024-01-05	AAPL	157.0	NaN	0.00
6	2024-01-07	AAPL	159.0	1300000.0	0.00
7	2024-01-08	AAPL	158.5	NaN	0.00
9	2024-01-10	AAPL	161.0	1400000.0	0.00

Or drop columns with any missing values:

```
df_no_missing_cols = df_missing.dropna(axis=1)
df_no_missing_cols
```

	date	ticker	dividend
0	2024-01-01	AAPL	0.00
1	2024-01-02	AAPL	0.00
2	2024-01-03	AAPL	0.00
3	2024-01-04	AAPL	0.25
4	2024-01-05	AAPL	0.00
5	2024-01-06	AAPL	0.00
6	2024-01-07	AAPL	0.00
7	2024-01-08	AAPL	0.00
8	2024-01-09	AAPL	0.00
9	2024-01-10	AAPL	0.00

In Polars, we use `drop_nulls()`:

```
df_complete_pl = df_missing_pl.drop_nulls()
df_complete_pl
```

date datetime[ns]	ticker str	price f64	volume f64	dividend f64
2024-01-01 00:00:00	”AAPL”	150.0	1e6	0.0
2024-01-04 00:00:00	”AAPL”	153.0	950000.0	0.25

date datetime[ns]	ticker str	price f64	volume f64	dividend f64
2024-01-07 00:00:00	"AAPL"	159.0	1.3e6	0.0
2024-01-10 00:00:00	"AAPL"	161.0	1.4e6	0.0

```
# Drop rows with null in specific column
df_price_complete_pl = df_missing_pl.drop_nulls(subset=['price'])
df_price_complete_pl
```

date datetime[ns]	ticker str	price f64	volume f64	dividend f64
2024-01-01 00:00:00	"AAPL"	150.0	1e6	0.0
2024-01-02 00:00:00	"AAPL"	152.5	null	0.0
2024-01-04 00:00:00	"AAPL"	153.0	950000.0	0.25
2024-01-05 00:00:00	"AAPL"	157.0	null	0.0
2024-01-07 00:00:00	"AAPL"	159.0	1.3e6	0.0
2024-01-08 00:00:00	"AAPL"	158.5	null	0.0
2024-01-10 00:00:00	"AAPL"	161.0	1.4e6	0.0

⚠ When deletion is problematic

Deletion can introduce bias if:

- You lose a large percentage of your data
- Missingness is not random (MAR or MNAR)
- Missing data occurs in key variables

In time series analysis, deleting observations can break the temporal structure of your data. Use deletion cautiously and always check how much data you're losing.

13.2.4. Handling missing data: imputation

Imputation means filling in missing values with plausible estimates. There are many imputation strategies, each with different assumptions and use cases.

Simple imputation methods

1. **Forward fill**: Use the last observed value
2. **Backward fill**: Use the next observed value
3. **Mean/median imputation**: Replace with the column mean or median

4. **Zero/constant imputation**: Replace with zero or another constant

In empirical finance, we usually want to avoid introducing look-ahead bias or imputing information that was not available at the time of observation. For this reason, forward fill and zero/constant imputation are the most commonly used imputation methods in financial research.

Forward fill uses the last observed value:

```
df_ffill = df_missing.copy()
df_ffill[['price', 'volume']] = df_ffill[['price', 'volume']].ffill()
df_ffill
```

	date	ticker	price	volume	dividend
0	2024-01-01	AAPL	150.0	1000000.0	0.00
1	2024-01-02	AAPL	152.5	1000000.0	0.00
2	2024-01-03	AAPL	152.5	1200000.0	0.00
3	2024-01-04	AAPL	153.0	950000.0	0.25
4	2024-01-05	AAPL	157.0	950000.0	0.00
5	2024-01-06	AAPL	157.0	1100000.0	0.00
6	2024-01-07	AAPL	159.0	1300000.0	0.00
7	2024-01-08	AAPL	158.5	1300000.0	0.00
8	2024-01-09	AAPL	158.5	1250000.0	0.00
9	2024-01-10	AAPL	161.0	1400000.0	0.00

Backward fill uses the next observed value:

```
df_bfill = df_missing.copy()
df_bfill[['price', 'volume']] = df_bfill[['price', 'volume']].bfill()
df_bfill
```

	date	ticker	price	volume	dividend
0	2024-01-01	AAPL	150.0	1000000.0	0.00
1	2024-01-02	AAPL	152.5	1200000.0	0.00
2	2024-01-03	AAPL	153.0	1200000.0	0.00
3	2024-01-04	AAPL	153.0	950000.0	0.25
4	2024-01-05	AAPL	157.0	1100000.0	0.00
5	2024-01-06	AAPL	159.0	1100000.0	0.00
6	2024-01-07	AAPL	159.0	1300000.0	0.00
7	2024-01-08	AAPL	158.5	1250000.0	0.00
8	2024-01-09	AAPL	161.0	1250000.0	0.00
9	2024-01-10	AAPL	161.0	1400000.0	0.00

Mean imputation replaces missing values with the column mean:

```
df_mean = df_missing.copy()
df_mean['price'] = df_mean['price'].fillna(df_mean['price'].mean())
df_mean['volume'] = df_mean['volume'].fillna(df_mean['volume'].mean())
df_mean
```

	date	ticker	price	volume	dividend
0	2024-01-01	AAPL	150.000000	1.000000e+06	0.00
1	2024-01-02	AAPL	152.500000	1.171429e+06	0.00
2	2024-01-03	AAPL	155.857143	1.200000e+06	0.00
3	2024-01-04	AAPL	153.000000	9.500000e+05	0.25
4	2024-01-05	AAPL	157.000000	1.171429e+06	0.00
5	2024-01-06	AAPL	155.857143	1.100000e+06	0.00
6	2024-01-07	AAPL	159.000000	1.300000e+06	0.00
7	2024-01-08	AAPL	158.500000	1.171429e+06	0.00
8	2024-01-09	AAPL	155.857143	1.250000e+06	0.00
9	2024-01-10	AAPL	161.000000	1.400000e+06	0.00

Median imputation is more robust to outliers:

```
df_median = df_missing.copy()
df_median['price'] = df_median['price'].fillna(df_median['price'].median())
df_median['volume'] = df_median['volume'].fillna(df_median['volume'].median())
df_median
```

	date	ticker	price	volume	dividend
0	2024-01-01	AAPL	150.0	1000000.0	0.00
1	2024-01-02	AAPL	152.5	1200000.0	0.00
2	2024-01-03	AAPL	157.0	1200000.0	0.00
3	2024-01-04	AAPL	153.0	950000.0	0.25
4	2024-01-05	AAPL	157.0	1200000.0	0.00
5	2024-01-06	AAPL	157.0	1100000.0	0.00
6	2024-01-07	AAPL	159.0	1300000.0	0.00
7	2024-01-08	AAPL	158.5	1200000.0	0.00
8	2024-01-09	AAPL	157.0	1250000.0	0.00
9	2024-01-10	AAPL	161.0	1400000.0	0.00

Constant value imputation:

```
df_zero = df_missing.copy()
df_zero['dividend'] = df_zero['dividend'].fillna(0)
df_zero
```

	date	ticker	price	volume	dividend
0	2024-01-01	AAPL	150.0	1000000.0	0.00
1	2024-01-02	AAPL	152.5	NaN	0.00
2	2024-01-03	AAPL	NaN	1200000.0	0.00
3	2024-01-04	AAPL	153.0	950000.0	0.25
4	2024-01-05	AAPL	157.0	NaN	0.00
5	2024-01-06	AAPL	NaN	1100000.0	0.00
6	2024-01-07	AAPL	159.0	1300000.0	0.00
7	2024-01-08	AAPL	158.5	NaN	0.00
8	2024-01-09	AAPL	NaN	1250000.0	0.00
9	2024-01-10	AAPL	161.0	1400000.0	0.00

In Polars, we use `forward_fill()` and `fill_null()`:

```
df_ffill_pl = df_missing_pl.select([
    pl.col('date'),
    pl.col('ticker'),
    pl.col('price').forward_fill(),
    pl.col('volume').forward_fill(),
    pl.col('dividend')
])
df_ffill_pl
```

date datetime[ns]	ticker str	price f64	volume f64	dividend f64
2024-01-01 00:00:00	”AAPL”	150.0	1e6	0.0
2024-01-02 00:00:00	”AAPL”	152.5	1e6	0.0
2024-01-03 00:00:00	”AAPL”	152.5	1.2e6	0.0
2024-01-04 00:00:00	”AAPL”	153.0	950000.0	0.25
2024-01-05 00:00:00	”AAPL”	157.0	950000.0	0.0
2024-01-06 00:00:00	”AAPL”	157.0	1.1e6	0.0
2024-01-07 00:00:00	”AAPL”	159.0	1.3e6	0.0
2024-01-08 00:00:00	”AAPL”	158.5	1.3e6	0.0
2024-01-09 00:00:00	”AAPL”	158.5	1.25e6	0.0
2024-01-10 00:00:00	”AAPL”	161.0	1.4e6	0.0

```
# Polars mean imputation
mean_price = df_missing_pl['price'].mean()
mean_volume = df_missing_pl['volume'].mean()

df_mean_pl = df_missing_pl.with_columns([
    pl.col('price').fill_null(mean_price),
```

```
    pl.col('volume').fill_null(mean_volume)
])
df_mean_pl
```

date datetime[ns]	ticker str	price f64	volume f64	dividend f64
2024-01-01 00:00:00	”AAPL”	150.0	1e6	0.0
2024-01-02 00:00:00	”AAPL”	152.5	1.1714e6	0.0
2024-01-03 00:00:00	”AAPL”	155.857143	1.2e6	0.0
2024-01-04 00:00:00	”AAPL”	153.0	950000.0	0.25
2024-01-05 00:00:00	”AAPL”	157.0	1.1714e6	0.0
2024-01-06 00:00:00	”AAPL”	155.857143	1.1e6	0.0
2024-01-07 00:00:00	”AAPL”	159.0	1.3e6	0.0
2024-01-08 00:00:00	”AAPL”	158.5	1.1714e6	0.0
2024-01-09 00:00:00	”AAPL”	155.857143	1.25e6	0.0
2024-01-10 00:00:00	”AAPL”	161.0	1.4e6	0.0

Choosing an imputation method for financial data

The appropriate method depends on your data and context:

- **Forward fill**: Good for prices and slowly-changing variables (assumes last observation is still valid)
- **Mean/median**: Reasonable for cross-sectional data, but can distort distributions and relationships
- **Zero**: Appropriate when zero is a meaningful value (e.g., dividends, corporate actions)
- **Interpolation**: Useful when you expect smooth changes over time

Never use imputation blindly—think about what each method assumes about your data.

Linear interpolation

For time series data, linear interpolation can provide more realistic estimates than simple forward/backward filling:

```
df_interp = df_missing.copy()
df_interp['price'] = df_interp['price'].interpolate(method='linear')
df_interp['volume'] = df_interp['volume'].interpolate(method='linear')
df_interp
```

	date	ticker	price	volume	dividend
0	2024-01-01	AAPL	150.00	1000000.0	0.00
1	2024-01-02	AAPL	152.50	1100000.0	0.00
2	2024-01-03	AAPL	152.75	1200000.0	0.00
3	2024-01-04	AAPL	153.00	950000.0	0.25
4	2024-01-05	AAPL	157.00	1025000.0	0.00
5	2024-01-06	AAPL	158.00	1100000.0	0.00
6	2024-01-07	AAPL	159.00	1300000.0	0.00
7	2024-01-08	AAPL	158.50	1275000.0	0.00
8	2024-01-09	AAPL	159.75	1250000.0	0.00
9	2024-01-10	AAPL	161.00	1400000.0	0.00

Time-based interpolation accounts for irregular spacing and requires a DatetimeIndex:

```
df_interp_time = df_missing.copy().set_index('date')
df_interp_time['price'] = df_interp_time['price'].interpolate(method='time')
df_interp_time = df_interp_time.reset_index()
df_interp_time
```

	date	ticker	price	volume	dividend
0	2024-01-01	AAPL	150.00	1000000.0	0.00
1	2024-01-02	AAPL	152.50	NaN	0.00
2	2024-01-03	AAPL	152.75	1200000.0	0.00
3	2024-01-04	AAPL	153.00	950000.0	0.25
4	2024-01-05	AAPL	157.00	NaN	0.00
5	2024-01-06	AAPL	158.00	1100000.0	0.00
6	2024-01-07	AAPL	159.00	1300000.0	0.00
7	2024-01-08	AAPL	158.50	NaN	0.00
8	2024-01-09	AAPL	159.75	1250000.0	0.00
9	2024-01-10	AAPL	161.00	1400000.0	0.00

