Empirical Finance with Python

Course Notes for MATH 60207/A @ HEC Montréal

Vincent Grégoire, CFA, Ph.D.

2025-01-01

Table of contents

1 Welcome

This book is aimed at students in the MSc in Finance program at HEC Montréal taking the course *MATH 60207 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.

Tip

This book is also available as a PDF file. Please note, however, that I will add content throught the semester, so you might want to avoid printing the whole document.

Work in Progress

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

Acknowledgments

I am grateful to my colleague Saad Ali Khan for his invaluable feedback and suggestions on the book, and for providing some the content used in the book.

2 About this book

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, making this book a valuable asset for students beginning their journey in Python. Even those with some familiarity with Python will find this book useful, as it offers practical insights into the application of programming in empirical finance.

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. I look forward to guiding you through this exciting journey into the world of empirical finance, statistics, econometrics, and Python programming.

⚠ Incomplete

This book is still far from complete. The structure listed below reflects the target structure, not the current sturcture as it currently is.

2.1 Structure

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

Part 1: Introduction to Python

The first part of the book is devoted to familiarizing you with Python and setting up the necessary coding environment. I begin with instructions on installing Python and understanding its basic syntax. I then move on to introduce various additional tools that are part of the programmer's toolbox, including the terminal, Git, and GitHub. You'll learn to use modern AI tools such as ChatGPT for coding and GitHub Copilot for code generation and assistance.

Part 2: Data Analysis with Python

In the second part, I introduce the main sources of financial data available publicly and by subscription at HEC Montréal, including WRDS, FRED, and SEC Edgar. We will use these data sources throughout the book and explore them using Python libraries such as pandas, matplotlib, and seaborn. You will learn about the main data analysis techniques, including data loading, merging, transformation, and visualization, and how they fit into the data analysis workflow.

Part 3: Financial Econometrics

In the second part, we dive into financial econometrics, starting with the basics of probabilities and statistics. Here, we begin to introduce key Python libraries, starting with scipy and numpy, which provide a foundation for numerical computing in Python.

Part 4: Presenting Results

In the final part of the book, we focus on a key skill in research: presenting results.

We will introduce Quarto, a publishing system that allows you to write in Markdown, a simple language used to format text for scientific documents. We will learn how to mix it with math expressions, how to manage citations and create a bibliography, and how to structure a document with sections, figures, tables, etc. Finally, we will learn how to embed Python code in a Quarto document to embed the analysis directly in the report.

Finally will also discuss how to structure and write a research paper, from the literature review to the discussion of results, including how to create effective tables and figures.

2.2 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 provide 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.

2.3 Tech Stack

This book is structured around a specific set of tools, forming a tech stack 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 3.13 (released in November 2024)
- uv for managing Python versions and environments
- Visual Studio Code for writing code
- Git and GitHub for version control and collaboration
- ChatGPT and GitHub Copilot for coding and assistance
- Quarto for writing the technical content.

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 analysts, researchers, and data scientists 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 Python-based platforms (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. 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 very 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 and documentation 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 you should always check the library's documentation and assess its reputation before using it. 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.

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 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, which I was using until recently, is poetry. Conda and poetry act as both an 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. But 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 (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 that is very 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.

[1]The latest version at the time of writing is 3.13. Since 2018, Python releases have been annual in October.

[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.

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. VS Code supports Jupyter notebooks natively with an extension. We will cover 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.

3 Installing Python

In this chapter, I cover installing Python 3.13 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.

3.1 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.

3.2 Installation

3.3 macOS

uv

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 `⌘` + `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
```

Visual Studio Code

Download Visual Studio Code from code.visualstudio.com.

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

[2]`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.

Git and GitHub

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.

3.4 Linux

To be honest, if you're using Linux, you probably already know how to install Python and other tools. The instructions below are for manual installation, but **you probably want to use your distribution's package manager instead.**

uv

Installation instructions can be found here.

Visual Studio Code

Download Visual Studio Code from code.visualstudio.com.

Git and GitHub

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.

3.5 Windows

uv

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:

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

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

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

[4]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.

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 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

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 many beginners prefer to use GitHub Desktop.

3.6 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 exmaples, you will need to create a GitHub account. You can create a free account 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.

3.7 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, if I want to try something quickly like reading a CSV file to explore it, I can do it without creating a new project. It was common to use the default (or base) environment for this, but I can lead to issues with updating packages, so many Python distribution now lock this default environment, preventing 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`. Then, I need to create a new project in this directory:

3.8 macOS

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

3.9 Linux

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

3.10 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
```

