O'REILLY®

2nd Edition

Python for Excel

A Modern Environment for Automation and Data Analysis

Early Release

RAW & UNEDITED

Felix Zumstein

Python for Excel

SECOND EDITION

A Modern Environment for Automation and Data Analysis

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Felix Zumstein

Python for Excel

by Felix Zumstein

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Michelle Smith

Development Editor: Melissa Potter

Production Editor: Jonathon Owen

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

March 2021: First Edition

Revision History for the Early Release

- 2025-10-27: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492081005 for release details.

979-8-341-64029-0

[LSI]

Brief Table of Contents (*Not Yet Final*)

Chapter 1. Why Python for Excel?

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at mpotter@oreilly.com.

Many Excel users start to reconsider their spreadsheet tools when they hit a limitation. A classic example is when Excel workbooks contain so much data and so many formulas that they become slow or in the worst case, crash. Don't let it get that far! If you work on mission-critical workbooks where errors can result in financial or reputational damage or if you spend hours updating Excel workbooks manually, you should add a programming language to your tool belt. A programming language allows you to automate error-prone copy/paste processes and can turn unreadable Excel formulas into maintainable functions. It can also connect your spreadsheet to external systems such as a database, allowing you to trim down your workbook and make it fast again. This book uses Python as the programming language for Excel, so let's take a quick look at the origins of our two main characters!

Given how fast technology moves, Excel and Python have both been around for a remarkably long time: Excel was first launched in 1985 by

Microsoft—and this may come as a surprise—it was only available for Apple Macintosh. Windows users had to wait until 1987 when Excel 2.0 came out. Microsoft wasn't the first player in the spreadsheet market, though: VisiCorp came out with VisiCalc in 1979, followed by Lotus Software in 1983 with Lotus 1-2-3. Even Microsoft had another application in the game: three years before Excel, they released Multiplan, a spreadsheet program for MS-DOS.

Python was born in 1991, just six years after Excel. While Excel became popular early on, Python had a longer journey before it got traction as a scripting language and in web development. In 2005, Python started to become a serious alternative for scientific computing when *NumPy*, a package for array-based computing and linear algebra, was released. NumPy combined two predecessor packages and therefore streamlined all development efforts around scientific computing into a single project. Today, it forms the basis of countless scientific packages, including *pandas*, which came out in 2008 and which is largely responsible for the widespread adoption of Python in the world of finance and data science that started to happen around 2010. Today, thanks to the growing scientific ecosystem, Python has become one of the most commonly used languages for machine learning (ML) and artificial intelligence (AI).

In this chapter, I will point out what the advantages of Python are compared to Excel's traditional programming language, VBA (Visual Basic for Applications). I will focus on Python's core principles, scientific computing capabilities, and its modern language features. To conclude, I will introduce a few tools that you will get access to as a Python developer.

Python Core Principles

Like every programming language, Python has a few core principles that shaped it into what it is today: a general-purpose language with an emphasis on readability. Python is open source and cross-platform, and it comes with a comprehensive standard library. In this section, I'll explain what all this means and why it matters.

General-Purpose Language

Python is a general-purpose programming language, meaning it wasn't designed for a specific domain like web development or data analysis: Python can be used for many different purposes, including Excel automation, machine learning, and large-scale web applications. This also means that you could turn your Python-powered Excel tools into fully-fledged web applications without having to rewrite the code with the business logic. In contrast, VBA was designed as a domain-specific language to automate and extend Microsoft Office applications. Consequently, VBA code can't be used outside of the Microsoft ecosystem—it would need to be rewritten in another language if you want to turn your Excel tool into a web app.

Beyond being general-purpose, Python was designed with another key principle in mind: readability.

Readability Counts

If your code is readable, it is easy to follow and understand. This makes it easier to spot errors and maintain the code going forward. And maintenence is where most time is spent in programming rather than writing the code in the first place. That's why one line in *The Zen of Python* is "readability counts." The Zen of Python is a concise summary of Python's core design principles, and we will learn how to print it in the next chapter. Let's have a look at the following VBA code snippet:

```
If i < 5 Then
    Debug.Print "i is smaller than 5"
ElseIf i <= 10 Then
    Debug.Print "i is between 5 and 10"
Else
    Debug.Print "i is bigger than 10"
End If
```

In VBA, you can reformat the code into the following, which is completely equivalent:

```
If i < 5 Then
    Debug.Print "i is smaller than 5"
    ElseIf i <= 10 Then
    Debug.Print "i is between 5 and 10"
    Else
    Debug.Print "i is bigger than 10"
End If
```

In the first version, the visual indentation aligns with the logic of the code. This makes it easy to read and understand. In the second version, it's much harder to spot the `ElseIf` and `Else` conditions when you glance over it.

To enforce readability, Python doesn't accept code that is formatted like the second example: it forces you to align the visual indentation with the logic of the code. To do this, Python relies on indentation to define code blocks as you use them in `if` statements. Instead of indentation, the majority of programming languages use curly braces to define code blocks, and VBA uses keywords such as `End If`, as we've just seen.

You will learn all about Python's indentation rules in Chapter 3. Now, however, I'll explain what it means when they say that Python comes with batteries included.

Batteries Included

Python comes with a rich set of built-in functionality delivered by its *standard library*. The Python community likes to refer to it by saying that Python comes with "batteries included." Whether you need to uncompress a ZIP file, read the values of a CSV file, or want to send an email, Python's standard library has you covered, and you can achieve all this in usually just a few lines of code. With VBA, you are often required to write a lot of code or install an add-in for these basic tasks. Another issue with VBA is that it isn't truly cross-platform. Let's see why this matters!

Cross-Platform

Even if you develop your programs on a laptop that runs on Windows or macOS, it's likely that one day, you want to run your application on a server

or in the cloud. Servers allow you to access your application over the web or run your Python script fully automated as part of a pipeline. In fact, I will show you how to run Python code on a server in the next chapter by introducing you to hosted Jupyter notebooks. The vast majority of servers run on Linux, an operating system that is stable, secure, and open source. And since Python programs run on all major operating systems, this will take out much of the pain when you transition from your local machine to a server or cloud setup.

In contrast, even though Excel VBA runs on Windows and macOS, it's easy to introduce functionality that only runs on Windows. In the official VBA documentation or on community forums, you will often see code like this:

```
Set fso = CreateObject("Scripting.FileSystemObject")
```

Whenever you have a `CreateObject` call or are being told to add a reference in the VBA editor, you are usually dealing with code that will only run on Windows.

With Python's core principles covered, let's now find out why scientific computing has become so big in Python.

Scientific Computing

As we've seen before, an important reason for Python's success is the fact that it was created as a general-purpose programming language. The capabilities for scientific computing were added later on in the form of third-party packages. This has the unique advantage that a data scientist and a web developer can work with the same programming language. Being able to build scientific applications out of a single language reduces friction, implementation time, and costs. Scientific packages like NumPy and pandas give us access to a very concise way of formulating mathematical problems. As an example, let's have a look at a famous financial formula from Modern Portfolio Theory, used to calculate the portfolio variance:

$$\sigma^2 = w^\intercal C w$$

The portfolio variance is denoted by σ^2, while w is the weight vector of the individual assets and C is the portfolio's covariance matrix. If `w` and `C` are Excel ranges, you can calculate the portfolio variance in VBA like so:

```
variance =
Application.MMult(Application.MMult(Application.Transpose(w), C),
w)
```

Compare this to the almost mathematical notation in Python, assuming that `w` and `C` are pandas DataFrames or NumPy arrays (I will formally introduce them in Part 2):

```
variance = w.T @ C @ w
```

But it's not just about aesthetics and readability: NumPy and pandas use compiled Fortran and C code under the hood, which is much faster than VBA when working with big matrices. Other packages that we'll use in this book are written in Rust or C++ for performance reasons, but no matter what low-level language is used, Python provides the tools to create user-friendly packages. This allows you to take advantage of high-performance libraries without needing to understand their underlying implementation.

Thanks to the growing scientific ecosystem, Python has become the de-facto lingua franca for machine learning and AI. All important machine learning frameworks, such as scikit-learn, Tensorflow, PyTorch, or XGBoost are available as Python packages. And AI platforms such as ChatGPT offer a Python library to interact with their service programmatically.