Group-based imputation

Often, you want to impute missing values differently for different groups. For example, you might want to impute missing returns for each industry separately:

```
# Create data with multiple tickers
data_grouped = {
    'date': pd.date_range('2024-01-01', periods=6, freq='D').tolist() * 2,
    'ticker': ['AAPL'] * 6 + ['MSFT'] * 6,
    'price': [150.0, np.nan, 155.0, 153.0, np.nan, 161.0,
```

```
            250.0, 252.0, np.nan, 258.0, 260.0, np.nan],
    'sector': ['Tech'] * 12
}

df_grouped = pd.DataFrame(data_grouped)
df_grouped
```

	date	ticker	price	sector
0	2024-01-01	AAPL	150.0	Tech
1	2024-01-02	AAPL	NaN	Tech
2	2024-01-03	AAPL	155.0	Tech
3	2024-01-04	AAPL	153.0	Tech
4	2024-01-05	AAPL	NaN	Tech
5	2024-01-06	AAPL	161.0	Tech
6	2024-01-01	MSFT	250.0	Tech
7	2024-01-02	MSFT	252.0	Tech
8	2024-01-03	MSFT	NaN	Tech
9	2024-01-04	MSFT	258.0	Tech
10	2024-01-05	MSFT	260.0	Tech
11	2024-01-06	MSFT	NaN	Tech

Forward fill within each ticker group:

```
df_grouped_ffill = df_grouped.copy()
df_grouped_ffill['price'] = df_grouped_ffill.groupby('ticker')['price'].ffill()
df_grouped_ffill
```

	date	ticker	price	sector
0	2024-01-01	AAPL	150.0	Tech
1	2024-01-02	AAPL	150.0	Tech
2	2024-01-03	AAPL	155.0	Tech
3	2024-01-04	AAPL	153.0	Tech
4	2024-01-05	AAPL	153.0	Tech
5	2024-01-06	AAPL	161.0	Tech
6	2024-01-01	MSFT	250.0	Tech
7	2024-01-02	MSFT	252.0	Tech
8	2024-01-03	MSFT	252.0	Tech
9	2024-01-04	MSFT	258.0	Tech
10	2024-01-05	MSFT	260.0	Tech
11	2024-01-06	MSFT	260.0	Tech

Mean imputation within each sector:

```
df_grouped_mean = df_grouped.copy()
df_grouped_mean['price'] = df_grouped_mean.groupby('sector')['price'].transform(
    lambda x: x.fillna(x.mean())
)
df_grouped_mean
```

	date	ticker	price	sector
0	2024-01-01	AAPL	150.000	Tech
1	2024-01-02	AAPL	204.875	Tech
2	2024-01-03	AAPL	155.000	Tech
3	2024-01-04	AAPL	153.000	Tech
4	2024-01-05	AAPL	204.875	Tech
5	2024-01-06	AAPL	161.000	Tech
6	2024-01-01	MSFT	250.000	Tech
7	2024-01-02	MSFT	252.000	Tech
8	2024-01-03	MSFT	204.875	Tech
9	2024-01-04	MSFT	258.000	Tech
10	2024-01-05	MSFT	260.000	Tech
11	2024-01-06	MSFT	204.875	Tech

In Polars, we use the `over()` method for group-based operations:

```
df_grouped_pl = pl.from_pandas(df_grouped)

# Forward fill within groups
df_grouped_ffill_pl = df_grouped_pl.with_columns([
    pl.col('price').forward_fill().over('ticker')
])
df_grouped_ffill_pl
```

date datetime[ns]	ticker str	price f64	sector str
2024-01-01 00:00:00	”AAPL”	150.0	”Tech”
2024-01-02 00:00:00	”AAPL”	150.0	”Tech”
2024-01-03 00:00:00	”AAPL”	155.0	”Tech”
2024-01-04 00:00:00	”AAPL”	153.0	”Tech”
2024-01-05 00:00:00	”AAPL”	153.0	”Tech”
…	…	…	…
2024-01-02 00:00:00	”MSFT”	252.0	”Tech”
2024-01-03 00:00:00	”MSFT”	252.0	”Tech”
2024-01-04 00:00:00	”MSFT”	258.0	”Tech”

date datetime[ns]	ticker str	price f64	sector str
2024-01-05 00:00:00	”MSFT”	260.0	”Tech”
2024-01-06 00:00:00	”MSFT”	260.0	”Tech”

```
# Mean imputation within groups
df_grouped_mean_pl = df_grouped_pl.with_columns([
    pl.col('price').fill_null(pl.col('price').mean()).over('sector')
])
df_grouped_mean_pl
```

date datetime[ns]	ticker str	price f64	sector str
2024-01-01 00:00:00	”AAPL”	150.0	”Tech”
2024-01-02 00:00:00	”AAPL”	204.875	”Tech”
2024-01-03 00:00:00	”AAPL”	155.0	”Tech”
2024-01-04 00:00:00	”AAPL”	153.0	”Tech”
2024-01-05 00:00:00	”AAPL”	204.875	”Tech”
…	…	…	…
2024-01-02 00:00:00	”MSFT”	252.0	”Tech”
2024-01-03 00:00:00	”MSFT”	204.875	”Tech”
2024-01-04 00:00:00	”MSFT”	258.0	”Tech”
2024-01-05 00:00:00	”MSFT”	260.0	”Tech”
2024-01-06 00:00:00	”MSFT”	204.875	”Tech”

13.3. Data Validation

Data validation is the process of checking whether your data meets quality standards and assumptions before analysis. In finance, invalid data can lead to catastrophic errors—imagine executing a trading strategy based on incorrect prices or publishing research with flawed data.

Validation encompasses several activities:

- Checking data types and formats
- Verifying data ranges and constraints
- Identifying outliers and anomalies
- Ensuring logical consistency
- Cross-checking against external sources

Automated validation with pandera

For systematic data validation, consider using pandera, a Python framework for automatic validation of DataFrames. Pandera lets you define schemas that specify expected data types, value ranges, and other constraints, then automatically validates your data against these schemas. It supports both pandas and Polars.

13.3.1. Validating data types and formats

First, ensure that each column has the correct data type:

```
# Create sample data with type issues
data_types = {
    'date': ['2024-01-01', '2024-01-02', '2024-01-03', 'invalid'],
    'ticker': ['AAPL', 'AAPL', 'AAPL', 'AAPL'],
    'price': ['150.0', '152.5', '155.0', '153.0'],  # Stored as strings
    'volume': [1000000, 1100000, 1200000, -950000]  # Negative volume
}

df_types = pd.DataFrame(data_types)
df_types
```

	date	ticker	price	volume
0	2024-01-01	AAPL	150.0	1000000
1	2024-01-02	AAPL	152.5	1100000
2	2024-01-03	AAPL	155.0	1200000
3	invalid	AAPL	153.0	-950000

```
df_types.dtypes
```

```
date      object
ticker    object
price     object
volume     int64
dtype: object
```

Now let's validate and fix the types. We use `errors='coerce'` to convert invalid values to `NaN` instead of raising an error:

```
# Convert price to numeric, coercing errors to NaN
df_types['price'] = pd.to_numeric(df_types['price'], errors='coerce')

# Convert date to datetime
df_types['date'] = pd.to_datetime(df_types['date'], errors='coerce')

df_types
```

	date	ticker	price	volume
0	2024-01-01	AAPL	150.0	1000000
1	2024-01-02	AAPL	152.5	1100000
2	2024-01-03	AAPL	155.0	1200000
3	NaT	AAPL	153.0	-950000

```
df_types.dtypes
```

```
date      datetime64[ns]
ticker            object
price            float64
volume             int64
dtype: object
```

We can identify rows with conversion errors by checking for `NaN` values:

```
invalid_dates = df_types['date'].isna()
print(f"Rows with invalid dates: {invalid_dates.sum()}")
```

```
Rows with invalid dates: 1
```

```
df_types[invalid_dates]
```

	date	ticker	price	volume
3	NaT	AAPL	153.0	-950000

Common type issues in financial data

- **Dates**: Mixed formats (MM/DD/YYYY vs DD/MM/YYYY), Excel date serials, timezone issues
- **Numbers**: Thousands separators, currency symbols, percentage signs stored as text
- **Missing values**: Coded as -999, 'NA', empty strings, or other sentinel values

- **Categorical variables**: Inconsistent capitalization or spelling

Always check types early in your pipeline and convert them explicitly.

13.3.2. Range and constraint validation

Financial data often has natural constraints. Prices must be positive, probabilities must be between 0 and 1, and returns are typically bounded (though extreme events can violate typical ranges).

```
# Create data with constraint violations
data_constraints = {
    'date': pd.date_range('2024-01-01', periods=5, freq='D'),
    'ticker': ['AAPL'] * 5,
    'price': [150.0, -152.5, 155.0, 1000000.0, 157.0],  # Negative and extreme prices
    'volume': [1000000, 1100000, 1200000, -950000, 1300000],  # Negative volume
    'return': [0.01, 0.02, -0.01, 5.0, -0.03]  # Extreme return
}

df_constraints = pd.DataFrame(data_constraints)

# Define validation rules
def validate_price(price):
    """Check if price is valid."""
    return (price > 0) & (price < 100000)

def validate_volume(volume):
    """Check if volume is valid."""
    return volume >= 0

def validate_return(ret):
    """Check if return is reasonable."""
    return (ret > -1) & (ret < 1)

# Apply validation
df_constraints['price_valid'] = validate_price(df_constraints['price'])
df_constraints['volume_valid'] = validate_volume(df_constraints['volume'])
df_constraints['return_valid'] = validate_return(df_constraints['return'])

df_constraints
```

	date	ticker	price	volume	return	price_valid	volume_valid	return_valid
0	2024-01-01	AAPL	150.0	1000000	0.01	True	True	True

	date	ticker	price	volume	return	price_valid	volume_valid	return_valid
1	2024-01-02	AAPL	-152.5	1100000	0.02	False	True	True
2	2024-01-03	AAPL	155.0	1200000	-0.01	True	True	True
3	2024-01-04	AAPL	1000000.0	-950000	5.00	False	False	False
4	2024-01-05	AAPL	157.0	1300000	-0.03	True	True	True

We can identify invalid rows by combining the validation columns:

```
invalid_mask = ~(df_constraints['price_valid'] &
                 df_constraints['volume_valid'] &
                 df_constraints['return_valid'])
print(f"Invalid rows: {invalid_mask.sum()}")
```

```
Invalid rows: 2
```

```
df_constraints[invalid_mask]
```

	date	ticker	price	volume	return	price_valid	volume_valid	return_valid
1	2024-01-02	AAPL	-152.5	1100000	0.02	False	True	True
3	2024-01-04	AAPL	1000000.0	-950000	5.00	False	False	False

In Polars, we can use expressions for validation:

```
df_constraints_pl = pl.DataFrame(data_constraints)

# Add validation columns
df_validated_pl = df_constraints_pl.with_columns([
    ((pl.col('price') > 0) & (pl.col('price') < 100000)).alias('price_valid'),
    (pl.col('volume') >= 0).alias('volume_valid'),
    ((pl.col('return') > -1) & (pl.col('return') < 1)).alias('return_valid')
])

df_validated_pl
```

date datetime[ns]	ticker str	price f64	volume i64	return f64	price_valid bool	volume_valid bool	return_valid bool
2024-01-01 00:00:00	”AAPL”	150.0	1000000	0.01	true	true	true
2024-01-02 00:00:00	”AAPL”	-152.5	1100000	0.02	false	true	true
2024-01-03 00:00:00	”AAPL”	155.0	1200000	-0.01	true	true	true

date datetime[ns]	ticker str	price f64	volume i64	return f64	price_valid bool	volume_valid bool	return_valid bool
2024-01-04 00:00:00	”AAPL”	1e6	-950000	5.0	false	false	false
2024-01-05 00:00:00	”AAPL”	157.0	1300000	-0.03	true	true	true

```
# Filter to invalid rows
invalid_pl = df_validated_pl.filter(
    ~pl.col('price_valid') | ~pl.col('volume_valid') | ~pl.col('return_valid')
)
invalid_pl
```

date datetime[ns]	ticker str	price f64	volume i64	return f64	price_valid bool	volume_valid bool	return_valid bool
2024-01-02 00:00:00	”AAPL”	-152.5	1100000	0.02	false	true	true
2024-01-04 00:00:00	”AAPL”	1e6	-950000	5.0	false	false	false

13.3.3. Detecting outliers

Outliers are observations that deviate significantly from other observations. In finance, outliers could indicate:

- Genuine extreme events (market crashes, flash crashes)
- Data errors (wrong decimal place, data feed glitches)
- Special situations (stock splits, spinoffs)

Outlier detection requires domain knowledge. A 50% daily return is an error for a typical stock but normal for a penny stock or during a takeover announcement.

Statistical methods for outlier detection

Common approaches include:

1. **Z-score method**: Flag observations more than N standard deviations from the mean
2. **IQR method**: Flag observations outside 1.5 times the interquartile range
3. **Modified Z-score**: Use median absolute deviation instead of standard deviation (more robust)
4. **Domain-specific rules**: Use knowledge about reasonable ranges

Choosing the right method for a particular application is beyond the scope of this book and requires careful consideration of the data characteristics and analysis goals.