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:

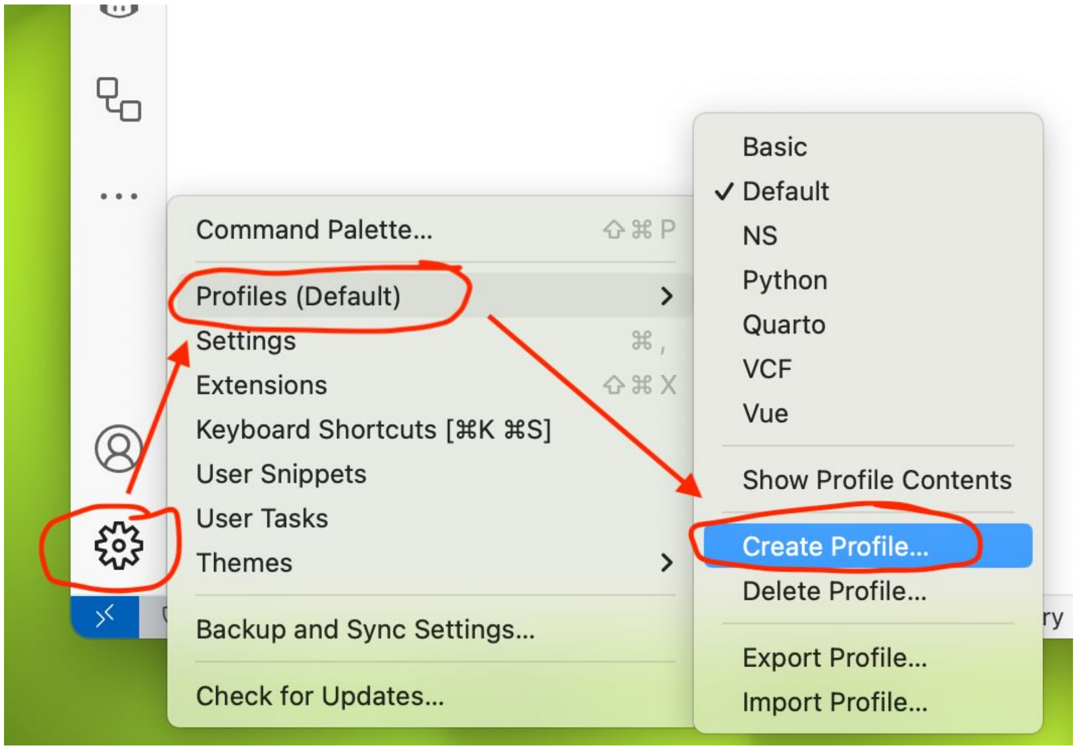
```
uv add pandas numpy scipy matplotlib seaborn statsmodels scikit-learn linearmodels pyarrow jupyter pytest
```

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.

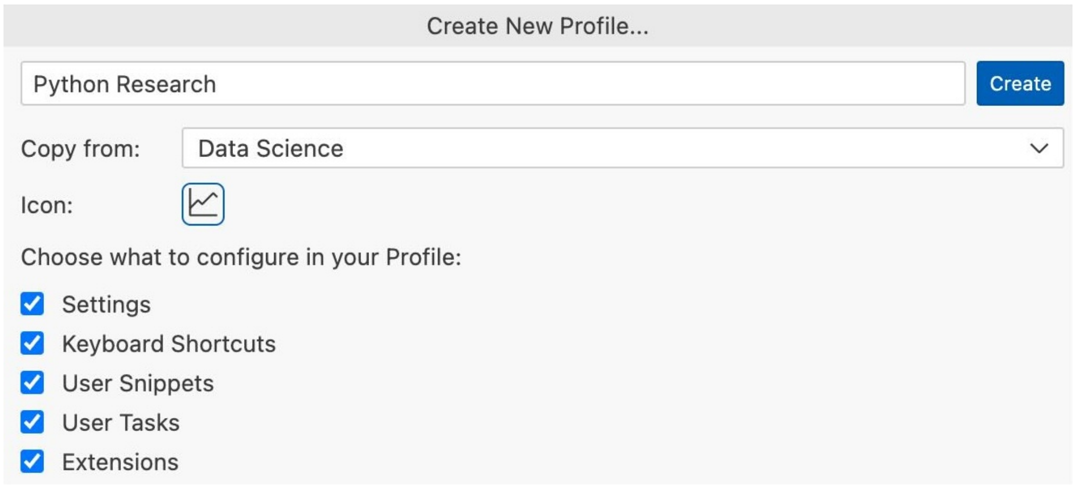
3.11 Configuring Visual Studio Code

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.