OPEN SOURCE SOFTWARE (OSS)

Python is open source software (OSS). If software is distributed under an open source license, it means that its source code is freely available at no cost, allowing everybody to contribute new functionality, bug fixes, or documentation. Python itself and almost all third-party Python packages are open source and most commonly maintained by developers in their spare time. This is not always an ideal state: if your company is relying on certain packages, you have an interest in the continued development and maintenance of these packages by professional programmers. Fortunately, the scientific Python community has recognized that some packages are too important to leave their fate in the hands of a few volunteers.

That's why in 2012, NumFOCUS, a nonprofit organization, was created to sponsor various Python packages and projects in the area of scientific computing. NumFOCUS sponsors many popular Python packages, including pandas and NumPy, but nowadays they also support projects from various other languages, including R, Julia, and JavaScript. Similarly, Python is overseen by the Python Software Foundation that welcomes new members and donors.

Missing support for scientific computing is an obvious limitation in VBA. But even if you look at the core language features, VBA has fallen behind.

Modern Language Features

Since Excel 2000, the VBA language isn't actively developed anymore. This, however, doesn't mean it's dead: Microsoft is still shipping updates to VBA to be able to automate new Excel features. For example, Excel 2016 added support to automate Power Query. A language that stopped evolving such a long time ago is missing out on modern language concepts that were introduced in all major programming languages over the years. In this

section, we're going to look at some of them, including text and error handling, array support, web APIs, and modules.

Working with Text

VBA has very limited tools to work with strings, i.e., text data. Let's have a look at the following sentence:

```
This was printed by a language called "VBA".
```

To print this with the language name stored in a variable, you would write the following code in VBA:

```
lang = "VBA"
message = "This was printed by a language called """ & lang &
""""."
Debug.Print message
```

This is a very short string, but it's already hard to know how many quotes you have to put where. Python, on the other hand, has powerful f-strings, which we will cover in Chapter 3. The same example reads like this in Python:

```
lang = "Python"
message = f'This was printed by a language called "{lang}".'
print(message)
```

Just like with text, handling arrays in VBA comes with its own set of challenges, as we will see next.

Array Support

Working with arrays in VBA is hard and verbose. For example, multiplying all values in an array by two looks something like this:

```
Dim numbers(1 To 4) As Integer
Dim doubled() As Integer
Dim i As Integer
```

```
' Initialize array
numbers(1) = 1
numbers(2) = 2
numbers(3) = 3
numbers(4) = 4

' Double each element
ReDim doubled(LBound(numbers) To UBound(numbers))
For i = LBound(numbers) To UBound(numbers)
    doubled(i) = numbers(i) * 2
Next i
```

In Python, you can write the same in two lines of code (I will show you how this works in Chapter 4):

```
import numpy as np

numbers = np.array([1, 2, 3, 4])
doubled = numbers * 2
```

Python makes working with arrays easy. The same is true for web APIs, our next topic.

Web APIs

Microsoft stopped investing in VBA around the time the internet became mainstream. It's therefore not surprising that downloading data from a Web API is a cumbersome task in VBA, especially if you want it to work across Windows and macOS.

In contrast, Python has various packages that make it easy to talk to a Web APIs. Since Web APIs usually send their response in the JSON (JavaScript Object Notation) format, accessing the information of interest is straight forward, as Python—unlike VBA—has built-in support to read JSON. We will see how all this works in Chapter 10.

Nowadays, Python also supports asynchronous programming via its asyncio library. This makes network requests such as talking to a web API or accessing databases highly efficient. Asynchronous programming is an

advanced topic though, so we will mostly ignore it in this book. Rather, let's continue with the differences in error handling between VBA and Python.

Error Handling

If you'd like to handle an error gracefully in VBA, it goes something like this:

```
Sub PrintReciprocal(number)
    ' There will be an error if "number" is 0 or a string
    On Error GoTo ErrorHandler
    result = 1 / number
    GoTo CleanUp

ErrorHandler:
    Debug.Print "There was an error: " & Err.Description
    result = "N/A"

CleanUp:
    Debug.Print "The reciprocal is: " & result
    On Error GoTo 0   ' Reset error handling
End Sub
```

Please note that you might calculate the reciprocal differently in real life—I only use it here as an example to make it easier to follow the code flow. VBA error handling involves the use of *labels* like `ErrorHandler` and `CleanUp`. You instruct the code to jump to these labels via the `GoTo` statements. Early on, labels were recognized to be responsible for what many programmers would call *spaghetti code*: a nice way of saying that the flow of the code is hard to follow and therefore difficult to maintain. That's why modern languages use more advanced error handling—Python uses the `try/except` statements that I will introduce in Chapter 11.

With error handling out of our way, let's conclude this section by comparing how modules differ in Python and VBA.

Modules

Modules are a way to break up your code into different sections, usually files. While VBA supports modules, it has limitations:

No isolation between modules

> No matter in which module you define e.g., a function, it will be available globally. This means that you can't use the same function name in different modules.

Hidden dependencies

> Since VBA doesn't require explicit imports when using objects from other modules, it's hard to understand which functions or variables a module actually relies on.

In Chapter 3, I will introduce you to how Python modules work and how they allow you to selectively import a function or variable from another module to prevent these issues.

With the modern language features covered, we can move on to the last section in this chapter: it explains the various tools that you get access to as a Python developer.

Tooling

Modern programming languages like Python rely on a set of tools that make software development a lot more reliable. In this section, I'll give you a short introduction to a few of these, including package management, testing, and version control.

Package Manager

While Python's standard library covers an impressive amount of functionality, it—as discussed before—doesn't come with any domain-specific functionality such as data science libraries or web frameworks. This is where PyPI comes in. PyPI stands for *Python Package Index* and is

a giant repository where everybody (including you!) can upload Python packages that add additional functionality to Python.

> **PYPI VS. PYPY**
>
> PyPI is pronounced “pie pea eye.” This is to differentiate PyPI from PyPy which is pronounced “pie pie” and which is a fast alternative implementation of Python.

A package manager allows you to install additional packages like pandas with a simple command. What sounds easy is in reality a very complex task as a package often depends on other packages and these again on yet another set of packages. The package manager makes sure that these dependencies and sub-dependencies are installed with compatible versions and that you can reproduce your exact set of dependencies on another machine. For example, when you install pandas, NumPy is automatically installed as a dependency. In this book, we will use the uv package manager, which I will introduce in the next chapter.

In VBA, there is no package manager, so you either have to use add-ins or copy/paste VBA code whenever you create a new Excel file. However, both of these solutions have issues: on the one hand, add-ins are global, so you have to use the same version of the add-in for every workbook. This can cause compatibility issues with workbooks you created with an older version of the add-in. On the other hand, copy/pasting VBA code is a nightmare for maintainability: if you fix the code in one workbook, you have to apply the fix to all other workbooks, too.

While the package manager makes it easy to add new functionality to your projects, another important aspect of professional software development is testing, which makes sure your code works as expected.

Testing

When you tell an Excel developer to test their workbooks, they will most likely perform a few random checks, e.g., click a button and see if the

macro still does what it is supposed to do. This is, however, a risky strategy: Excel makes it easy to introduce errors that are hard to spot. For example, you can overwrite a formula with a hardcoded value. Or you forget to adjust a formula in a hidden column.

When you tell a Python developer or an AI assistant to test their code, they will write *unit tests*. As the name suggests, it's a mechanism to test individual components of your program. For example, unit tests make sure that a single function of a program works properly. Most programming languages offer a way to run unit tests automatically. Running automated tests will increase the reliability of your codebase and can prevent you from introducing bugs while making changes to your codebase.

Pretty much all programming languages offer a test framework to write unit tests without much effort—but not Excel. Fortunately, by connecting Excel with Python, you get access to Python's unit testing frameworks, most notably pytest. While a more in-depth presentation of unit tests is beyond the scope of this book, I invite you to have a look at my blog post, in which I walk you through the topic.

Unit tests are often set up to run automatically when you commit your code to your version control system. The next section explains what a version control system is and why working with Excel files can make version control difficult to use.