Handling outliers

Once you've identified outliers, you have several options:

1. **Keep them**: If they're genuine extreme events
2. **Remove them**: If they're clearly errors
3. **Cap/Winsorize**: Replace extreme values with a threshold
4. **Transform the data**: Use log transformation or other methods to reduce the impact
5. **Investigate**: Manually check each outlier

> ⚠ Be cautious with outlier removal
>
> In finance, extreme events are often the most important observations. The 2008 financial crisis, COVID-19 market crash, and other tail events are "outliers" but carry critical information. Before removing outliers:
>
> 1. Investigate each one individually if possible
> 2. Check if they coincide with known events (earnings announcements, news, market crises)
> 3. Consider the impact on your conclusions if you keep vs. remove them
> 4. Document your decision and reasoning
>
> When in doubt, perform your analysis both with and without outliers to assess sensitivity.

14. Data Structuring and Aggregation

Working with financial data requires more than knowing how to load datasets and run calculations. The structure of your data—how observations are identified, indexed, and organized—fundamentally shapes what analyses are possible and what mistakes are easy to make. A portfolio returns dataset indexed by date allows straightforward time-series operations but makes cross-sectional comparisons awkward. The same data with a multi-level index on date and ticker enables both temporal and cross-sectional analysis but requires more careful handling of group operations.

This chapter covers three essential skills for working with structured financial data. First, we examine how keys, indices, and identifiers work—the foundation for correctly organizing observations. Second, we explore grouping and aggregation operations that let you compute statistics within subsets of data (returns by sector, volatility by year, etc.). Third, we discuss common pitfalls specific to financial data, particularly look-ahead bias and survivorship bias, which can invalidate empirical results if not handled carefully.

14.1. Keys, Indices, and Identifiers

Financial datasets rarely consist of independent observations. Stock returns are measured repeatedly over time for many different securities. Portfolio weights change across rebalancing dates. Trades occur at specific timestamps for specific instruments. To work with such data correctly, you need to understand how observations are identified and how that identification is represented in your data structures.

14.1.1. What Makes an Observation Unique?

Consider a dataset of daily stock returns. What information do you need to uniquely identify a single return observation? At minimum, you need to know which stock (ticker symbol or identifier) and which date. Neither alone suffices: Apple's ticker AAPL appears thousands of times in a long dataset (once per day), and any given date like 2024-01-15 appears once for every stock in your universe. The combination of ticker and date uniquely identifies each return.

This combination is called a **composite key** or **multi-level identifier**. Other examples in finance include:

- Trade data: exchange + timestamp + order ID
- Options prices: underlying + expiration + strike + call/put
- Cross-country macro data: country + date + variable
- Portfolio holdings: portfolio ID + date + security ID

The key affects everything downstream: what joins are valid, what grouping operations make sense, what reshaping is possible, and critically, what the index of a time series means.

14.1.2. Representing Keys in DataFrames

There are two main approaches to representing composite keys in DataFrames: as regular columns in the data or as a special index structure. Both pandas and Polars support the regular column approach, while only pandas supports the index approach.

Approach 1: Keys as Regular Columns

The simplest representation keeps all identifiers as ordinary columns:

```
import pandas as pd
import polars as pl

# Pandas version
df_pandas = pd.DataFrame({
    'ticker': ['AAPL', 'AAPL', 'MSFT', 'MSFT'],
    'date': pd.to_datetime(['2024-01-15', '2024-01-16', '2024-01-15', '2024-01-16']),
    'return': [0.012, -0.005, 0.008, 0.003]
})

# Polars version
df_polars = pl.DataFrame({
    'ticker': ['AAPL', 'AAPL', 'MSFT', 'MSFT'],
    'date': pl.date_range(pl.date(2024, 1, 15), pl.date(2024, 1, 16), "1d").repeat_by(2),
    'return': [0.012, -0.005, 0.008, 0.003]
})
```

This representation is explicit and flexible. Every piece of identifying information is visible in the data. Operations like filtering, grouping, and joining reference columns by name, making code clear and intentions obvious.

Approach 2: Pandas MultiIndex

Pandas offers an alternative where identifying columns become a special index structure:

```
df_indexed = df_pandas.set_index(['ticker', 'date'])
```

The result looks like:

```
                   return
ticker date
AAPL   2024-01-15   0.012
       2024-01-16  -0.005
MSFT   2024-01-15   0.008
       2024-01-16   0.003
```

Now `ticker` and `date` are no longer regular columns but levels of a hierarchical index. This structure enables convenient operations like:

```
# Select all observations for AAPL
df_indexed.loc['AAPL']

# Select specific ticker-date combination
df_indexed.loc[('AAPL', '2024-01-15')]

# Unstack to wide format
df_indexed.unstack('ticker')
```

The MultiIndex approach can make certain operations more concise, particularly when working with panel data where you frequently filter by identifier values or reshape between long and wide formats. However, it also introduces complexity. Index levels behave differently from columns: they don't appear in `df.columns`, they require different syntax to modify, and they can complicate operations when you need to treat identifiers as data (e.g., creating a new column based on ticker characteristics).

14.1.3. Practical Guidance on Index Choice

Our general recommendation is to use regular columns for key variables, unless you have an explicit need for a pandas index. Regular columns make code more explicit and portable across libraries. However, a pandas index can simplify your work in specific situations: certain time-series operations (like resampling), creating plots or tables where the index provides axis labels, or when using third-party libraries that expect an index.

14.1.4. Temporal Identifiers and DatetimeIndex

Time is special in financial data. Unlike categorical identifiers like ticker symbols, dates and timestamps have inherent ordering and spacing. A pandas DatetimeIndex (single-level index with datetime values) provides specialized functionality for time-series work:

```
# Daily returns with date index
returns = pd.Series(
    [0.012, -0.005, 0.008],
    index=pd.to_datetime(['2024-01-15', '2024-01-16', '2024-01-17']),
    name='AAPL'
)

# Convenient date-based selection
returns['2024-01-16']  # Single date
returns['2024-01-15':'2024-01-16']  # Date range

# Resampling and frequency conversion
```

```
monthly_returns = (1 + returns).resample('ME').prod() - 1                    ①

# Time-aware operations
returns.shift(1)                                                             ②
returns.rolling(window=5).mean()                                             ③
```

① Compound daily returns to monthly by converting to gross returns (1 + r), multiplying within each month, then converting back to net returns.
② Lag the series by one period—useful for avoiding look-ahead bias.
③ Compute a 5-day rolling mean.

This is one area where pandas indexing genuinely simplifies common operations. The alternative using regular columns requires more verbose syntax:

```
# Polars equivalent for date filtering
df_polars.filter(
    (pl.col('date') >= pl.date(2024, 1, 15)) &
    (pl.col('date') <= pl.date(2024, 1, 16))
)

# Polars rolling mean with explicit ordering
df_polars.sort('date').select([
    'date',
    pl.col('return').rolling_mean(window_size=5).alias('ma_5')
])
```

The pandas DatetimeIndex approach is particularly valuable when working with a single time series (one security or portfolio) where time is the primary organizing principle. For panel data with many securities, a regular date column often proves more flexible.

Polars encourages explicit column-based operations and provides excellent performance for time-based filtering and aggregation without special index structures. Its approach scales better to large datasets and makes parallelization transparent.

14.1.5. Identifier Best Practices

Several practical considerations matter when choosing and working with identifiers:

Uniqueness and Validation

Always verify that your chosen keys actually uniquely identify observations:

```
# Pandas: check for duplicates in composite key
duplicates = df_pandas.duplicated(subset=['ticker', 'date'], keep=False)
if duplicates.any():
    print(f"Found {duplicates.sum()} duplicate observations")
    print(df_pandas[duplicates])

# Polars: check for duplicates
duplicate_count = (
    df_polars
    .group_by(['ticker', 'date'])
    .agg(pl.len().alias('count'))
    .filter(pl.col('count') > 1)
)
```

Duplicate keys often indicate data quality problems: double-counting trades, duplicate downloads, or errors in joins. Finding them early prevents subtle errors in aggregation and analysis.

Identifier Stability

Ticker symbols change. Companies get acquired or reorganize. CUSIP identifiers remain stable but aren't always available. When working with long time-series, consider using permanent identifiers like PERMNO (from CRSP) or assigning your own stable internal IDs that you map to tickers.

```
# Maintaining a ticker-to-permno mapping
ticker_map = pd.DataFrame({
    'permno': [14593, 10107],
    'ticker': ['AAPL', 'MSFT'],
    'start_date': pd.to_datetime(['1980-12-12', '1986-03-13']),
    'end_date': pd.to_datetime(['2024-12-31', '2024-12-31'])
})

# Join with date validation to handle ticker changes
def get_permno(df, ticker_map):
    return df.merge(
        ticker_map,
        on='ticker',
        how='left'
    ).query('date >= start_date and date <= end_date')
```

Missing Identifiers

Financial data often has gaps. A stock might not trade on certain days. An option chain might lack certain strikes. Design your data structures to make missing data explicit rather than creating ambiguous rows:

```
# Bad: using dummy values
df_bad = pd.DataFrame({
    'ticker': ['AAPL', 'NONE', 'MSFT'],  # 'NONE' for missing
    'return': [0.01, 0.0, 0.02]
})

# Good: using None/null
df_good = pd.DataFrame({
    'ticker': ['AAPL', None, 'MSFT'],
    'return': [0.01, None, 0.02]
})

# Polars explicitly handles null
df_polars = pl.DataFrame({
    'ticker': ['AAPL', None, 'MSFT'],
    'return': [0.01, None, 0.02]
})
```

This distinction matters because real zeros and missing values have different meanings in finance. A zero return is information; a missing return means we don't know what happened.

14.2. Grouping and Aggregation

Much of financial data analysis involves computing statistics within subsets: average returns by sector, volatility by year, portfolio weights by strategy. The split-apply-combine pattern—dividing data into groups, computing something for each group, and combining results—is fundamental to empirical work.

14.2.1. The GroupBy Operation

The core operation splits your dataset into groups based on one or more columns, applies a function to each group, and combines the results. Both pandas and Polars implement this pattern, though with different syntax and performance characteristics.

Basic GroupBy Example

Consider computing average returns by sector:

```
# Sample data: daily returns for multiple stocks
data = pd.DataFrame({
    'ticker': ['AAPL', 'AAPL', 'MSFT', 'MSFT', 'JPM', 'JPM'],
    'date': pd.to_datetime(['2024-01-15', '2024-01-16'] * 3),
    'sector': ['Technology', 'Technology', 'Technology', 'Technology', 'Financials', 'Financials'],
    'return': [0.012, -0.005, 0.008, 0.003, -0.002, 0.015]
```

```
})

# Pandas groupby
sector_avg_pandas = data.groupby('sector')['return'].mean()

# Polars groupby
data_polars = pl.DataFrame(data)
sector_avg_polars = (
    data_polars
    .group_by('sector')
    .agg(pl.col('return').mean())
)
```

Both produce the same result: average returns for Technology and Financials. The key difference is syntax style. Pandas uses method chaining with implicit column selection (`['return']`), while Polars uses explicit expressions (`pl.col('return').mean()`).

14.2.2. Multiple Aggregations

Real analysis rarely computes just one statistic. You typically want mean, standard deviation, count, and other metrics simultaneously:

```
# Pandas: multiple aggregations
sector_stats_pandas = data.groupby('sector')['return'].agg([
    ('mean', 'mean'),
    ('std', 'std'),
    ('count', 'count')
])

# Alternative pandas syntax using dictionary
sector_stats_pandas_alt = data.groupby('sector').agg({
    'return': ['mean', 'std', 'count']
})

# Polars: explicit expression for each statistic
sector_stats_polars = (
    data_polars
    .group_by('sector')
    .agg([
        pl.col('return').mean().alias('mean'),
        pl.col('return').std().alias('std'),
        pl.col('return').count().alias('count')
    ])
)
```

The Polars approach requires more typing but makes each calculation explicit. This verbosity pays off in complex aggregations where you're computing different statistics on different columns or using custom expressions.

14.2.3. Grouping by Multiple Keys

Financial data often requires grouping by multiple dimensions. For example, computing monthly returns by stock requires grouping by both ticker and month:

```
# Extended data with more dates
np.random.seed(42)
dates = pd.date_range('2024-01-01', '2024-03-31', freq='D')
tickers = ['AAPL', 'MSFT', 'JPM']

data_panel = pd.DataFrame({
    'ticker': np.repeat(tickers, len(dates)),
    'date': np.tile(dates, len(tickers)),
    'return': np.random.normal(0.001, 0.02, len(dates) * len(tickers))
})

# Add month identifier
data_panel['month'] = data_panel['date'].dt.to_period('M')

# Pandas: group by ticker and month
monthly_returns_pandas = (
    data_panel
    .groupby(['ticker', 'month'])['return']
    .apply(lambda x: (1 + x).prod() - 1)  # Compound returns
)

# Polars: explicit grouping
data_panel_polars = pl.DataFrame(data_panel)
monthly_returns_polars = (
    data_panel_polars
    .group_by(['ticker', 'month'])
    .agg(
        ((1 + pl.col('return')).product() - 1).alias('monthly_return')
    )
)
```

Notice the calculation itself: daily returns compound multiplicatively, not additively. This is a common pattern in financial aggregation where the mathematical operation matters. Simple averaging would be incorrect for returns.