Create New Profile...

Python Research

Copy from: Data Science

Icon:

Choose what to configure in your Profile:

- ✓ Settings
- ✓ Keyboard Shortcuts
- ✓ User Snippets
- ✓ User Tasks
- ✓ Extensions

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 `⌘ + 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.

3.12 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.

3.12.1 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)

3.13 What's next?

Now that you have a complete Python environment, you can start learning Python. Watch out for my upcoming tutorials on Python for finance research in the coming days.

4 Python Basics

4.1 Introduction

In this tutorial, we lay the foundation for your programming skills by exploring the basic syntax of Python 3.13.

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 tutorial 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 tutorial, 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 avoiding more advanced topics such as object-oriented programming, functional programming, and parallel computing. These topics are important, but they are not necessary to get started with Python.

Let's dive into the world of Python and begin your journey toward becoming a proficient financial empiricist.

4.2 Prerequisites

This tutorial assumes no prior knowledge of Python or programming in general. However, it is helpful to have some familiarity with basic mathematical concepts such as functions, variables, and equations. Because most of the examples in this tutorial are related to finance, it is also helpful to have some basic knowledge of finance and economics. However, this is not a requirement.

4.3 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 4.1.

Table 4.1: 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`
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`

4.3.1 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.

4.3.1.1 `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
```

4.3.1.2 `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
```

4.3.1.3 `complex`

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

```
2.3 + 4.5j
```

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

4.3.1.4 `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
```

4.3.1.5 `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
```

4.3.2 `str`

Strings are sequences of characters. Strings literals are written by enclosing a sequence of characters in single or double quotes. Note that doubles quotes are preferred by the black code 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.

```
"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
```

4.3.3 `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.

4.4 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 4.2, and Table 4.3 presents some examples of valid and invalid variable names.

Table 4.2: 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 4.3: 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 4.4 summarizes the most common conventions.

Table 4.4: 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:

```python
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.

4.4.1 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.

```python
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 Jupyter 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.

```python
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.

4.4.2 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
```

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

```
int
```

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

```
bool
```

```python
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:

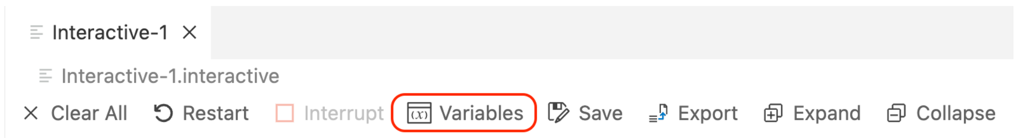
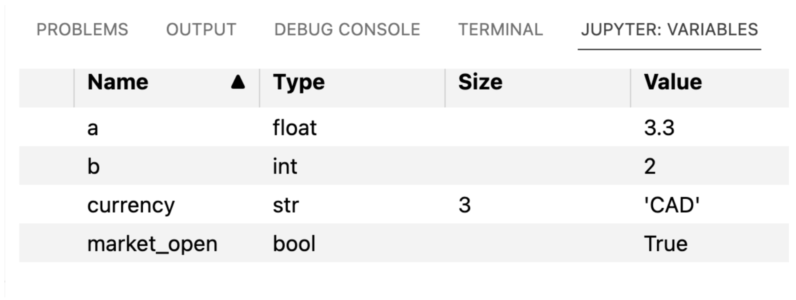


Figure 4.1: Variable View button

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:



Name	Type	Size	Value
a	float		3.3
b	int		2
currency	str	3	'CAD'
market_open	bool		True

Figure 4.2: Variable View

4.4.2.1 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.

```python
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`.

```python
bool(0)
```

```
False
```

```python
bool(1)
```

```
True
```

```python
bool("")
```

```
False
```

```python
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.

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

```
NoneType
```

4.5 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 4.5 when deciding when to use comments and when to avoid them:

Table 4.5: 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.

4.5.1 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 ▯.

4.6 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.

ℹ 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.

4.6.1 Operations

The Python language supports many mathematical operations. Table 4.6 lists some of the most commonly used operators.

Table 4.6: 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`

```python
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 4.8.2.