Version Control

A *version control system* (VCS) tracks changes in your source code over time, allowing you to see who changed what, when, and why, and allows you to revert to old versions at any point in time. The most popular version control system nowadays is Git. It was originally created to manage the Linux source code and since then has conquered the programming world. Excel offers a version history if you save your file on OneDrive or SharePoint and are subscribed to Microsoft 365, but this doesn't cover VBA code. Therefore, the most popular version control system for Excel still looks like this:

```
currency_converter_v1.xlsx
currency_converter_v2_2020_04_21.xlsx
currency_converter_final_edits_Bob.xlsx
currency_converter_final_final.xlsx
```

If, unlike in this sample, the Excel developer sticks to a certain convention in the file name, there's nothing inherently wrong with that. But keeping a version history of your files locally locks you out of important aspects of source control in the form of easier collaboration, peer reviews, sign-off processes, and audit logs. And if you want to make your workbooks more secure and stable, you don't want to miss out on these things. Most commonly, Git is used in connection with a web-based platform like GitHub or GitLab. These platforms allow you to work with so-called *pull requests*, which allow developers to formally request that their changes are merged into the main codebase. A pull request offers the following information:

- Who is the author of the changes
- When were the changes made
- What is the purpose of the changes
- Which code was added, changed, or removed

This allows a coworker or an AI assistant to review the changes and flag any potential issues. VBA can't be easily version-controlled with Git as it is stored in the Excel workbook, which Git can't read. However, when you write Python code in standalone files, you can use Git to its full potential. Not all code the code in this book lives in a standalone Python file, though. For example, Python in Excel, which we'll meet in Chapter 6 stores the Python code in Excel cells, so you can't easily commit it to Git. But you can use Git for many other use cases that we'll meet in this book, like the Excel tool we'll build in Chapter 10.

Conclusion

In this chapter, we met Python and Excel, two very popular technologies that have been around for decades—a long time compared to many other software that we use today.

Python comes with powerful features that are missing in Excel: the standard library, package managers, libraries for scientific computing, and cross-platform compatibility. By learning how to combine Excel with Python, you get the best of both worlds: Excel provides a familiar and easy-to-use user interface, while Python allows you to save time through automation, leverage scientific packages, and gain the flexibility to scale your application beyond Excel.

Now that you know why Python is such a powerful companion for Excel, it's time to set up your development environment so you can to write your first lines of Python code!

Chapter 2. Development Environment

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at mpotter@oreilly.com.

To write VBA code, it's enough to launch Excel and open the VBA editor. To use a full Python installation, it's a bit more work. If this book were only about Python in Excel (Part 3) and xlwings Lite (Part 5), we wouldn't have to install anything locally. But using Python from Excel covers only one side of the story—we don't want to miss out on the other side, which is using Excel from Python (Part 4 and Part 6). Besides that, both Python in Excel and xlwings Lite have certain limitations that a full Python installation doesn't have. This chapter will make you familiar with all the tools you need as a Python developer. We will start with a quick introduction to the Terminal, the most fundamental programming tool. We'll continue with the installation of uv, a modern package manager, before using it to install Python itself. With Python installed, we will run our first code snippets in Jupyter notebooks and VS Code: *Jupyter notebooks* allow us to work with data, code, and charts in an interactive way, which makes them a serious competitor to Excel workbooks. *Visual Studio Code* (VS Code) is a powerful text editor that works great for

writing, running, and debugging Python code. As this book is about Excel, I am focusing on Windows and macOS in this chapter. Let's get started by downloading the companion repository!

Companion Repository

To install Python and run the examples in this book, you first need to download the companion repository. Go to *https://github.com/fzumstein/python-for-excel/tree/2e* and click on the green Code button, then select Download ZIP. This downloads a file called *python-for-excel-2e.zip*. Once downloaded, extract the ZIP file into your *Documents* folder by following the instructions according to your operating system:

Windows

> Right-click the downloaded ZIP file and select Extract All. In the dialog box, click on Browse, select the *Documents* folder, then click the Select Folder button to confirm. The dialog box should now show a path similar to this one: *C:\Users\username\Documents*. To finish, click on Extract. This will unzip the contents to something along these lines: *C:\Users\username\Documents\python-for-excel-2e*.

macOS

> Double-click the ZIP file to extract the files into a folder called *python-for-excel-2e* (note that some browsers like Safari do this automatically). Then copy the extracted folder to the *Documents* directory so that the full path becomes somthing along these lines: */Users/username/Documents/python-for-excel-2e*.

Alternatively, if you know how to work with Git version control, feel free to clone the repository to your *Documents* folder instead of downloading the ZIP file manually.

With the companion repository sitting in your *Documents* folder, let me give you a quick introduction to the Terminal, as we will use it to install Python.

Using the Terminal

The Terminal is the most fundamental programming tool, and we will use it extensively to e.g., run Python scripts. If you have never used a Terminal, don't worry: you only need to know a handful of commands to work through this book. Once you get used to it, using the Terminal is often faster and more convenient than clicking your way through graphical user menus. Before we start, let me remind you that you should only type in the commands shown in **`bold`**. Text in regular font represents Terminal output that you won't need to type. Let's now open a Terminal:

Windows

> Click on the Windows Start menu, type in **`Command Prompt`** in the search box and click on the Command Prompt entry that appears. Alternatively, use the keyboard shortcut Windows+R to open the Run box. Then type **`cmd`** into the box and hit Enter. The input line of the Command Prompt will look something like this:
>
> ```
> C:\Users\username>
> ```

macOS

> Press Command-Space or open the Launchpad, then type **`Terminal`** and hit Enter. Alternatively, open the Finder and navigate to *Applications* > *Utilities*, where you will find the Terminal app that you can double-click. Once the Terminal appears, its input line looks something like this:
>
> ```
> username@MacBook ~ %
> ```

The tilde stands for the home directory, which is usually */Users/username*. To see the full path of your current directory, type **`pwd`** followed by Enter. `pwd` stands for *print working directory*.

> **TERMINAL**
>
> Going forward, whenever I write "Terminal," I am referring to the Command Prompt on Windows and the Terminal on macOS. Recent versions of Windows even default to opening the Command Prompt inside a new app, which is, indeed, called Terminal.

In a Terminal, try out the commands outlined in Table 2-1. I'll explain each command in more detail after the table.

Table 2-1. Commands for the Terminal

Command	Windows	macOS
List files in current directory	`dir`	`ls -la`
Change directory (relative)	`cd path\to\dir`	`cd path/to/dir`
Change directory (absolute)	`cd C:\path\to\dir`	`cd /path/to/dir`
Change to D drive	`D:`	(doesn't exist)
Change to parent directory	`cd ..`	`cd ..`
Scroll through previous commands	↑ (`up-arrow`)	↑ (`up-arrow`)

List files in current directory

On Windows, type in **`dir`** for *directory*, then hit Enter. This will print the content of the directory you are currently in.

On macOS, type in **`ls -la`** followed by Enter. `ls` is short for *list directory contents*, and `-la` will print the output in the *long listing* format and include *all* files, including hidden ones.

FILE EXTENSIONS

Unfortunately, Windows and macOS hide file extensions by default in the Windows Explorer or macOS Finder, respectively. This can make it harder to work with Python scripts and the Terminal, as they will require you to refer to files including their extensions. When working with Excel, showing file extensions also helps you understand whether you're dealing with the default *xlsx* file, a macro-enabled *xlsm* file, or any of the other Excel file formats. Here is how you make the file extensions visible:

Windows

Open a File Explorer and click on the View tab. Under Show, click on "File name extensions" to activate it.

macOS

Open the Finder and go to Preferences by hitting Command-, (Command-comma). On the Advanced tab, check the box next to "Show all filename extensions."

Change directory

Type **`cd Doc`** and hit the Tab key. If you are in your home folder and using an English version of Windows or macOS, hitting the Tab key

will autocomplete `cd Doc` to `cd Documents` (`cd` stands for *change directory*). If you are in a different folder, simply start to type the beginning of one of the directory names you saw with the previous command (`dir` or `ls -la`) before hitting the Tab key to autocomplete. Then hit Enter to change into the autocompleted directory. If you are on Windows and need to change your drive, you first need to type in the drive name before you can change into the correct directory:

```
C:\Users\username> D:

D:\> cd data

D:\data>
```

Note that by starting your path with a directory or file name that is within your current directory, you are using a *relative path*, e.g., `cd Documents`. If you would like to go outside of your current directory, you can type in an *absolute path*, e.g., `cd C:\Users` on Windows or `cd /Users` on macOS (mind the forward slash at the beginning).

PERMISSIONS ON MACOS

When you change into the *Documents* directory on a Terminal on macOS and run a command such as `ls` for the first time, you will be shown the following pop-up window: "Terminal would like to access files in your *Documents* folder." Make sure to click on "Allow." If you clicked "Don't Allow" by mistake, go to Apple menu > System Settings > Privacy Security > Files & Folders > Terminal. Click on the dropdown so that you can enable the toggle for the *Documents* Folder.