14.2.4. Custom Aggregation Functions

Sometimes built-in aggregations aren't sufficient. You might need to compute Sharpe ratios, apply winsorization, or implement custom risk metrics:

```
def sharpe_ratio(returns, risk_free_rate=0.0):
    """Compute annualized Sharpe ratio from daily returns."""
    excess_returns = returns - risk_free_rate / 252                    ①
    return np.sqrt(252) * excess_returns.mean() / excess_returns.std()  ②

# Pandas: apply custom function
sector_sharpe_pandas = (
    data_panel
    .groupby(['ticker'])['return']
    .apply(sharpe_ratio)
)
```

① Convert annual risk-free rate to daily by dividing by 252 trading days.
② Annualize the Sharpe ratio by multiplying by $\sqrt{252}$ (volatility scales with $\sqrt{T}$).

```
# Polars: requires more explicit approach
# Option 1: Convert to pandas for complex custom functions
sector_sharpe_polars_via_pandas = (
    data_panel_polars
    .to_pandas()
    .groupby('ticker')['return']
    .apply(sharpe_ratio)
)

# Option 2: Use Polars expressions (more efficient)
def sharpe_ratio_polars(returns_col, risk_free_rate=0.0):
    """Sharpe ratio as Polars expression."""
    excess = returns_col - risk_free_rate / 252
    return (pl.lit(252).sqrt() * excess.mean() / excess.std()).alias('sharpe')

sector_sharpe_polars = (
    data_panel_polars
    .group_by('ticker')
    .agg(sharpe_ratio_polars(pl.col('return')))
)
```

Pandas's `apply` method accepts arbitrary Python functions, offering maximum flexibility at the cost of performance (it applies the function to each group sequentially in Python). Polars requires expressing custom operations using its expression language, which enables query optimization and parallel execution but requires more thought about how to structure the calculation.

For complex custom functions, pandas is often more convenient. For operations that can be expressed using Polars's built-in functions (which cover most standard statistical and mathematical operations), Polars provides better performance.

14.2.5. Time-Based Aggregation and Resampling

Financial data frequently needs aggregation at different time frequencies: daily data to monthly, tick data to minute bars, etc. Pandas provides convenient resampling functionality:

```
# Single time series of daily returns
aapl_returns = (
    data_panel[data_panel['ticker'] == 'AAPL']
    .set_index('date')['return']
)

# Resample to monthly, compounding returns
monthly_aapl = (1 + aapl_returns).resample('ME').prod() - 1

# Resample to weekly, computing volatility
weekly_vol = aapl_returns.resample('W').std() * np.sqrt(5)

# Multiple statistics at monthly frequency
monthly_stats = aapl_returns.resample('ME').agg({
    'return': [
        ('compound_return', lambda x: (1 + x).prod() - 1),
        ('volatility', lambda x: x.std() * np.sqrt(21)),
        ('days', 'count')
    ]
})
```

For panel data (multiple securities), combine groupby with resampling:

```
# Panel data: group by ticker, then resample
monthly_panel = (
    data_panel
    .set_index('date')
    .groupby('ticker')
    .resample('ME')['return']
    .apply(lambda x: (1 + x).prod() - 1)
    .reset_index()
)
```

Polars handles time-based aggregation through explicit grouping by time periods:

```
# Polars: group by ticker and month
monthly_panel_polars = (
    data_panel_polars
    .group_by(['ticker', pl.col('date').dt.month().alias('month')])
    .agg(
        ((1 + pl.col('return')).product() - 1).alias('monthly_return')
    )
)

# More sophisticated: group by ticker and dynamic time windows
monthly_panel_polars_alt = (
    data_panel_polars
    .sort(['ticker', 'date'])
    .group_by_dynamic(
        'date',
        every='1mo',
        by='ticker'
    )
    .agg([
        ((1 + pl.col('return')).product() - 1).alias('monthly_return'),
        pl.col('return').std().alias('volatility'),
        pl.col('return').count().alias('days')
    ])
)
```

The `group_by_dynamic` method in Polars provides powerful time-based grouping with clear syntax for the window specification.

14.2.6. Performance Considerations

For small to medium datasets (thousands to low millions of rows), both pandas and Polars perform well. Differences become significant with larger datasets or complex operations:

- **Pandas** uses single-threaded execution for most operations. Custom `apply` functions run sequentially in Python, which can be slow for large groups or expensive functions.
- **Polars** uses parallel execution by default and query optimization. Operations expressed using Polars expressions (rather than Python functions) run much faster, often 10-100x faster than pandas on large datasets.

For production pipelines or research involving millions of rows, Polars's performance advantages justify the learning curve. For interactive analysis and moderate-sized datasets, pandas's ecosystem maturity and flexibility often win.

14.2.7. Grouped Transformations

Aggregation reduces groups to single values, but sometimes you want to transform data within groups while preserving the original shape. Common examples include:

- Demeaning returns within sectors (excess returns over sector average)
- Computing lagged observations in panel data
- Normalizing values within groups
- Computing ranks within categories
- Forward-filling missing data within securities

Pandas Transform

```
# Demean returns within each sector
data['excess_return'] = (
    data.groupby('sector')['return']
    .transform(lambda x: x - x.mean())
)

# Standardize returns within each ticker
data_panel['standardized_return'] = (
    data_panel.groupby('ticker')['return']
    .transform(lambda x: (x - x.mean()) / x.std())
)

# Rank returns within each date (cross-sectional ranks)
data_panel['return_rank'] = (
    data_panel.groupby('date')['return']
    .rank(pct=True)
)
```

Transform operations return a Series with the same index as the original data, allowing direct assignment to new columns.

Polars Window Functions

Polars uses window functions (similar to SQL) for grouped transformations:

```
# Demean returns within each sector
data_polars_with_excess = data_polars.with_columns(
    (pl.col('return') - pl.col('return').mean().over('sector')).alias('excess_return')  ①
)

# Standardize returns within each ticker
data_panel_polars_standardized = data_panel_polars.with_columns(
    (
```

```
        (pl.col('return') - pl.col('return').mean().over('ticker')) /
        pl.col('return').std().over('ticker')
    ).alias('standardized_return')                                          ②
)

# Rank returns within each date
data_panel_polars_ranked = data_panel_polars.with_columns(
    pl.col('return').rank().over('date').alias('return_rank')               ③
)
```

① The `.over('sector')` computes the mean within each sector group, similar to SQL window functions.
② Z-score normalization: subtract the group mean and divide by the group standard deviation.
③ Cross-sectional ranking: rank returns among all stocks on each date.

The `.over()` syntax specifies the grouping for the window function. This approach is more explicit than pandas transform and integrates naturally with Polars's expression system.

14.3. Common Pitfalls in Financial Data

Financial data has specific characteristics that create traps for the unwary. Two problems—look-ahead bias and survivorship bias—are particularly insidious because they can produce plausible-looking results that are completely invalid. Understanding these issues and designing your data structures to avoid them is essential for credible empirical work.

14.3.1. Look-Ahead Bias

Look-ahead bias occurs when your analysis uses information that would not have been available at the time a decision would have been made. This is surprisingly easy to do accidentally, and it typically makes strategies appear more profitable than they actually would have been.

Example 1: Using Future Data for Current Decisions

Consider computing a moving average of returns to generate trading signals:

```
# WRONG: Look-ahead bias
returns = pd.Series(
    [0.01, -0.02, 0.03, -0.01, 0.02],
    index=pd.date_range('2024-01-01', periods=5, freq='D')
)

# This looks innocent but is wrong!
returns['ma_5'] = returns.rolling(window=5, center=True).mean()
```

The `center=True` parameter makes each moving average value include both past and future returns. A trading signal based on the 2024-01-03 moving average would use returns through 2024-01-05, which wouldn't be known on 2024-01-03. The correct approach:

```
# CORRECT: Only use past data
returns_with_ma = pd.DataFrame({
    'return': returns,
    'ma_5': returns.rolling(window=5).mean()                                    ①
})

# Generate signal based on past information
returns_with_ma['signal'] = (
    returns_with_ma['return'].shift(1) > returns_with_ma['ma_5'].shift(1)       ②
).astype(int)
```

① Default rolling window uses only current and past observations (no future data).
② The `.shift(1)` ensures today's signal uses only yesterday's values—information available at market open.

Example 2: Point-in-Time Data Issues

Financial datasets are often revised. A company's book value reported in 2024 financial statements might differ from what was reported in earlier vintages due to restatements. Academic databases like Compustat provide "point-in-time" datasets that preserve what was known at each historical date.

```
# Problematic: Using latest restated values
fundamentals = pd.DataFrame({
    'ticker': ['AAPL', 'AAPL', 'AAPL'],
    'report_date': pd.to_datetime(['2023-09-30', '2023-12-31', '2024-03-31']),
    'book_value': [150e9, 155e9, 160e9]  # Current restated values
})

# Computing book-to-market ratio using restated book values
# would create look-ahead bias if values were adjusted retroactively

# Better: Use point-in-time database or verify no retroactive changes
# Query data as it existed at each decision point
```

The safest approach is to use datasets explicitly designed to avoid this issue (CRSP/Compustat point-in-time, FactSet's point-in-time fundamentals) or carefully verify that your data source doesn't include retroactive revisions.

Example 3: Survivorship Bias in Filtering

Filtering data before creating historical samples can inadvertently create look-ahead bias:

```
# WRONG: Filtering on current characteristics
# Get list of large-cap stocks as of 2024
large_caps_2024 = get_stocks_by_market_cap(min_cap=10e9, date='2024-01-01')

# Use this list to analyze returns from 2020-2024
historical_returns = get_returns(
    tickers=large_caps_2024,
    start_date='2020-01-01',
    end_date='2024-01-01'
)
# This is wrong! Uses 2024 information to select 2020 sample

# CORRECT: Filter at each point in time
def get_large_cap_returns(start_date, end_date, min_cap):
    """Get returns for stocks that were large-cap at each date."""
    all_dates = pd.date_range(start_date, end_date, freq='D')
    results = []

    for date in all_dates:
        # Get stocks that qualified on this date
        eligible = get_stocks_by_market_cap(min_cap=min_cap, date=date)
        # Get returns for these stocks on this date
        returns = get_returns(tickers=eligible, start_date=date, end_date=date)
        results.append(returns)

    return pd.concat(results)
```

The correct approach checks eligibility at each date, allowing the composition to change over time as stocks grow or shrink.

⚠ Detecting Look-Ahead Bias

Ask these questions about any empirical analysis:

1. **Could I have computed this value in real-time?** If your calculation requires future data, it's look-ahead bias.
2. **Does the analysis use the latest version of the data?** If you're using restated fundamentals or revised economic data, verify those revisions were available when decisions would have been made.
3. **Are sample filters time-varying?** Any filter based on characteristics (market cap, industry, etc.) should be applied using values from that point in time, not current values.
4. **Are there suspicious shifts?** If lagged variables are used to predict returns, ensure lags are implemented correctly with `.shift()` operations.

14.3.2. Survivorship Bias

Survivorship bias occurs when your dataset includes only entities that survived until the sample endpoint, excluding those that disappeared (delisted stocks, defunct funds, closed portfolios). This creates artificially optimistic results because you're analyzing a selected sample of winners.

Example 1: Stock Return Analysis

Consider analyzing stock returns from 2020-2024 using a current stock list:

```
# WRONG: Current stock universe
current_stocks = ['AAPL', 'MSFT', 'GOOGL', 'AMZN', 'NVDA']
returns = get_returns(tickers=current_stocks, start='2020-01-01', end='2024-01-01')

# Average return calculation
mean_return = returns.mean()
# This is biased upward! Only includes stocks that survived and remained liquid
```

These are all large, successful companies. Hundreds of stocks that existed in 2020 but delisted due to bankruptcy, acquisition, or poor performance are missing. The average return will be substantially higher than the true market average.

The solution requires a database that includes delisted securities:

```
# CORRECT: Include delisted stocks
# Using a complete database (e.g., CRSP with delisting codes)
all_stocks = get_stocks(start='2020-01-01', include_delisted=True)
complete_returns = get_returns(
    tickers=all_stocks,
    start='2020-01-01',
    end='2024-01-01'
)

# Handle delisting returns
# CRSP provides delisting returns accounting for final liquidation values
complete_returns['total_return'] = (
    complete_returns['return'].fillna(0) +
    complete_returns['delisting_return'].fillna(0)
)

mean_return_unbiased = complete_returns['total_return'].mean()
```

Academic databases like CRSP explicitly address survivorship bias by maintaining historical records of all listed stocks, including delisting codes and final returns.