4.6.2 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.

```python
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:

```python
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.

```python
math.pi
```

```
3.141592653589793
```

4.6.3 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.

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

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
```

```
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.

4.6.4 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.

4.6.5 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.

4.6.5.1 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.

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

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

present_value
```

```
783.5261664684588
```

4.6.5.2 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:

```python
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

4.7 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:

```python
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 4.12.

4.8 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:

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

4.8.1 String operations

The Python language provides many common string operations. Table 4.7 lists some of the most commonly used operations.

Table 4.7: Common string operations

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

Operation	Example	Description
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.

```python
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.

```python
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.

```python
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.

4.8.2 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:

```python
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.

```python
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)
```

```
2024-11-19
```

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.

```python
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:

```python
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.

4.9 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 4.8 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 4.8: 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"`

4.9.1 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.

```python
# 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
```

4.9.2 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.

```python
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.

```python
# 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.

```python
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.

4.9.2.1 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:

```python
# 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:

```python
# 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.

4.9.3 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.

```python
# 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.

```python
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.

4.9.4 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.

```python
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.

```python
# 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}")
```

```python
# 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.

4.10 Comparison operators and branching

4.10.1 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 4.9 lists the comparison operators available in Python.

Table 4.9: 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 4.10 lists the logical operators available in Python.

Table 4.10: 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.

```python
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
```

4.10.2 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`.

```python
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`.

```python
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.

```python
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.

```python
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.

```python
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.
```

4.10.3 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`.

```python
price = 150

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

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

4.11 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.

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

```python
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]
```

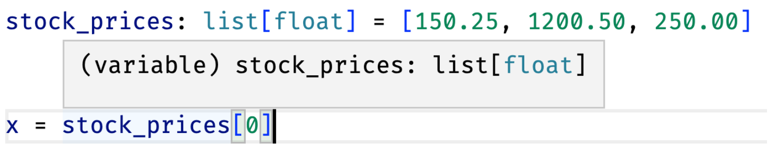
```python
# You can also specify function parameters and return types:


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


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

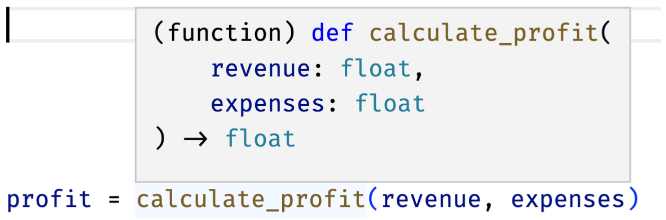
Type hints in Visual Studio Code

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.



```
stock_prices: list[float] = [150.25, 1200.50, 250.00]
    (variable) stock_prices: list[float]
x = stock_prices[0]
```

Figure 4.3: Tooltip when hovering over variable.



```
(function) def calculate_profit(
    revenue: float,
    expenses: float
) → float
profit = calculate_profit(revenue, expenses)
```

Figure 4.4: 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)

4.12 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 4.7; 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.

4.12.1 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.

[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.

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.

```python
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.

```python
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.

```python
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:

```python
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.

```python
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.

```python
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}")
```

```python
# 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.

4.12.2 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.

4.12.3 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.

```python
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
```

4.12.4 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.

```python
# 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.

4.13 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.

4.13.1 `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.

4.13.2 `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
```

```python
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
```

```python
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.

```python
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
```

4.13.3 `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
```

4.13.4 `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.

```python
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.

```python
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.

```python
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.

```python
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.

```python
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 0x10e818840>
list(zipped): [(150.25, 10), (1200.5, 5), (250.0, 20)]
```

4.13.5 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.

```python
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
```

4.13.6 `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:

```python
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.

```python
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`.

4.14 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.

4.14.1 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.

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

print(squared1)

# This is much shorter!
```

```python
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.

```python
# 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]
```

4.14.2 Dictionary comprehensions

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

```python
# 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}
```

4.14.3 Filtering and transforming

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

```python
# 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.

```python
# 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}
```

4.14.4 Nested comprehensions

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

```python
# 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), ('MSF
```

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.

4.15 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.

```python
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
```

4.16 Additional resources

- Python 3.13 documentation
- Lubanovic, Bill. *Introducing Python*, 2nd Edition, O'Reilly Media, Inc., 2019