Change to parent directory

To go to your parent directory, i.e., one level up in your directory hierarchy, type **`cd ..`** followed by Enter (make sure that there is a space between `cd` and the dots). You can combine this with a directory name: if you want to go up one level and change to the *Desktop*, enter

`cd ..\Desktop`. On macOS, replace the backslash with a forward slash.

CHANGE TO THE DIRECTORY OF THE COMPANION REPOSITORY

Most Terminal commands in this book need to be run from the directory of the companion repository. In these cases, to prevent confusion, I'll always include the appropriate `cd` command:

Windows

```
cd %USERPROFILE%\Documents\python-for-excel-2e
```

`%USERPROFILE%` is a so-called environment variable, which points to your home directory, e.g., *C:\Users\username*. If you already are in the correct directory, you can ignore this line. The Terminal, i.e., Command Prompt, shows the full path on its input line, so it's easy to check in which directory you are:

```
C:\Users\username\Documents\python-for-excel-2e>
```

macOS

```
cd ~/Documents/python-for-excel-2e
```

As I mentioned earlier in this section, ~ points to your home directory, e.g., */Users/username*. If you already are in the correct directory, you can ignore this line. Since macOS only shows the name of the current directory, you may want to run the `pwd` command that we met earlier to check the full path of your current directory:

```
username@MacBook python-for-excel-2e % pwd
/Users/username/Documents/python-for-excel-2e
```

Scroll through previous commands

Use the up-arrow key to scroll through the previous commands. This will save you many keystrokes if you need to run the same commands over and over again. If you scroll too far, use the down-arrow key to scroll back.

And that's it! You are now able to fire up a Terminal and run commands in the desired directory. Let's use your new skills right away to install Python!

Installing Python

In this section, you will install Python by using uv, a modern command-line tool. We therefore start this section by installing uv itself. At the end of this section, you'll be running an interactive Python session.

Installing uv

While uv is primarily a package manager (see Chapter 1), it covers a lot more functionality, including the installation of Python itself. To install uv, use the following instructions according to your operating system. If you hit an issue, consult the original installation instructions for up-to-date information.

Windows

In a Terminal, i.e., Command Prompt, type the following command:

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

Now hit Enter to run the command and—importantly—close and reopen the Command Prompt to finish the installation.

macOS

In the Terminal, type the following command:

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

Now hit Enter to run the command and—importantly—close and reopen the Terminal to finish the installation. To properly close the Terminal, make sure to use the keyboard shortcut Command-Q.

After reopening the Terminal, type **`uv`** and hit Enter: this should print the help page of uv. If it doesn't, make sure to properly restart the Terminal before trying again. Let's now use uv to install Python and the third-party packages that we'll use!

Installing Python and Packages

Now that we have uv installed, let's use it to install Python and all the packages that we will use in this book. In a Terminal, first change into the directory of the companion repo, then install Python and the packages by running the following commands (hit Enter after each line):

Windows (Terminal)

```
cd %USERPROFILE%\Documents\python-for-excel-2e
uv sync
```

macOS (Terminal)

```
cd ~/Documents/python-for-excel-2e
uv sync
```

uv will rely on the following files from the companion repo to know what it has to install:

- *.python-version* (defines the Python version)
- *pyproject.toml* (project meta data)
- *uv.lock* (contains all packages with their exact version)

During the installation, uv creates the folder *.venv* in the root of the companion repository, which is a *virtual environment*, see sidebar. To get a more detailed introduction to uv and see how you can create your own projects, have a look at Appendix A.

VIRTUAL ENVIRONMENT

A virtual environment is an isolated workspace with its own Python interpreter and set of packages. Imagine you're working on two projects at the same time: project A might use Python 3.13 with pandas 2.3.2, while Project B requires Python 3.14 with pandas 3.0.0. Since code written for pandas 2.3.2 might need modifications to be compatible with pandas 3.0.0, using a single version of pandas for all your projects doesn't work. Virtual environments solve this issue by keeping each project's dependencies separate.

With Python and our packages installed, let's start an interactive Python session!

Python REPL

You can start an interactive Python session by running `uv run python` in a Terminal. An interactive Python session is called a *REPL*, which stands for *read-eval-print loop*: Python reads your input, evaluates it, and prints the result, then waits for your next input. Remember, if you already are in the directory of the companion repository, you don't need to run the first line with the `cd` command:

Windows (Terminal)

```
cd %USERPROFILE%\Documents\python-for-excel-2e
uv run python
Python 3.13.7 (main, Aug 28 2025, 17:02:49) [MSC v.1944 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

macOS (Terminal)

```
cd ~/Documents/python-for-excel-2e
uv run python
Python 3.13.7 (main, Aug 28 2025, 17:02:49) [Clang 19.1.6 ] on
darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

UV RUN

uv makes it easy to use the correct virtual environment if you follow these rules:

1. Change into the directory of your project, in our case, that's the directory of the companion repository.
2. Run your commands with `uv run`. As we've just seen, `uv run python` starts an interactive interpreter and `uv run hello_world.py` runs a Python script, as we will see in a moment.

Let's play around a bit! Note that >>> in an interactive session means that Python expects your input; you don't have to type this in. In contrast, the lines without >>> show the output of the command on the previous line. Follow along by typing in each line that starts with >>> and confirm with the Enter key:

```
>>> 3 + 4
7
>>> "python " * 3
'python python python '
```

Remember the Zen of Python that I mentioned in the previous chapter? Run the following code (i.e., hit Enter after typing it in) and you'll get a witty version of the guiding principles of Python:

```
>>> import this
```

To exit out of your Python session, type **`quit`** followed by the Enter key. Alternatively, hit Ctrl-D on macOS or Ctrl+Z on Windows. On Windows, you need to additionally press Enter.

Running code in a Python REPL can be helpful at times, but there's a more comfortable way to run code interactively: Jupyter notebooks!

Jupyter Notebooks

In the previous section, I showed you how to start an interactive Python session in a Terminal. This is useful if you want a bare-bones environment to test out something simple. For the majority of your work, however, you want an environment that is easier to use. For example, going back to previous commands and displaying charts is hard with a Python REPL. That's why *Jupyter notebooks* have emerged as one of the most popular ways to run scientific Python code. You interact with Jupyter notebooks in the browser and they allow you to tell a story by combining executable Python code with formatted text, pictures, and charts. They are beginner-friendly and thus especially useful for the first steps of your Python journey.

They are, however, also hugely popular for teaching, prototyping, and researching, as they facilitate reproducible research.

Jupyter notebooks have become a serious competitor to Excel as they cover roughly the same use case as a workbook: you can quickly prepare, analyze, and visualize data. The difference from Excel is that Jupyter notebooks don't mix data and business logic: the Jupyter notebook holds your code and charts, whereas you typically consume data from an external file or a database. Having Python code visible in your notebook makes it easy to see what's going on, whereas in Excel the formulas are hidden behind a cell's value.

In this section, I'll show you the basics of how you run and navigate a Jupyter notebook. Along the way, you will learn about different types of notebook cells and why the run order of cells matters. Let's get started with our first notebook!

Running Jupyter Notebooks

Using a Terminal, launch a Jupyter notebook server:

Windows (Terminal)

```
cd %USERPROFILE%\Documents\python-for-excel-2e
uv run jupyter notebook
```

macOS (Terminal)

```
cd ~/Documents/python-for-excel-2e
uv run jupyter notebook
```

This will automatically open your browser and show the Jupyter dashboard with the files in the directory from where you were running the command.

On the top right of the Jupyter dashboard, click on New, then select Python 3 from the dropdown list (see Figure 2-1).