Example 2: Fund Performance Analysis

Mutual fund and hedge fund databases are notorious for survivorship bias. Failed or poorly performing funds often stop reporting, creating databases that over-represent successful funds:

```
# Analyzing fund performance with survivorship bias
funds_database = get_fund_returns(database='current_funds')  # Only active funds

# Average alpha appears positive
average_alpha = compute_alpha(funds_database)
# Misleading! Unsuccessful funds that closed are missing

# Better approach: Use survivorship-bias-free database
funds_complete = get_fund_returns(database='complete_with_defunct')

# Explicitly track fund status
active_funds = funds_complete[funds_complete['status'] == 'active']
defunct_funds = funds_complete[funds_complete['status'] == 'defunct']

# Compare performance
print(f"Active funds average return: {active_funds['return'].mean():.2%}")
print(f"Defunct funds average return: {defunct_funds['return'].mean():.2%}")
print(f"All funds average return: {funds_complete['return'].mean():.2%}")
```

Studies have found that survivorship bias can inflate reported mutual fund returns, a massive distortion in performance measurement.

⚠ Detecting Survivorship Bias

Guard against survivorship bias by:

1. **Using complete databases:** CRSP, Compustat, or commercial providers that maintain historical records of delisted securities.
2. **Checking delisting treatment:** Verify how your data source handles stocks that stopped trading. Are delisting returns included?
3. **Comparing universe sizes:** If analyzing 2020-2024 data in 2024, check how many securities existed in 2020 versus 2024. A large decline might indicate missing delistings.
4. **Examining data provider methodology:** Read documentation about survivorship bias treatment. Reputable providers are explicit about whether datasets include defunct entities.

14.4. Summary

Effective financial data analysis requires understanding how data structure shapes what questions you can answer and what mistakes you might make. Keys and identifiers determine how observations are uniquely

identified; choosing whether to use regular columns or special index structures affects code clarity and operation convenience. Grouping and aggregation operations are central to computing statistics within subsets, whether sector averages, monthly volatility, or portfolio characteristics. Both pandas and Polars provide powerful capabilities, with pandas offering flexibility and ecosystem maturity while Polars delivers better performance through query optimization and parallel execution.

Look-ahead bias and survivorship bias represent the most serious threats to validity in financial empirical work. Look-ahead bias uses information that wouldn't have been available at decision time, making strategies appear more profitable than possible. Survivorship bias analyzes only entities that survived to the sample endpoint, excluding failures and creating upward-biased performance measures. Both require careful attention to data structure, filtering logic, and temporal alignment.

Key practices for robust financial data analysis include:

- Explicitly validate that composite keys uniquely identify observations
- Choose index structures based on clarity and operation convenience, not capability
- Use time-aware operations (resampling, shifting) carefully to maintain temporal integrity
- Express grouped calculations explicitly, making aggregation logic clear
- Use complete databases that include delisted securities and defunct entities
- Document assumptions about data availability and point-in-time values

These skills form the foundation for more advanced topics: joining datasets, reshaping between long and wide formats, and handling complex temporal relationships. The ability to structure data correctly and aggregate it appropriately without introducing bias separates valid empirical research from plausible-looking but incorrect analyses.

15. Reshaping Data

Financial data rarely arrives in the exact format you need for analysis. Stock returns might come as one row per stock-date combination, but you need a matrix with dates as rows and stocks as columns. Or you might receive a wide table of quarterly earnings across columns that needs to be converted into a tidy long format for regression analysis. These transformations—collectively known as "reshaping"—are fundamental operations in empirical finance.

This chapter covers the essential reshaping operations you'll use repeatedly: converting between long and wide formats, pivoting and melting data, and stacking and unstacking hierarchical indices. We'll work through practical examples using both pandas and Polars, showing how each library handles these transformations.

15.1. Long vs. Wide Formats

The distinction between long and wide formats is fundamental to data analysis. Understanding when to use each format, and how to convert between them, will save you countless hours of frustration.

15.1.1. What Are Long and Wide Formats?

Wide format stores each entity as a row and each time period (or category) as a separate column. This is how many people naturally think about panel data—it looks like a spreadsheet:

```
date        AAPL    MSFT    GOOGL
2024-01-01  0.012   0.008   0.015
2024-01-02  -0.005  0.003   0.002
2024-01-03  0.018   0.012   0.010
```

Long format (also called "tidy" or "narrow" format) stores each observation as a separate row, with columns indicating the entity, time period, and value:

```
date        ticker  return
2024-01-01  AAPL    0.012
2024-01-01  MSFT    0.008
2024-01-01  GOOGL   0.015
2024-01-02  AAPL    -0.005
2024-01-02  MSFT    0.003
...
```

The same information appears in both formats, but the organization is completely different.

15.1.2. When to Use Each Format

Use wide format when:

- Performing matrix operations (correlation matrices, portfolio optimization)
- Calculating cross-sectional statistics (comparing values across entities at a point in time)
- Creating visualizations that show multiple series over time
- Working with time series models that expect matrices

Use long format when:

- Running regressions or statistical models (most modeling libraries expect long data)
- Creating grouped or faceted visualizations
- Filtering or aggregating by category
- Following "tidy data" principles for data analysis

In empirical finance, you'll often start with data in one format and need to convert it for your specific analysis. For example, stock return data from CRSP arrives in long format (one row per stock-date), but calculating a covariance matrix requires wide format (dates as rows, stocks as columns).

15.1.3. A Simple Example

Let's create a small dataset of daily stock returns in long format:

```
import pandas as pd
import polars as pl
from datetime import datetime, timedelta

# Create sample return data in long format
dates = pd.date_range('2024-01-01', periods=5, freq='D')
tickers = ['AAPL', 'MSFT', 'GOOGL']

# pandas version
data_long_pd = pd.DataFrame([
    {'date': date, 'ticker': ticker, 'return': round(0.01 * (hash(str(date) + ticker) % 100 - 50) / 100, 4)}
    for date in dates
    for ticker in tickers
])

data_long_pd.head(10)
```

	date	ticker	return
0	2024-01-01	AAPL	0.0004
1	2024-01-01	MSFT	0.0023
2	2024-01-01	GOOGL	0.0034
3	2024-01-02	AAPL	-0.0010
4	2024-01-02	MSFT	-0.0025
5	2024-01-02	GOOGL	-0.0010
6	2024-01-03	AAPL	-0.0041
7	2024-01-03	MSFT	0.0019
8	2024-01-03	GOOGL	-0.0043
9	2024-01-04	AAPL	0.0035

```
# Polars version
data_long_pl = pl.DataFrame([
    {'date': date, 'ticker': ticker, 'return': round(0.01 * (hash(str(date) + ticker) % 100 - 50) / 100, 4)}
    for date in dates
    for ticker in tickers
])

data_long_pl.head(10)
```

date datetime[μs]	ticker str	return f64
2024-01-01 00:00:00	”AAPL”	0.0004
2024-01-01 00:00:00	”MSFT”	0.0023
2024-01-01 00:00:00	”GOOGL”	0.0034
2024-01-02 00:00:00	”AAPL”	-0.001
2024-01-02 00:00:00	”MSFT”	-0.0025
2024-01-02 00:00:00	”GOOGL”	-0.001
2024-01-03 00:00:00	”AAPL”	-0.0041
2024-01-03 00:00:00	”MSFT”	0.0019
2024-01-03 00:00:00	”GOOGL”	-0.0043
2024-01-04 00:00:00	”AAPL”	0.0035

This long format is ideal for most statistical operations. Each row is an observation: a specific stock on a specific date. You can easily filter to specific stocks, calculate summary statistics by ticker, or run panel regressions.

15.2. Pivot, Melt, Stack, Unstack

Python’s data libraries provide several operations for reshaping data. While they might seem redundant at first, each serves a specific purpose and understanding all of them makes you more fluent in data manipulation.

15.2.1. Pivot: Long to Wide

The `pivot` operation transforms long format data into wide format. You specify which column becomes the index (rows), which becomes the columns, and which contains the values to fill the resulting matrix.

pandas syntax:

```
# Convert long to wide: dates as rows, tickers as columns
data_wide_pd = data_long_pd.pivot(
    index='date',                                                    ①
    columns='ticker',                                                ②
    values='return'                                                  ③
)

data_wide_pd
```

① Column that becomes the row index
② Column whose unique values become new columns
③ Column containing the values to fill the matrix

ticker date	AAPL	GOOGL	MSFT
2024-01-01	0.0004	0.0034	0.0023
2024-01-02	-0.0010	-0.0010	-0.0025
2024-01-03	-0.0041	-0.0043	0.0019
2024-01-04	0.0035	0.0039	0.0020
2024-01-05	-0.0026	-0.0049	-0.0025

Notice that `pivot` creates a DataFrame where:

- Each unique date becomes a row
- Each unique ticker becomes a column
- Return values fill the cells

Polars syntax:

```
# Polars uses pivot differently - need to specify aggregation
data_wide_pl = data_long_pl.pivot(
    index='date',
    columns='ticker',
    values='return'
)

data_wide_pl
```

```
/var/folders/jr/cn9h86ld68qb5rtvs9gsb1vr0000gn/T/ipykernel_60752/105412148.py:2: DeprecationWarning: the argument `columns` for `Da
  data_wide_pl = data_long_pl.pivot(
```

date datetime[μs]	AAPL f64	MSFT f64	GOOGL f64
2024-01-01 00:00:00	0.0004	0.0023	0.0034
2024-01-02 00:00:00	-0.001	-0.0025	-0.001
2024-01-03 00:00:00	-0.0041	0.0019	-0.0043
2024-01-04 00:00:00	0.0035	0.002	0.0039
2024-01-05 00:00:00	-0.0026	-0.0025	-0.0049

Polars' `pivot` is similar but requires you to think about aggregation from the start. If multiple rows could map to the same cell (same date and ticker), you must specify how to combine them (sum, mean, first, etc.).

15.2.2. Handling Duplicate Entries

One common issue with pivoting is duplicate entries. What happens if your data has multiple return observations for the same stock on the same date?

```
# Create data with duplicates
data_with_dups_pd = pd.DataFrame([
    {'date': '2024-01-01', 'ticker': 'AAPL', 'return': 0.01},
    {'date': '2024-01-01', 'ticker': 'AAPL', 'return': 0.02},  # duplicate!
    {'date': '2024-01-01', 'ticker': 'MSFT', 'return': 0.03},
])

# pandas pivot will fail with duplicates
try:
    data_with_dups_pd.pivot(index='date', columns='ticker', values='return')
except ValueError as e:
    print(f"Error: {e}")
```

```
Error: Index contains duplicate entries, cannot reshape
```

When pandas encounters duplicates, it raises an error. You have two options:

1. Use `pivot_table` with an aggregation function
2. Clean your data first to remove duplicates

```
# Option 1: Use pivot_table with aggregation
wide_with_agg_pd = data_with_dups_pd.pivot_table(
    index='date',
    columns='ticker',
    values='return',
    aggfunc='mean'  # or 'sum', 'first', 'last', etc.
)

wide_with_agg_pd
```

ticker date	AAPL	MSFT
2024-01-01	0.015	0.03

Polars requires you to specify aggregation from the start, so duplicates are handled automatically:

```
data_with_dups_pl = pl.DataFrame([
    {'date': '2024-01-01', 'ticker': 'AAPL', 'return': 0.01},
    {'date': '2024-01-01', 'ticker': 'AAPL', 'return': 0.02},
    {'date': '2024-01-01', 'ticker': 'MSFT', 'return': 0.03},
])

# Polars pivot with aggregation
wide_with_agg_pl = data_with_dups_pl.pivot(
    index='date',
    columns='ticker',
    values='return',
    aggregate_function='mean'
)

wide_with_agg_pl
```

```
/var/folders/jr/cn9h86ld68qb5rtvs9gsb1vr0000gn/T/ipykernel_60752/556176386.py:8: DeprecationWarning: the argument `columns` for
  wide_with_agg_pl = data_with_dups_pl.pivot(
```

date str	AAPL f64	MSFT f64
”2024-01-01”	0.015	0.03

15.2.3. Melt/Unpivot: Wide to Long

The melt operation (called `unpivot` in Polars) is the inverse of pivot—it transforms wide format data back into long format. You specify which columns to keep as identifiers and which to "melt" into a single column.

pandas syntax:

```
# Reset index to make date a regular column
data_wide_pd_reset = data_wide_pd.reset_index()

# Melt back to long format
data_melted_pd = data_wide_pd_reset.melt(
    id_vars=['date'],                                    ①
    value_vars=['AAPL', 'MSFT', 'GOOGL'],                ②
    var_name='ticker',                                   ③
    value_name='return'                                  ④
)

data_melted_pd.head(10)
```

① Columns to keep as identifiers
② Columns to melt (optional—melts all others if not specified)
③ Name for the new column containing original column names
④ Name for the new column containing values

	date	ticker	return
0	2024-01-01	AAPL	0.0004
1	2024-01-02	AAPL	-0.0010
2	2024-01-03	AAPL	-0.0041
3	2024-01-04	AAPL	0.0035
4	2024-01-05	AAPL	-0.0026
5	2024-01-01	MSFT	0.0023
6	2024-01-02	MSFT	-0.0025
7	2024-01-03	MSFT	0.0019
8	2024-01-04	MSFT	0.0020
9	2024-01-05	MSFT	-0.0025

Polars syntax:

```
# Polars uses unpivot (melt is deprecated)
data_melted_pl = data_wide_pl.unpivot(
    index=['date'],                                      ①
    on=['AAPL', 'MSFT', 'GOOGL'],                        ②
    variable_name='ticker',
```

```
    value_name='return'
)

data_melted_pl.head(10)
```

① Columns to keep as identifiers (called `index` in Polars)
② Columns to unpivot (called `on` in Polars)

date datetime[μs]	ticker str	return f64
2024-01-01 00:00:00	”AAPL”	0.0004
2024-01-02 00:00:00	”AAPL”	-0.001
2024-01-03 00:00:00	”AAPL”	-0.0041
2024-01-04 00:00:00	”AAPL”	0.0035
2024-01-05 00:00:00	”AAPL”	-0.0026
2024-01-01 00:00:00	”MSFT”	0.0023
2024-01-02 00:00:00	”MSFT”	-0.0025
2024-01-03 00:00:00	”MSFT”	0.0019
2024-01-04 00:00:00	”MSFT”	0.002
2024-01-05 00:00:00	”MSFT”	-0.0025

15.2.4. Stack and Unstack: Index-Based Reshaping

While `pivot` and `melt` work with columns, `stack` and `unstack` work with index levels. They’re particularly useful when you have MultiIndex DataFrames (hierarchical indices with multiple levels).

Stack: Moves a column level into the row index (wide to long)

Unstack: Moves a row index level into the columns (long to wide)

```
# Create a MultiIndex DataFrame
# Set both date and ticker as index
data_multi_pd = data_long_pd.set_index(['date', 'ticker'])

data_multi_pd.head()
```

date	ticker
2024-01-01	AAPL
	MSFT
	GOOGL
2024-01-02	AAPL
	MSFT

date	ticker	ret

```
# Unstack: move ticker from index to columns (long to wide)
data_unstacked_pd = data_multi_pd.unstack(level='ticker')

data_unstacked_pd
```

ticker date	return AAPL	GOOGL	MSFT
2024-01-01	0.0004	0.0034	0.0023
2024-01-02	-0.0010	-0.0010	-0.0025
2024-01-03	-0.0041	-0.0043	0.0019
2024-01-04	0.0035	0.0039	0.0020
2024-01-05	-0.0026	-0.0049	-0.0025

```
# Stack: move ticker from columns back to index (wide to long)
data_stacked_pd = data_unstacked_pd.stack(level='ticker')

data_stacked_pd.head()
```

```
/var/folders/jr/cn9h86ld68qb5rtvs9gsb1vr0000gn/T/ipykernel_60752/4172318071.py:2: FutureWarning: The previous implementation of sta
  data_stacked_pd = data_unstacked_pd.stack(level='ticker')
```

date	ticker	ret
2024-01-01	AAPL	0.0
	GOOGL	0.0
	MSFT	0.0
2024-01-02	AAPL	-0.0
	GOOGL	-0.0

The key difference between pivot/melt and stack/unstack:

- **pivot/melt**: Work with regular columns, create new structure from scratch
- **stack/unstack**: Work with existing MultiIndex structure, move levels between index and columns

Polars doesn't have direct equivalents to stack/unstack because it doesn't use MultiIndex. Instead, you would use combinations of `pivot`, `melt`, `group_by`, and `join` to achieve the same results.

15.3. Summary

Reshaping data is a fundamental skill in empirical finance. The key operations are:

- **Long vs. Wide**: Choose the format appropriate for your analysis
  - Long for modeling and tidy operations
  - Wide for matrix operations and cross-sectional analysis
- **Pivot/Melt**: Convert between long and wide formats
  - `pivot`: long ⟶ wide (create matrix from observations)
  - `melt`/`unpivot`: wide ⟶ long (create observations from matrix)
  - Handle duplicates carefully with aggregation
- **Stack/Unstack**: Work with MultiIndex structures
  - `unstack`: move index level to columns
  - `stack`: move column level to index
  - Primarily useful in pandas (Polars doesn't use MultiIndex)

The specific syntax differs between pandas and Polars, but the concepts are universal. pandas uses `pivot`, `pivot_table`, `melt`, `stack`, and `unstack`. Polars uses `pivot` and `unpivot`, relying more on grouped operations for reshaping tasks.

Practice these operations until they become second nature. You'll use them constantly in empirical finance, and fluency with reshaping will make your analyses both faster and more reliable.

16. Joins and Merges

Combining datasets is fundamental to empirical finance. Stock returns live in one database (CRSP), accounting data in another (Compustat), analyst forecasts in a third (I/B/E/S). Real research requires merging these sources, and doing it wrong can silently corrupt your results.

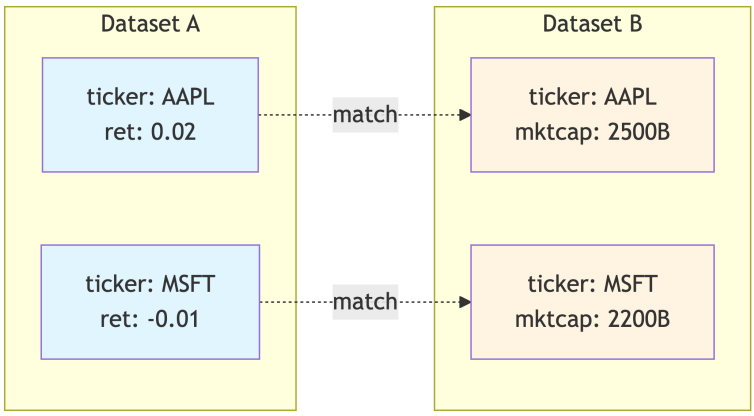
This chapter develops the theory and practice of joins. We start with the conceptual framework—what happens when you match rows between datasets—then move to specific merge operations. Special attention goes to asof joins, essential for time-series data with mismatched timestamps. We close with diagnostic techniques to catch the merge errors that plague real-world research.

16.1. Join Types: Cardinality Matters

Before learning merge syntax, understand what happens to row counts. The relationship between keys in your datasets determines whether you end up with more rows, fewer rows, or the same number. Getting this wrong is catastrophic—you might duplicate observations, lose data, or create spurious patterns.

16.1.1. One-to-One Joins

Each key appears at most once in both datasets. This is the simplest case: rows either match or they don't.



One-to-one join: each key matches at most once

Example: merging monthly portfolio returns with portfolio characteristics. Each portfolio-month appears once in each dataset.

```python
import pandas as pd
import polars as pl
from datetime import date

# Monthly portfolio returns
returns = pd.DataFrame({
    'portfolio': ['growth', 'value', 'momentum'],
    'month': ['2024-01', '2024-01', '2024-01'],
    'return': [0.025, -0.010, 0.035]
})

# Portfolio characteristics (computed separately)
characteristics = pd.DataFrame({
    'portfolio': ['growth', 'value', 'momentum'],
    'month': ['2024-01', '2024-01', '2024-01'],
    'avg_size': [5000, 3000, 4000],
    'avg_bm': [0.5, 2.5, 1.2]
})

# One-to-one merge
merged = returns.merge(
    characteristics,
    on=['portfolio', 'month'],                    # ①
    validate='one_to_one'                         # ②
)
merged
```

① Columns that must match exactly in both DataFrames.
② Raises `MergeError` if duplicate keys exist in either DataFrame.

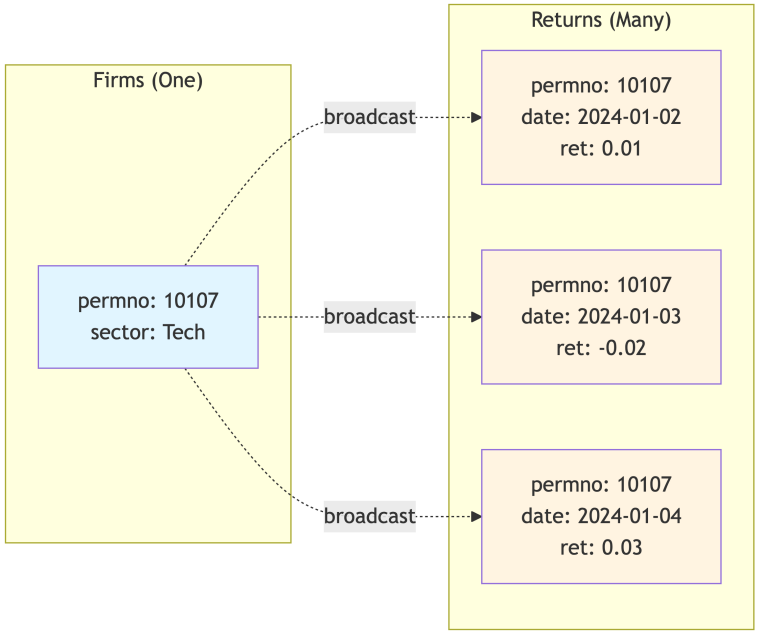
	portfolio	month	return	avg_size	avg_bm
0	growth	2024-01	0.025	5000	0.5
1	value	2024-01	-0.010	3000	2.5
2	momentum	2024-01	0.035	4000	1.2

! Always Validate Cardinality

Use `validate='one_to_one'` in pandas or check row counts before and after. One-to-one merges should never change the number of rows (except for unmatched keys with inner joins). If row count changes unexpectedly, you have duplicate keys somewhere.

16.1.2. One-to-Many Joins

Keys are unique in one dataset but repeated in the other. This expands rows from the unique side.



One-to-many join: firm-level data matched to daily returns

Example: merging firm characteristics (one row per firm) with daily returns (many rows per firm).

```
# Firm characteristics (one row per firm)
firms = pd.DataFrame({
    'permno': [10107, 10107, 14593],
    'year': [2023, 2024, 2024],
    'sector': ['Tech', 'Tech', 'Finance'],
    'headquarters': ['CA', 'CA', 'NY']
})

# Daily returns (many rows per firm-year)
returns = pd.DataFrame({
    'permno': [10107, 10107, 10107, 14593, 14593],
    'year': [2024, 2024, 2024, 2024, 2024],
    'date': ['2024-01-02', '2024-01-03', '2024-01-04',
             '2024-01-02', '2024-01-03'],
    'ret': [0.01, -0.02, 0.03, 0.005, -0.001]
})

# One-to-many merge: firm characteristics broadcast to each return
merged = returns.merge(
```

```
    firms,
    on=['permno', 'year'],
    validate='many_to_one'  # Ensures firms has unique keys
)
merged
```

	permno	year	date	ret	sector	headquarters
0	10107	2024	2024-01-02	0.010	Tech	CA
1	10107	2024	2024-01-03	-0.020	Tech	CA
2	10107	2024	2024-01-04	0.030	Tech	CA
3	14593	2024	2024-01-02	0.005	Finance	NY
4	14593	2024	2024-01-03	-0.001	Finance	NY

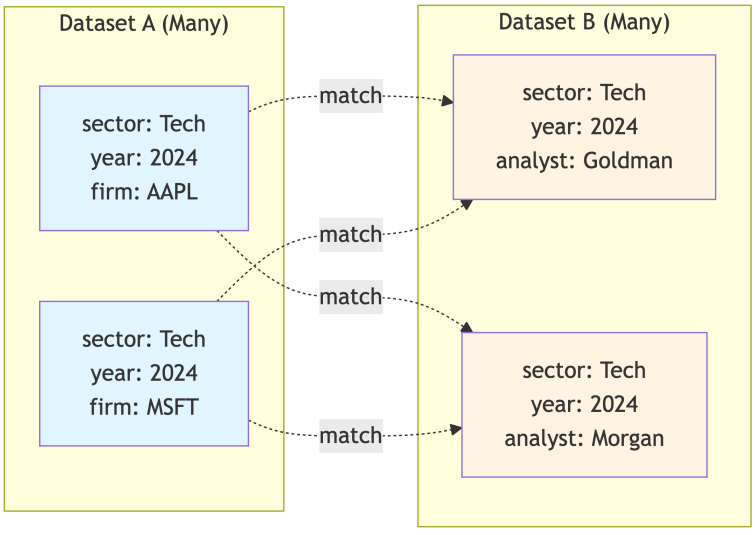
Notice how `sector` and `headquarters` repeat for each date. This is broadcasting: the single firm row expands to match multiple return rows.

Watch for Unintended Duplicates

If you think you have a one-to-many join but the "one" side has duplicates, you'll get a many-to-many join instead. This multiplies rows in ways that corrupt analyses. Always validate.

16.1.3. Many-to-Many Joins

Keys repeat in both datasets. Every combination of matching keys produces a row. This explodes row counts and is rarely what you want.