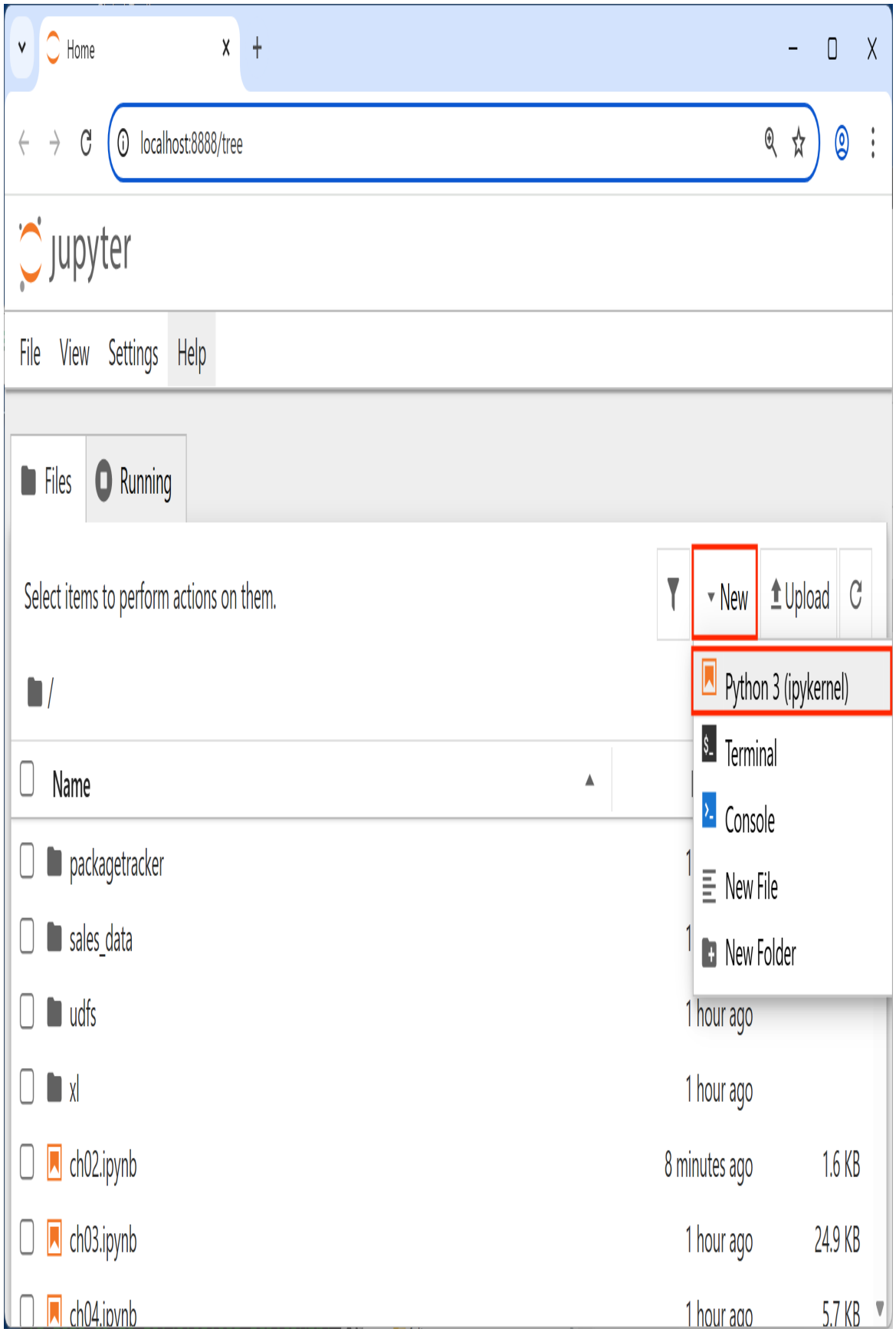
Home
localhost:8888/tree
jupyter
File View Settings Help
Files
Running
Select items to perform actions on them.
New
Upload
/
Python 3 (ipykernel)
Terminal
Console
New File
New Folder
Name
packagetracker
sales_data
udfs
1 hour ago
xl
1 hour ago
ch02.ipynb
8 minutes ago
1.6 KB
ch03.ipynb
1 hour ago
24.9 KB
ch04.ipynb
1 hour ago
5.7 KB

Figure 2-1. The Jupyter dashboard

This will open a new browser tab with your first empty Jupyter notebook as shown in Figure 2-2.

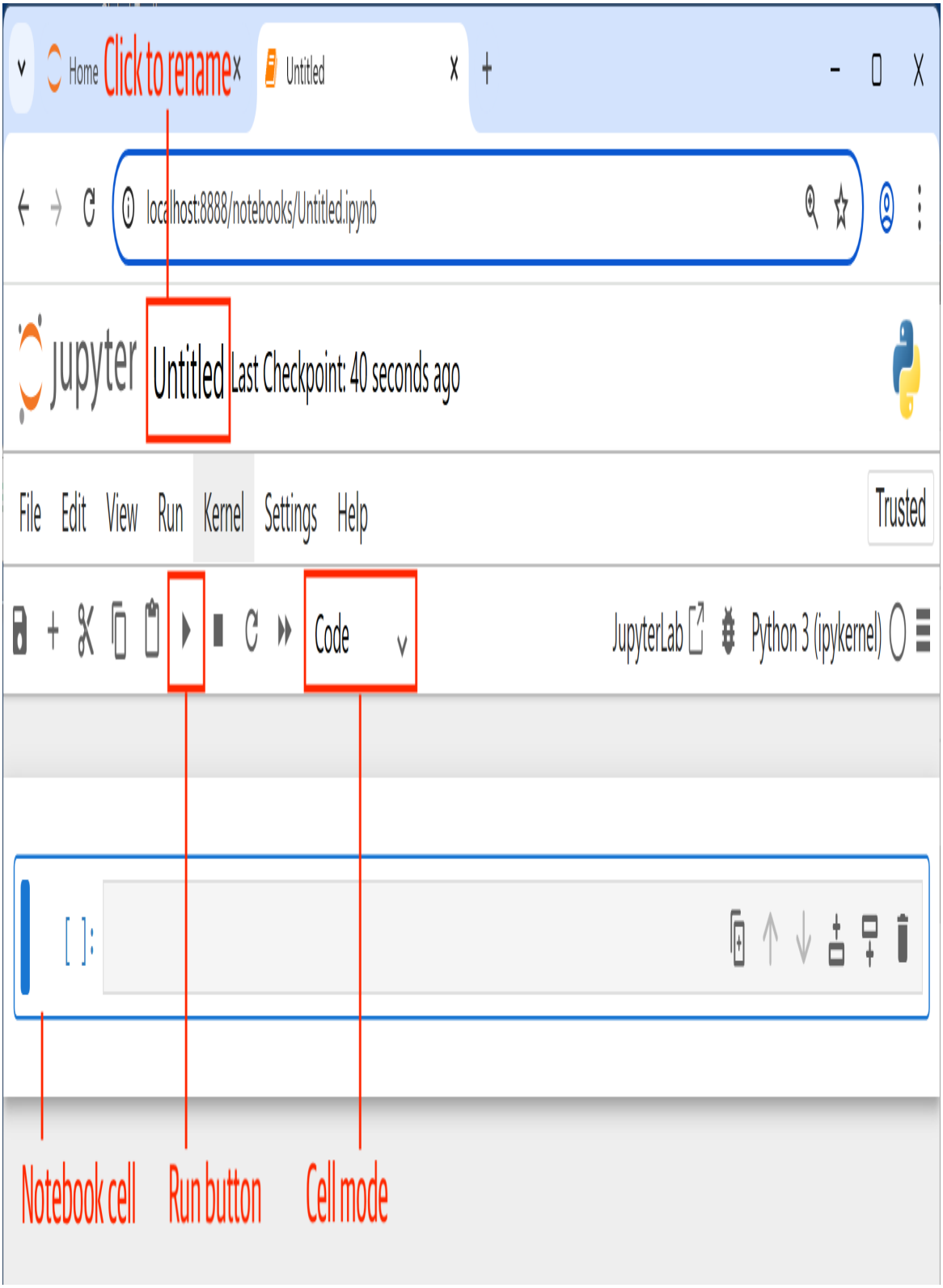

Figure 2-2. An empty Jupyter notebook

JUPYTER NOTEBOOKS IN THE CLOUD

Jupyter notebooks have become so popular that they are offered as a hosted solution by various cloud providers. I am introducing three services here that are all free to use. The great thing about running notebooks in the cloud is that you don't need to install anything locally.

Binder

> Binder is a service provided by Project Jupyter, the organization behind Jupyter notebooks. Binder is meant to try out the Jupyter notebooks from public Git repositories. It's read-only—you can't store anything on Binder and hence you don't need to sign up or log in to use it.

Google Colab

> Google Colab (short for Colaboratory) is Google's notebook platform. Unfortunately, the majority of the Jupyter notebook keyboard shortcuts don't work, but you can access files on your Google Drive, including Google Sheets.

Kaggle Notebooks

> Kaggle is a platform for data science and machine learning with a huge collection of datasets. Kaggle has been part of Google since 2017.

The easiest way to run the Jupyter notebooks of the companion repository in the cloud is by going to its Binder URL. You will be working on a read-only copy of the companion repository, so feel free to edit and break stuff as you like! Note, however, that since Part 4 requires a local installation of Excel, you also need to run the notebooks locally for that part.

It’s a good habit to click on Untitled next to the Jupyter logo to rename your workbook into something more meaningful, e.g., *first_notebook*. The lower part of Figure 2-2 shows a notebook cell—move on to the next section to learn more about them!

Notebook Cells

In Figure 2-2, you see an empty cell with a blinking cursor. If the cursor doesn't blink, click into the cell with your mouse, i.e., to the right of `[ ]`. Now repeat the exercise from the last section: type in **`3 + 4`** and run the cell by either clicking on the Run button in the menu bar at the top or—much easier—by hitting Shift+Enter. This will run the code in the cell, print the result below the cell and jump to the next cell. In this case, it inserts an empty cell below as we only have one cell so far. While a cell is calculating, it shows `[*]` and when it's done, the asterisk turns into a number, e.g., `[1]`. Below the cell you will have the corresponding output labeled with the same number. Every time you run a cell, the number increases by one, helping you understand the order in which cells were executed. Going forward, I will show the code samples in this format:

```
In [1]: 3 + 4
Out[1]: 7
```

As you can see, I am additionally showing the labels `In` and `Out` in front of the cell number to make it clearer what the input and output is. This notation allows you to follow along easily by typing **`3 + 4`** into a notebook cell. When running it by hitting Shift+Enter, you will get what I show as output under `Out[1]`. If you read this book in an electronic format supporting colors, you will notice that the input cell formats strings, numbers, and so on with different colors to make it easier to read. This is called *syntax highlighting*.

DISPLAY OF CELL OUTPUT

If the last line in a cell returns a value, it is automatically printed by the Jupyter notebook under `Out[ ]`. However, when you use the `print` function or when you get an exception, it is printed without the `Out[ ]` label. The code samples in this book are formatted to reflect this behavior.

RUNNING CODE EXAMPLES

To run the examples in this book, you have two options: create a new Jupyter notebook and type the code manually, or open the respective Jupyter notebook file, e.g., *ch02.ipynb*, by clicking on it in the Jupyter dashboard (see Figure 2-1). Using the Jupyter notebooks from the companion repository allows you to run and edit the examples without typing them from scratch.

Cells can have different types, two of which are of interest to us:

Code

This is the default type. Use it whenever you want to run Python code.

Markdown

Markdown is a syntax that uses standard text characters for formatting and can be used to include nicely formatted explanations and instructions in your notebook.

To change a cell's type to Markdown, select the cell, then choose Markdown in the cell mode dropdown (see Figure 2-2). I'll show you a keyboard shortcut to change the cell mode in Table 2-2. After changing an empty cell into a Markdown cell, type in the following text, which explains a few Markdown rules:

```
# This is a first-level heading

## This is a second-level heading

You can make your text *italic* or **bold** or `monospaced`.

* This is a bullet point
* This is another bullet point
```

After hitting Shift+Enter, the text will be rendered into nicely formatted HTML. At this point, your notebook should look like what's in Figure 2-3.

Markdown cells also allow you to include images, videos, or formulas; see the Jupyter notebook docs.

[1]:
```
3 + 4
```

[1]:
```
7
```

This is a first-level heading

This is a second-level heading

You can make your text *italic* or **bold** or `monospaced` .

- This is a bullet point
- This is another bullet point

[ ]:
```

```

Figure 2-3. The notebook after running a code cell and a Markdown cell

Now that you know about the code and Markdown cell types, it's time to learn an easier way to navigate between cells: the next section introduces the edit and command mode along with a few keyboard shortcuts.

Edit vs. Command Mode

When you interact with cells in a Jupyter notebook, you are either in the *edit mode* or *command mode*:

Edit mode

Clicking into a cell starts the edit mode: the cursor in the cell is blinking. Instead of clicking into a cell, you can also hit Enter when the cell is selected.

Command mode

To switch into command mode, hit the Escape key; the blinking cursor will disappear. The most important keyboard shortcuts that you can use while being in command mode are shown in Table 2-2.

Table 2-2. Keyboard shortcuts (command mode)

Shortcut	Action
Shift+Enter	Run the cell and proceed to the next cell (works also in edit mode)
↑ (`up-arrow`)	Move cell selector up
↓ (`down-arrow`)	Move cell selector down
`b`	Insert a new cell *below* the current cell
`a`	Insert a new cell *above* the current cell
`dd`	Delete the current cell (type two times the letter `d`)
`m`	Change cell type to Markdown
`y`	Change cell type to code

Knowing these keyboard shortcuts allows you to work with notebooks very efficiently. In the next section, I'll show you a common gotcha that you need to be aware of when using Jupyter notebooks: the importance of running cells in order.

Run Order Matters

As easy and user-friendly notebooks are to get started, they also make it easy to get into confusing states if you don't run cells sequentially. Assume you have the following notebook cells that are run from top to bottom:

```
In [2]: a = 1
In [3]: a
Out[3]: 1
In [4]: a = 2
```

Cell `Out[3]` prints the value `1` as expected. However, if you go back and run `In[3]` again, you will end up in this situation:

```
In [2]: a = 1
In [5]: a
Out[5]: 2
In [4]: a = 2
```

`Out[5]` now shows the value 2, which is probably not what you would expect when you read the notebook from the top, especially if cell `[4]` is further down. To prevent such cases, I would recommend rerunning not just a single cell, but all of its previous cells, too. Jupyter notebooks offer an easy way to accomplish this under the menu Run > Run All Above Selected Cell.

MARIMO NOTEBOOKS

The issue with the run order isn't the only limitation with Jupyter notebooks. marimo notebooks are a modern notebook solution that attempts to address these limitations. Here are the main advantages of marimo notebooks:

Reactive

> As we've just seen, in Jupyter notebooks, you need to make sure that you run cells in the correct order. marimo notebooks solve this issue via reactivity: whenever the value of a variable changes, all cells that depend on that variable are recalculated automatically. This, however, implies that you can't redefine your variables: you can't define `a = 1` in one cell and write `a = 2` in another cell.

Compatible with version control

> Jupyter notebooks are hard to version control with Git as they are stored in JSON format rather than as a Python file and because they include output values. marimo notebooks are regular Python files instead.

Run as script

> Jupyter notebooks were designed to run interactively via a web interface. marimo notebooks can also be run from the Terminal, accepting command-line parameters.

Run as web app

> marimo notebooks can be deployed as web apps with interactive controls while the source code is hidden.

This book sticks to Jupyter notebooks as they are still more popular than marimo notebooks, but if you want to go all in with notebooks, make sure to give marimo a try!

Now that you know how to work with Jupyter notebooks, you'll learn about how to write and run regular Python files. To do this, we'll use Visual Studio Code, a powerful text editor with great Python support.

Visual Studio Code

In this section, we'll install and configure *Visual Studio Code* (VS Code), a free text editor from Microsoft. After introducing its most important components, we'll write a first Python script and run it in a few different ways. To begin with, however, let me explain the difference between Python scripts and Jupyter notebooks and why I chose VS Code for this book.

While Jupyter notebooks are amazing for interactive workflows like researching, teaching, and experimenting, they are less ideal if you want to write Python scripts geared toward a production environment. Also, more complex projects that involve multiple files and developers are hard to manage with only Jupyter notebooks. In this case, you want to use a proper text editor to write and run regular Python files. In theory, you could use just about any text editor, but in reality, you want one that "understands" Python. That is, a text editor that supports at least the following features:

Syntax highlighting

> The editor colors words differently based on whether they represent a function, a string, a number, etc. This makes it much easier to read the code.

Autocomplete

> Autocomplete or *IntelliSense*, as Microsoft calls it, automatically suggests code completions so that you have to type less and consult documentation less frequently.

And soon enough, you have additional needs:

Run code

Switching back and forth between the text editor and an external Terminal to run your code can be a hassle.

Debugger

A debugger allows you to step through the code line by line.

Version control

Having built-in support for Git version control saves you from using an external tool.

AI assistant

AI coding assistants like GitHub Copilot can read and edit your files directly in the editor.

There is a wide spectrum of tools that can help you with all that, and as usual, every developer has different needs and preferences. Some may indeed want to use a no-frills text editor together with an external Terminal. Others may prefer an *integrated development environment* (IDE): IDEs try to put everything you'll ever need into a single tool, which can make them bloated.

I chose VS Code for this book as it has quickly become one of the most popular code editors after its initial release in 2015. What makes VS Code such a popular tool? In essence, it's the right mix between a bare-bones text editor and a full-blown IDE: VS Code is a mini IDE that comes with the essential tools out of the box, but everything else needs to be added via extensions:

Cross-platform

VS Code runs on Windows, macOS, and Linux. There are also cloud-hosted versions like GitHub Codespaces.