Many-to-many join: combinatorial explosion

Example: joining firm-level data to analyst coverage by sector-year creates all combinations.

```
# Firms in each sector-year
firms = pd.DataFrame({
    'sector': ['Tech', 'Tech', 'Finance'],
    'year': [2024, 2024, 2024],
    'ticker': ['AAPL', 'MSFT', 'JPM'],
    'return': [0.25, 0.30, 0.15]
})

# Analyst reports by sector-year (multiple analysts per sector)
analysts = pd.DataFrame({
    'sector': ['Tech', 'Tech', 'Finance'],
    'year': [2024, 2024, 2024],
    'analyst': ['Goldman', 'Morgan', 'Citi'],
    'recommendation': ['Buy', 'Hold', 'Buy']
})

# Many-to-many merge: DANGER!
merged = firms.merge(analysts, on=['sector', 'year'])
merged
```

	sector	year	ticker	return	analyst	recommendation
0	Tech	2024	AAPL	0.25	Goldman	Buy
1	Tech	2024	AAPL	0.25	Morgan	Hold
2	Tech	2024	MSFT	0.30	Goldman	Buy
3	Tech	2024	MSFT	0.30	Morgan	Hold
4	Finance	2024	JPM	0.15	Citi	Buy

Notice how the row count exploded:

```
print(f"Original firms: {len(firms)}, analysts: {len(analysts)}, merged: {len(merged)}")
```

```
Original firms: 3, analysts: 3, merged: 5
```

Many-to-Many is Usually Wrong

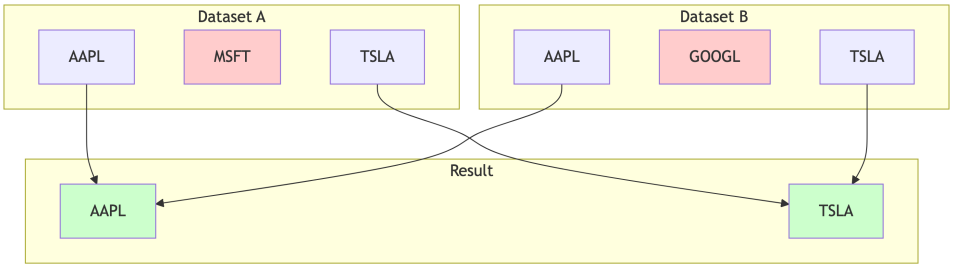
In finance, many-to-many joins typically indicate a modeling error. You probably need to aggregate one dataset first (e.g., average analyst recommendations by sector-year) or use a more specific join key. Pandas allows many-to-many by default; consider this a bug, not a feature.

16.2. Merge Operations: Which Rows Survive?

Join type (one-to-one, etc.) describes cardinality. Merge operation (inner, outer, left, right) describes *which rows to keep* when keys don't match perfectly.

16.2.1. Inner Join: Intersection Only

Keep only rows where keys match in both datasets. This is conservative—you lose any observation that doesn't have a partner—but ensures every row has complete data.



Inner join: only matching keys survive

Example: merging returns with accounting data. Only firm-years with both survive.

```
# Stock returns
returns = pd.DataFrame({
    'ticker': ['AAPL', 'MSFT', 'TSLA'],
    'year': [2023, 2023, 2023],
    'return': [0.45, 0.35, 0.10]
})

# Accounting data (missing MSFT)
accounting = pd.DataFrame({
    'ticker': ['AAPL', 'TSLA', 'GOOGL'],
    'year': [2023, 2023, 2023],
    'roa': [0.20, 0.05, 0.15],
    'leverage': [0.30, 0.10, 0.05]
})

# Inner join: only AAPL and TSLA have both
inner = returns.merge(accounting, on=['ticker', 'year'], how='inner')
inner
```

	ticker	year	return	roa	leverage
0	AAPL	2023	0.45	0.20	0.3
1	TSLA	2023	0.10	0.05	0.1

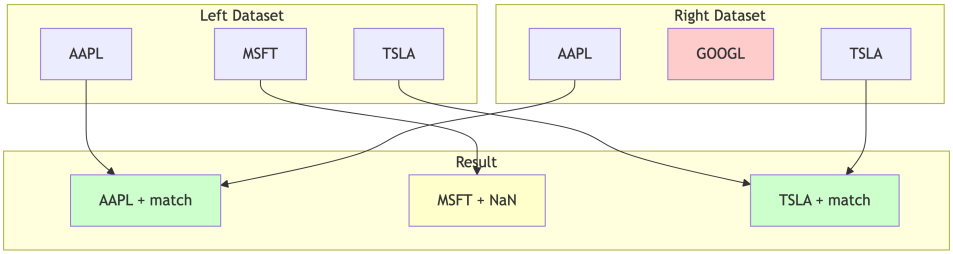
	ticker	year	return	roa	leverage

When to Use Inner Joins

Use inner joins when you *need* data from both sources for your analysis. For example, computing correlations between returns and accounting ratios requires both variables. The sample restriction is a feature, not a bug—you're explicitly limiting to observations where analysis is possible.

16.2.2. Left Join: Keep All from Left Dataset

Keep every row from the left dataset, adding data from the right where available. Unmatched right rows disappear. Missing values (NaN) fill columns from the right dataset where no match exists.



Left join: all left-side rows survive, right-side data added where available

Example: keeping all returns, adding accounting data where available.

```
# Left join: keep all returns
left = returns.merge(accounting, on=['ticker', 'year'], how='left')
left
```

	ticker	year	return	roa	leverage
0	AAPL	2023	0.45	0.20	0.3
1	MSFT	2023	0.35	NaN	NaN
2	TSLA	2023	0.10	0.05	0.1

Left Joins Preserve Sample

Use left joins when the left dataset defines your sample. For example, if you've carefully constructed a set of stocks for analysis, use those as the left dataset and merge in additional characteristics. This ensures you keep your full sample even if some characteristics are missing.

16.2.3. Right Join: Keep All from Right Dataset

The mirror of left join. Keep every row from the right dataset, add data from left where available. In practice, just swap your datasets and use a left join—right joins exist for symmetry but rarely clarify code.

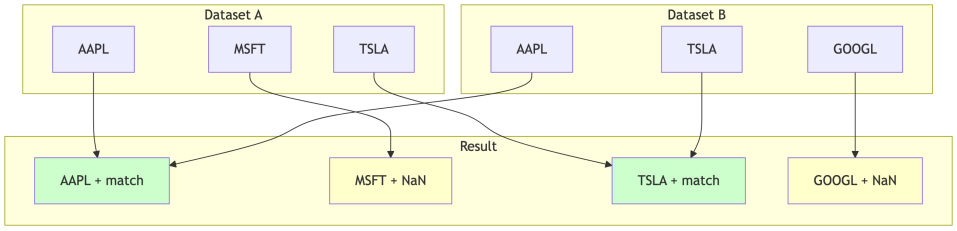
```
# Right join: keep all accounting observations
right = returns.merge(accounting, on=['ticker', 'year'], how='right')
right
```

	ticker	year	return	roa	leverage
0	AAPL	2023	0.45	0.20	0.30
1	TSLA	2023	0.10	0.05	0.10
2	GOOGL	2023	NaN	0.15	0.05

> **Prefer Left Joins for Clarity**
>
> Write `accounting.merge(returns, how='left')` instead of `returns.merge(accounting, how='right')`. Reading left-to-right matches execution order and makes the priority explicit: you're starting with accounting data and adding returns.

16.2.4. Outer Join: Keep Everything

Keep all rows from both datasets. This maximizes sample size but can create large numbers of missing values.



Outer join: all rows from both datasets survive

Example: keeping all tickers from either dataset.

```
# Outer join: keep everything
outer = returns.merge(accounting, on=['ticker', 'year'], how='outer')
outer
```

	ticker	year	return	roa	leverage
0	AAPL	2023	0.45	0.20	0.30

	ticker	year	return	roa	leverage
1	GOOGL	2023	NaN	0.15	0.05
2	MSFT	2023	0.35	NaN	NaN
3	TSLA	2023	0.10	0.05	0.10

Outer Joins Hide Problems

Outer joins are dangerous because they succeed even when keys match poorly. You might merge stock returns with bond yields on date, get mostly NaN, but never realize the datasets use different date formats. Always inspect outer join results carefully and report match rates.

16.2.5. Polars Merge Syntax

Polars uses similar concepts but different method names. The key difference: Polars is more explicit about join types and discourages many-to-many joins by default.

```
# Same data in Polars
returns_pl = pl.DataFrame({
    'ticker': ['AAPL', 'MSFT', 'TSLA'],
    'year': [2023, 2023, 2023],
    'return': [0.45, 0.35, 0.10]
})

accounting_pl = pl.DataFrame({
    'ticker': ['AAPL', 'TSLA', 'GOOGL'],
    'year': [2023, 2023, 2023],
    'roa': [0.20, 0.05, 0.15],
    'leverage': [0.30, 0.10, 0.05]
})

# Inner join
inner_pl = returns_pl.join(
    accounting_pl,
    on=['ticker', 'year'],
    how='inner'
)
inner_pl
```

ticker str	year i64	return f64	roa f64	leverage f64
”AAPL”	2023	0.45	0.2	0.3

ticker str	year i64	return f64	roa f64	leverage f64
”TSLA”	2023	0.1	0.05	0.1

```
# Left join
left_pl = returns_pl.join(
    accounting_pl,
    on=['ticker', 'year'],
    how='left'
)
left_pl
```

ticker str	year i64	return f64	roa f64	leverage f64
”AAPL”	2023	0.45	0.2	0.3
”MSFT”	2023	0.35	null	null
”TSLA”	2023	0.1	0.05	0.1

```
# Outer join (called 'full' in Polars)
outer_pl = returns_pl.join(
    accounting_pl,
    on=['ticker', 'year'],
    how='full'
)
outer_pl
```

ticker str	year i64	return f64	ticker_right str	year_right i64	roa f64	leverage f64
”AAPL”	2023	0.45	”AAPL”	2023	0.2	0.3
”TSLA”	2023	0.1	”TSLA”	2023	0.05	0.1
null	null	null	”GOOGL”	2023	0.15	0.05
”MSFT”	2023	0.35	null	null	null	null

Polars also provides a `validate` parameter similar to pandas:

```
# Validate one-to-one
inner_pl = returns_pl.join(
    accounting_pl,
    on=['ticker', 'year'],
    how='inner',
    validate='1:1'  # Raises error if violated
)
```

16.3. Asof Joins for Time-Series Data

Financial data rarely aligns perfectly in time. You have daily stock prices, quarterly earnings announcements, irregular analyst forecast updates. Matching these requires asof joins: find the most recent value as of each timestamp, without requiring exact matches.

16.3.1. The Asof Join Problem

Suppose you want to merge earnings announcement dates with stock returns. Earnings come out on irregular dates (whenever the company reports). You need to tag each daily return with the most recent earnings announcement as of that date. A standard merge fails because announcement dates don't match return dates. An asof join solves this: for each return date, find the most recent announcement date on or before that date.

16.3.2. Asof Join Mechanics

Asof joins require sorted data and directional matching. The syntax varies between pandas and Polars, but the logic is identical:

1. Sort both datasets by time
2. For each row in the left dataset, find the row in the right dataset with the closest timestamp that doesn't exceed the left timestamp (backward-looking)
3. Optionally match exactly on other columns (e.g., ticker)

Example: merging daily returns with quarterly earnings.

```
import numpy as np

# Daily returns (many observations)
returns = pd.DataFrame({
    'ticker': ['AAPL'] * 10,
    'date': pd.date_range('2024-01-01', periods=10, freq='D'),
    'return': np.random.randn(10) * 0.02
})

# Earnings announcements (sparse, irregular)
earnings = pd.DataFrame({
    'ticker': ['AAPL', 'AAPL', 'AAPL'],
    'announce_date': pd.to_datetime(['2023-12-28', '2024-01-03', '2024-01-08']),
    'eps': [1.25, 1.30, 1.35],
    'surprise': [0.02, -0.01, 0.03]
})

# CRITICAL: Sort by time
returns = returns.sort_values('date')                                    ①
```

```
earnings = earnings.sort_values('announce_date')

# Asof join: match each return to most recent earnings
merged = pd.merge_asof(
    returns,
    earnings,
    left_on='date',                                                    ②
    right_on='announce_date',
    by='ticker',                                                       ③
    direction='backward'                                               ④
)

merged[['date', 'return', 'announce_date', 'eps', 'surprise']]
```

① Both DataFrames must be sorted by their time columns before asof join.
② Time columns to match on (can have different names).
③ Additional columns that must match exactly.
④ Use most recent prior value (avoids look-ahead bias).

	date	return	announce_date	eps	surprise
0	2024-01-01	0.011268	2023-12-28	1.25	0.02
1	2024-01-02	0.012405	2023-12-28	1.25	0.02
2	2024-01-03	-0.006166	2024-01-03	1.30	-0.01
3	2024-01-04	-0.010879	2024-01-03	1.30	-0.01
4	2024-01-05	-0.020510	2024-01-03	1.30	-0.01
5	2024-01-06	-0.012813	2024-01-03	1.30	-0.01
6	2024-01-07	0.011687	2024-01-03	1.30	-0.01
7	2024-01-08	0.003256	2024-01-08	1.35	0.03
8	2024-01-09	-0.006405	2024-01-08	1.35	0.03
9	2024-01-10	-0.039662	2024-01-08	1.35	0.03

Notice how the earnings data "fills forward":

- Jan 1-2 use the Dec 28 announcement
- Jan 3-7 use the Jan 3 announcement
- Jan 8+ use the Jan 8 announcement

This is exactly what you want for event studies: tag each return with the most recent earnings information available at that time.