Integrated tools

VS Code comes with a debugger, a Terminal, support for Git version control, and has GitHub Copilot built-in.

Extensions

Additional functionality, e.g., Python support, is added via extensions that can be installed with a simple click.

Lightweight

Depending on your operating system, the VS Code installer is around 100–150 MB.

> **VISUAL STUDIO CODE VS. VISUAL STUDIO**
>
> Don't confuse Visual Studio Code with Visual Studio, the IDE! While you can use Visual Studio for Python development, it's a heavy installation and is traditionally used to work with .NET languages like C#.

To find out if you agree with my praise for VS Code, there is no better way than trying it out yourself. The next section gets you started!

Installation and Configuration

Download the installer from the VS Code home page. For the installation, use the following instructions according to your operating system. In case of an issue, please refer to the original documentation.

Windows

Double-click the installer, accept the license agreement, then accept all defaults. Once installed, open VS Code via the Windows Start menu, where you will find it under Visual Studio Code.

macOS

Double-click the ZIP file to unpack the app (some browsers, like Safari, do this automatically). Then drag and drop *Visual Studio Code.app* into

the *Applications* folder: you can now start it from the Launchpad. You need to confirm the pop-up message saying “Visual Studio Code.app is an app downloaded from the internet. Are you sure you want to open it?” by clicking the Open button.

When you open VS Code for the first time, it looks like Figure 2-4. Note that I have switched from the default dark theme to a light theme to make the screenshots easier to read. I have also closed the welcome message.

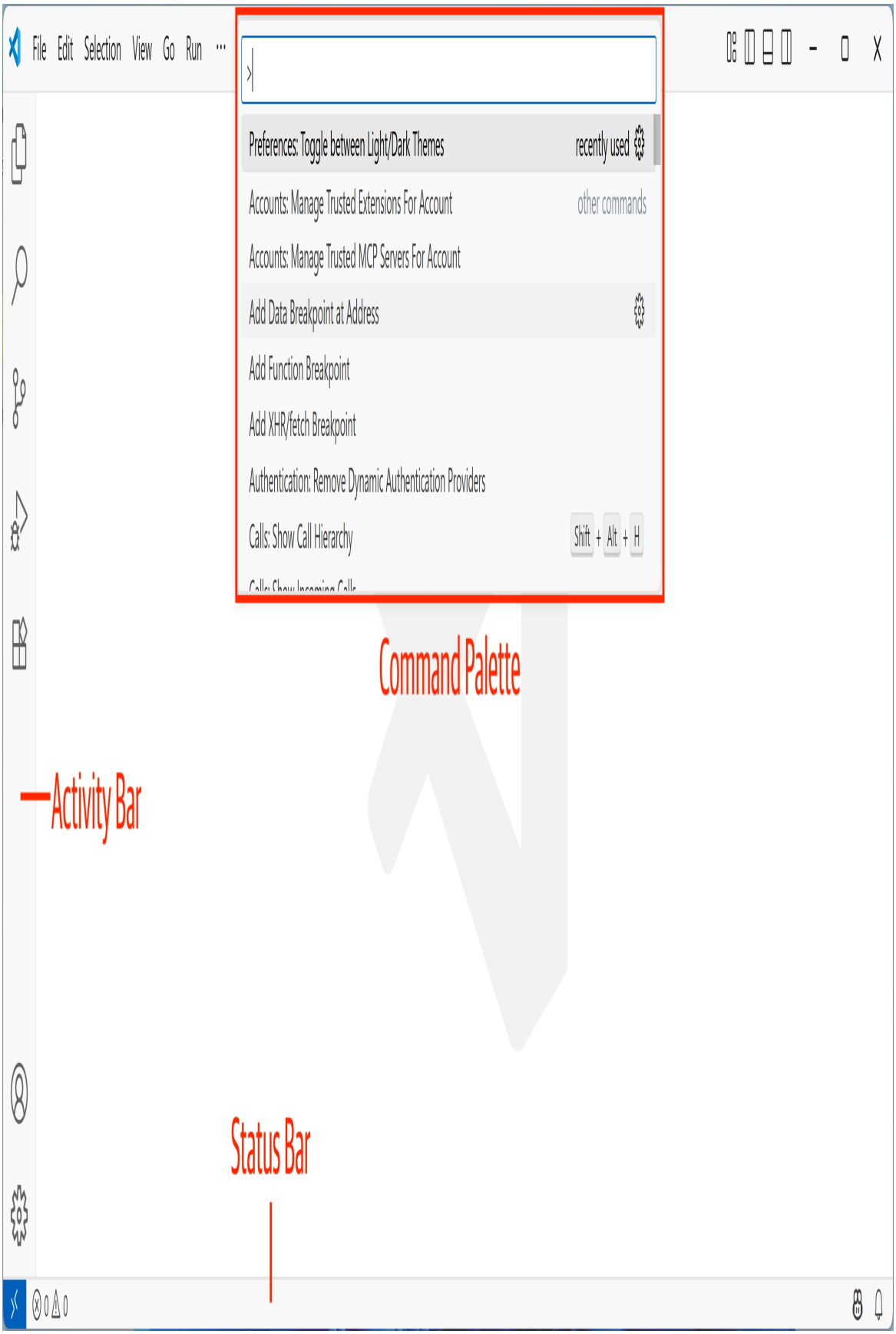
File Edit Selection View Go Run ...
>
Preferences: Toggle between Light/Dark Themes
recently used
Accounts: Manage Trusted Extensions For Account
other commands
Accounts: Manage Trusted MCP Servers For Account
Add Data Breakpoint at Address
Add Function Breakpoint
Add XHR/fetch Breakpoint
Authentication: Remove Dynamic Authentication Providers
Calls: Show Call Hierarchy
Shift + Alt + H
Command Palette
Activity Bar
Status Bar

Figure 2-4. Visual Studio Code

Here is an overview of the main menus and interface elements highlighted in Figure 2-4:

Activity Bar

On the lefthand side, you see the Activity Bar with the following icons from top to bottom:

- Explorer
- Search
- Source Control
- Run and Debug
- Extensions

Status Bar

At the bottom of the editor, you have the Status Bar. Once you have completed the configuration and edit a Python file, you will see e.g., which Python interpreter is being used.

Command Palette

You can show the Command Palette via F1 or with the keyboard shortcut Ctrl+Shift+P (Windows) or Command-Shift-P (macOS). If you are unsure about something, your first stop should always be the Command Palette, as it gives you easy access to almost everything you can do with VS Code. For example, if you are looking for keyboard shortcuts, type in **`keyboard shortcuts`**, select the entry "Help: Keyboard Shortcuts Reference," and hit Enter.

VS Code is a great text editor out of the box, but to make it work nicely with Python, you need to install two extensions:

- Python by Microsoft (core language support)
- Jupyter by Microsoft (Jupyter notebook support)

To install them, click on the Extensions icon on the Activity Bar and search for "Python." Install the Python extension that shows Microsoft as the author. Once this is installed, search for "Jupyter", again authored by Microsoft, and click Install. After installing an extension, VS Code will show a welcome document—feel free to close it. Now finalize the configuration according to your platform:

Windows

Open the Command Palette and type **`default profile`**. Select the entry that reads "Terminal: Select Default Profile" and hit Enter. In the dropdown menu, select Command Prompt and confirm by hitting Enter. This prevents VS Code to run into a permission error when activating the virtual environment in PowerShell, which is the default Terminal.

macOS

Open the Command Palette and type **`shell command`**. Select the entry that reads "Shell Command: Install code command in PATH" and hit Enter. This is required so that you can start VS Code conveniently from the Terminal.

Now that VS Code is installed and configured, let's use it to write and run our first Python script!

Running a Python Script

While you can open VS Code via the Start menu on Windows or Launchpad on macOS, it's often faster to open VS Code from the Terminal, where you are able to launch it via the `code` command. Therefore, start a

new Terminal and use `cd` to change into the directory you want to open (in our case that's the directory of the companion repository), then instruct VS Code to open the current directory, which is represented by the dot:

Windows (Terminal)

```
cd %USERPROFILE%\Documents\python-for-excel-2e
code .
```

macOS (Terminal)

```
cd ~/Documents/python-for-excel-2e
code .
```

Starting VS Code this way opens the companion repository directly in the Explorer on the Activity Bar. Note that VS Code will ask "Do you trust the authors of the files in this folder?" Confirm this dialog by clicking the "Yes, I trust the authors" button. If you don't want to use the `code` command, just start VS Code normally and open the companion repository via File > Open Folder.