! Asof Joins Require Sorted Data

Both datasets must be sorted by the time column. Pandas doesn't always check this, and unsorted data produces wrong results silently. Always sort explicitly before asof joins.

16.3.3. Asof Join Directions

The `direction` parameter controls which timestamp to match:

- `direction='backward'` (default): Use most recent prior value (as of semantics)
- `direction='forward'`: Use next future value (rare in finance)
- `direction='nearest'`: Use closest value in either direction (dangerous for event studies—can introduce look-ahead bias)

```
# Forward-looking (CAREFUL: creates look-ahead bias)
forward = pd.merge_asof(
    returns,
    earnings,
    left_on='date',
    right_on='announce_date',
    by='ticker',
    direction='forward'  # Use next announcement
)

# Nearest (AVOID for event studies)
nearest = pd.merge_asof(
    returns,
    earnings,
    left_on='date',
    right_on='announce_date',
    by='ticker',
    direction='nearest'  # Use closest announcement
)
```

> **Beware Look-Ahead Bias**
>
> Using `direction='forward'` or `direction='nearest'` can introduce look-ahead bias: your analysis uses information that didn't exist at the time. For example, tagging a January return with a February earnings announcement. This inflates backtested strategy returns and invalidates empirical tests. Stick with `direction='backward'` unless you have a specific reason to do otherwise.

16.3.4. Asof Joins in Polars

Polars provides asof joins through the `join_asof` method. The syntax is cleaner and performance is often better for large datasets.

```
# Same data in Polars
returns_pl = pl.DataFrame({
    'ticker': ['AAPL'] * 10,
    'date': pl.date_range(
        date(2024, 1, 1),
        date(2024, 1, 10),
        interval='1d',
        eager=True
    ),
    'return': np.random.randn(10) * 0.02
})

earnings_pl = pl.DataFrame({
    'ticker': ['AAPL', 'AAPL', 'AAPL'],
    'announce_date': [
        date(2023, 12, 28),
        date(2024, 1, 3),
        date(2024, 1, 8)
    ],
    'eps': [1.25, 1.30, 1.35],
    'surprise': [0.02, -0.01, 0.03]
})

# Asof join in Polars
merged_pl = returns_pl.sort('date').join_asof(
    earnings_pl.sort('announce_date'),
    left_on='date',
    right_on='announce_date',
    by='ticker',
    strategy='backward'  # Equivalent to pandas direction='backward'
)

merged_pl.select(['date', 'return', 'announce_date', 'eps', 'surprise'])
```

```
/var/folders/jr/cn9h86ld68qb5rtvs9gsb1vr0000gn/T/ipykernel_65773/4247637574.py:25: UserWarning: Sortedness of columns cannot be
  merged_pl = returns_pl.sort('date').join_asof(
```

date date	return f64	announce_date date	eps f64	surprise f64
2024-01-01	-0.022832	2023-12-28	1.25	0.02
2024-01-02	-0.011063	2023-12-28	1.25	0.02
2024-01-03	-0.006162	2024-01-03	1.3	-0.01
2024-01-04	0.022057	2024-01-03	1.3	-0.01

date date	return f64	announce_date date	eps f64	surprise f64
2024-01-05	-0.020681	2024-01-03	1.3	-0.01
2024-01-06	0.024423	2024-01-03	1.3	-0.01
2024-01-07	-0.000846	2024-01-03	1.3	-0.01
2024-01-08	0.00782	2024-01-08	1.35	0.03
2024-01-09	0.008991	2024-01-08	1.35	0.03
2024-01-10	0.025194	2024-01-08	1.35	0.03

Polars also supports tolerance (maximum time difference) and handles missing values more explicitly:

```
# Only match if announcement is within 90 days
merged_pl = returns_pl.sort('date').join_asof(
    earnings_pl.sort('announce_date'),
    left_on='date',
    right_on='announce_date',
    by='ticker',
    strategy='backward',
    tolerance='90d'  # NaN if no announcement within 90 days
)
```

16.3.5. Common Asof Join Patterns in Finance

Pattern 1: Point-in-time accounting data

Merge daily returns with quarterly accounting variables. Each return should use the most recent financial statement available at that date.

```
# Daily returns
returns = pd.DataFrame({
    'permno': [10107] * 100,
    'date': pd.date_range('2024-01-01', periods=100, freq='D'),
    'ret': np.random.randn(100) * 0.02
})

# Quarterly accounting (fiscal quarter end dates)
accounting = pd.DataFrame({
    'permno': [10107, 10107, 10107, 10107],
    'datadate': pd.to_datetime(['2023-09-30', '2023-12-31',
                                '2024-03-31', '2024-06-30']),
    'book_equity': [1000, 1100, 1150, 1200],
    'total_assets': [5000, 5200, 5300, 5400]
})
```

```
# Asof join: each return uses most recent accounting data
merged = pd.merge_asof(
    returns.sort_values('date'),
    accounting.sort_values('datadate'),
    left_on='date',
    right_on='datadate',
    by='permno',
    direction='backward'
)
```

Pattern 2: Analyst forecast updates

Analyst forecasts arrive irregularly. Tag each return with the current consensus forecast.

```
# Daily returns
returns = pd.DataFrame({
    'ticker': ['AAPL'] * 30,
    'date': pd.date_range('2024-01-01', periods=30, freq='D'),
    'ret': np.random.randn(30) * 0.02
})

# Analyst forecast updates (irregular)
forecasts = pd.DataFrame({
    'ticker': ['AAPL'] * 5,
    'forecast_date': pd.to_datetime([
        '2023-12-15', '2024-01-05', '2024-01-12',
        '2024-01-20', '2024-01-28'
    ]),
    'consensus_eps': [5.25, 5.30, 5.28, 5.35, 5.40],
    'num_analysts': [25, 26, 27, 28, 29]
})

# Asof join: current forecast as of each date
merged = pd.merge_asof(
    returns.sort_values('date'),
    forecasts.sort_values('forecast_date'),
    left_on='date',
    right_on='forecast_date',
    by='ticker',
    direction='backward'
)
```

Pattern 3: Bid-ask spreads and trades

Match each trade to the prevailing bid-ask spread (within milliseconds).

```
# Trades (exact timestamps)
trades = pd.DataFrame({
    'symbol': ['AAPL'] * 5,
    'trade_time': pd.to_datetime([
        '2024-01-02 09:30:00.123',
        '2024-01-02 09:30:00.456',
        '2024-01-02 09:30:01.789',
        '2024-01-02 09:30:02.012',
        '2024-01-02 09:30:03.345'
    ]),
    'trade_price': [150.25, 150.26, 150.24, 150.25, 150.27],
    'trade_size': [100, 200, 150, 300, 250]
})

# Quote updates (irregular, microsecond timestamps)
quotes = pd.DataFrame({
    'symbol': ['AAPL'] * 4,
    'quote_time': pd.to_datetime([
        '2024-01-02 09:30:00.100',
        '2024-01-02 09:30:00.500',
        '2024-01-02 09:30:02.000',
        '2024-01-02 09:30:03.000'
    ]),
    'bid': [150.24, 150.25, 150.23, 150.26],
    'ask': [150.26, 150.27, 150.25, 150.28]
})

# Asof join: prevailing quote at each trade
merged = pd.merge_asof(
    trades.sort_values('trade_time'),
    quotes.sort_values('quote_time'),
    left_on='trade_time',
    right_on='quote_time',
    by='symbol',
    direction='backward',
    tolerance=pd.Timedelta('1s'),  # Only match within 1 second
    allow_exact_matches=False  # Use quote *before* the trade, not at same instant
)

# Calculate effective spread
merged['spread'] = merged['ask'] - merged['bid']
merged['effective_spread'] = abs(merged['trade_price'] -
                                 (merged['bid'] + merged['ask']) / 2)
```

16.4. Diagnosing and Correcting Merge Errors

Merges fail silently. You run the code, get a DataFrame back, and assume it worked. But maybe 80% of rows didn't match. Maybe you have duplicate keys creating a many-to-many join. Maybe your date formats don't align. The code runs, but your results are garbage.

Run the following checks every time until they become automatic.

16.4.1. Essential checks before and after merging

Before merging:

- **Predict row counts**: Inner join should give ≤ min(left, right) rows; left join should preserve left row count for one-to-one or one-to-many; outer join gives ≥ max(left, right) rows
- **Check for duplicate keys**: Use `df.duplicated(key_cols).sum()` or `validate` parameter to enforce cardinality
- **Verify data types match**: Use `df[key_cols].dtypes` to ensure join keys have compatible types
- **For string keys**: Normalize with `.str.strip().str.upper()` to handle whitespace and case differences
- **For dates**: Ensure both use `datetime64`, same time zone, and same granularity

After merging:

- **Check row count changes**: Unexpected changes indicate duplicate keys or wrong join type
- **Check match rates**: Use `indicator=True` to see how many rows matched (use `merged['_merge'].value_counts()`)
- **Inspect missing values**: NaN in right-side columns after left join indicates failed matches

16.4.2. Using the indicator parameter

The `indicator=True` parameter adds a `_merge` column showing match status:

```
# Re-create sample data for merge diagnostics
returns = pd.DataFrame({
    'ticker': ['AAPL', 'MSFT', 'TSLA'],
    'year': [2023, 2023, 2023],
    'return': [0.45, 0.35, 0.10]
})

accounting = pd.DataFrame({
    'ticker': ['AAPL', 'TSLA', 'GOOGL'],
    'year': [2023, 2023, 2023],
    'roa': [0.20, 0.05, 0.15],
    'leverage': [0.30, 0.10, 0.05]
})

# Left join with indicator
merged = returns.merge(
```

```
    accounting,
    on=['ticker', 'year'],
    how='left',
    indicator=True                                                    ①
)

# Check match rates
merged['_merge'].value_counts()
```

① Adds a `_merge` column showing whether each row matched: `left_only`, `right_only`, or `both`.

```
_merge
both          2
left_only     1
right_only    0
Name: count, dtype: int64
```

```
print(f"Match rate: {(merged['_merge'] == 'both').mean():.1%}")
```

```
Match rate: 66.7%
```

> **!** Investigate low match rates
>
> A match rate below 90% usually indicates problems: incorrect join keys, different identifier schemes (CUSIP vs PERMNO), date format mismatches, inconsistent capitalization, or datasets filtered to different subsets. Don't just accept low match rates—understand why they occurred.

16.4.3. Common merge error patterns

Error	Symptom	Cause	Solution
Cartesian explosion	Row count explodes (10,000x+ more rows)	Duplicate keys in both datasets	Aggregate one dataset or use more specific join keys
Zero matches	All rows are `left_only`	Keys don't overlap at all	Check identifier mapping (CUSIP vs PERMNO), date formats
Type mismatch	Low or zero matches despite overlapping keys	Different data types (int vs float, str vs object)	Convert both to same type before merge
Float precision	Values that should match don't	`150.25` vs `150.25000001`	Round floats or use integer keys

Error	Symptom	Cause	Solution
Time zone mismatch	Asof join produces all NaN	One dataset in UTC, other in local time	Convert both to same time zone
Case/whitespace	Low matches on string keys	"AAPL" vs "aapl" vs " AAPL"	Use `.str.strip().str.upper()`
Unsorted asof join	Wrong matches or NaN	Data not sorted by time column	Always `sort_values()` before asof joins

16.5. Summary and Best Practices

Merging is where most data errors occur in empirical finance. Follow these practices to catch problems early:

1. **Understand cardinality before merging.** Is this one-to-one, one-to-many, or many-to-many? Use `validate` to enforce expectations.
2. **Choose the right merge type.** Inner for complete cases only, left to preserve your sample, outer rarely (and carefully).
3. **Use asof joins for time-series data.** Don't try to match timestamps exactly—use asof joins with `direction='backward'` to avoid look-ahead bias.
4. **Validate every merge.** Check row counts, match rates, duplicates, and data types until these checks become automatic.
5. **Sort before asof joins.** Unsorted data produces silent errors.
6. **Document match rates.** Report how many observations matched in your analysis. Low match rates indicate problems.
7. **Normalize strings.** Strip whitespace, standardize case, remove special characters before merging on text.
8. **Use indicator columns.** Add `indicator=True` to understand what matched and what didn't.
9. **Prefer left joins for clarity.** Write `base.merge(extra, how='left')` instead of `extra.merge(base, how='right')`.
10. **Consider Polars for large datasets.** Polars' lazy evaluation and better error messages can save time on big merges.

The datasets in empirical finance are messy. Companies change tickers, databases use different identifiers, timestamps don't align. Merging correctly requires skepticism, validation, and careful attention to detail. The techniques in this chapter won't eliminate merge errors, but they'll help you catch them before they corrupt your research.

Yadan, Omry. 2019. "Hydra - a Framework for Elegantly Configuring Complex Applications." Github. https://github.com/facebookresearch/hydra.