When you hover over the file list in the Explorer on the Activity Bar, you will see the New File button appear as shown in Figure 2-5. Click on New File and call your file *hello_world.py*, then hit Enter. Once it opens in the editor, write the following line of code:

```
print("hello world!")
```

Remember that Jupyter notebooks print the result of the last line automatically? When you run a regular Python script, you need to tell Python explicitly what to print, which is why you need to use the `print` function here. While you are in a Python file, the Status Bar will now show your Python environment "py4xl"; this name is defined in *pyproject.toml*. Since VS Code automatically selects virtual environments in the project's

root folder, you don’t have to select the environment manually. Your setup should now look like the one in Figure 2-5.

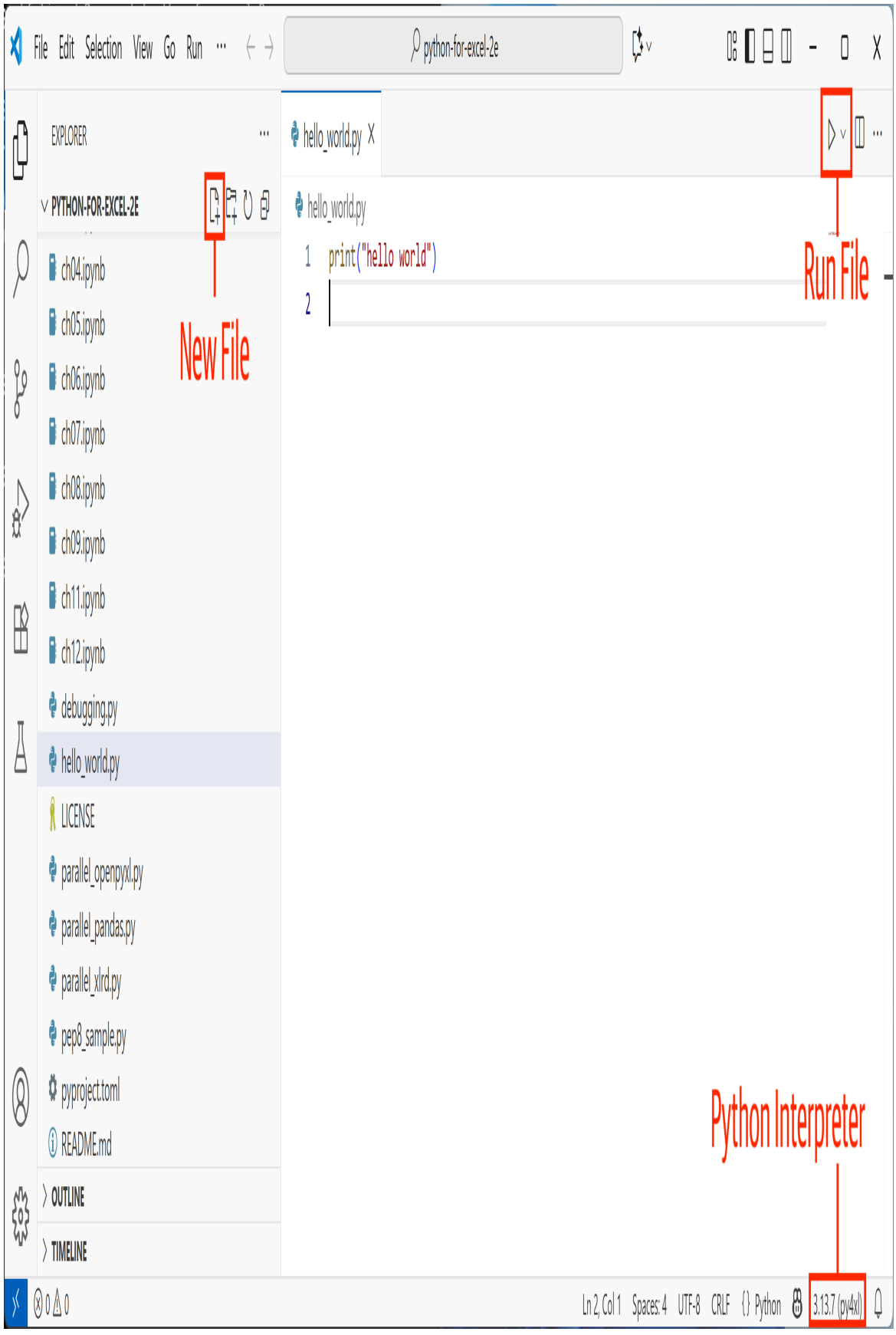
File Edit Selection View Go Run ···
python-for-excel-2e
EXPLORER
hello_world.py
PYTHON-FOR-EXCEL-2E
hello_world.py
1 print("hello world")
2
ch04.ipynb
ch05.ipynb
ch06.ipynb
ch07.ipynb
ch08.ipynb
ch09.ipynb
ch11.ipynb
ch12.ipynb
debugging.py
hello_world.py
LICENSE
parallel_openpyxl.py
parallel_pandas.py
parallel_xlrd.py
pep8_sample.py
pyproject.toml
README.md
OUTLINE
TIMELINE
New File
Run File
Python Interpreter
Ln 2, Col 1 Spaces: 4 UTF-8 CRLF {} Python 3.13.7 (py4xl)

Figure 2-5. VS Code with hello_world.py open

Before we can run the script, make sure to save it by hitting Ctrl+S on Windows or Command-S on macOS. With Jupyter notebooks, we could hit Shift+Enter to run a cell. With VS Code, you can run your code from either the Terminal or by clicking the Run button. Running Python code from the Terminal is how you usually run scripts on a server, so it's important to know how this works.

Terminal

In a Terminal, in the directory of *hello_world.py*, i.e., in the directory of the companion repository, run the script by using `uv run`:

Windows (Terminal)

```
cd %USERPROFILE%\Documents\python-for-excel-2e
uv run hello_world.py
```

macOS (Terminal)

```
cd ~/Documents/python-for-excel-2e
uv run hello_world.py
```

This will show the following output:

```
hello world!
```

If you are not in the same directory as your Python file, you need to use the full path to your Python file:

Windows (Terminal)

```
uv run %USERPROFILE%\Documents\python-for-excel-
2e\hello_world.py
```

macOS (Terminal)

```
uv run ~/Documents/python-for-excel-2e/hello_world.py
```

> **LONG FILE PATHS ON THE TERMINAL**
>
> A convenient way to deal with long file paths is to drag and drop the file onto your Terminal. This will write the full path to wherever the cursor is in the Terminal. Note that it will write the full absolute path to the Terminal, so it won't use `%USERPROFILE%` or `~`.

Instead of using an external Terminal, you can also use the Terminal that's integrated into VS Code.

Terminal in VS Code

You don't need to switch away from VS Code to work with the Terminal: VS Code has an integrated Terminal that you can show via the keyboard shortcut Ctrl+` or via View > Terminal. Since it opens in the project folder, you don't need to change the directory first:

Windows and macOS (Terminal)

```
uv run hello_world.py
```

The integrated Terminal allows you to run a Python script directly from VS Code. For an even easier experience, VS Code offers the Run button.

Run button

When you edit a Python file, you will see a Play icon at the top right—this is the Run Python File button, as shown in Figure 2-5. Clicking it will open

the Terminal at the bottom automatically and run the code.

> **OPENING FILES IN VS CODE**
>
> VS Code has an unconventional default behavior when you single-click a file in the Explorer on the Activity Bar: the file is opened in *preview mode*, which means that the next file that you single-click will open in the same tab unless you have edited the file. If you want to switch off the single-click behavior (so only a double-click will open it), go to Preferences > Settings (Ctrl+, on Windows or Command-, on macOS). Under Workbench > "List: Open Mode" set the dropdown to "doubleClick."

If you encounter an issue when running a Python script, a debugger can help you find the cause.

Debugger

If you've ever used the VBA debugger in Excel, I have good news for you: debugging with VS Code is a very similar experience. Let's start by opening the file *debugging.py* from the companion repository in VS Code. Click into the margin to the left of line number 4 so that a red dot appears—this is your breakpoint where code execution will be paused. Now click on the dropdown of the play button at the top right and select "Python Debugger: Debug Python File." This runs the code until it hits the breakpoint. The line with the breakpoint will be highlighted and code execution pauses, see Figure 2-6.

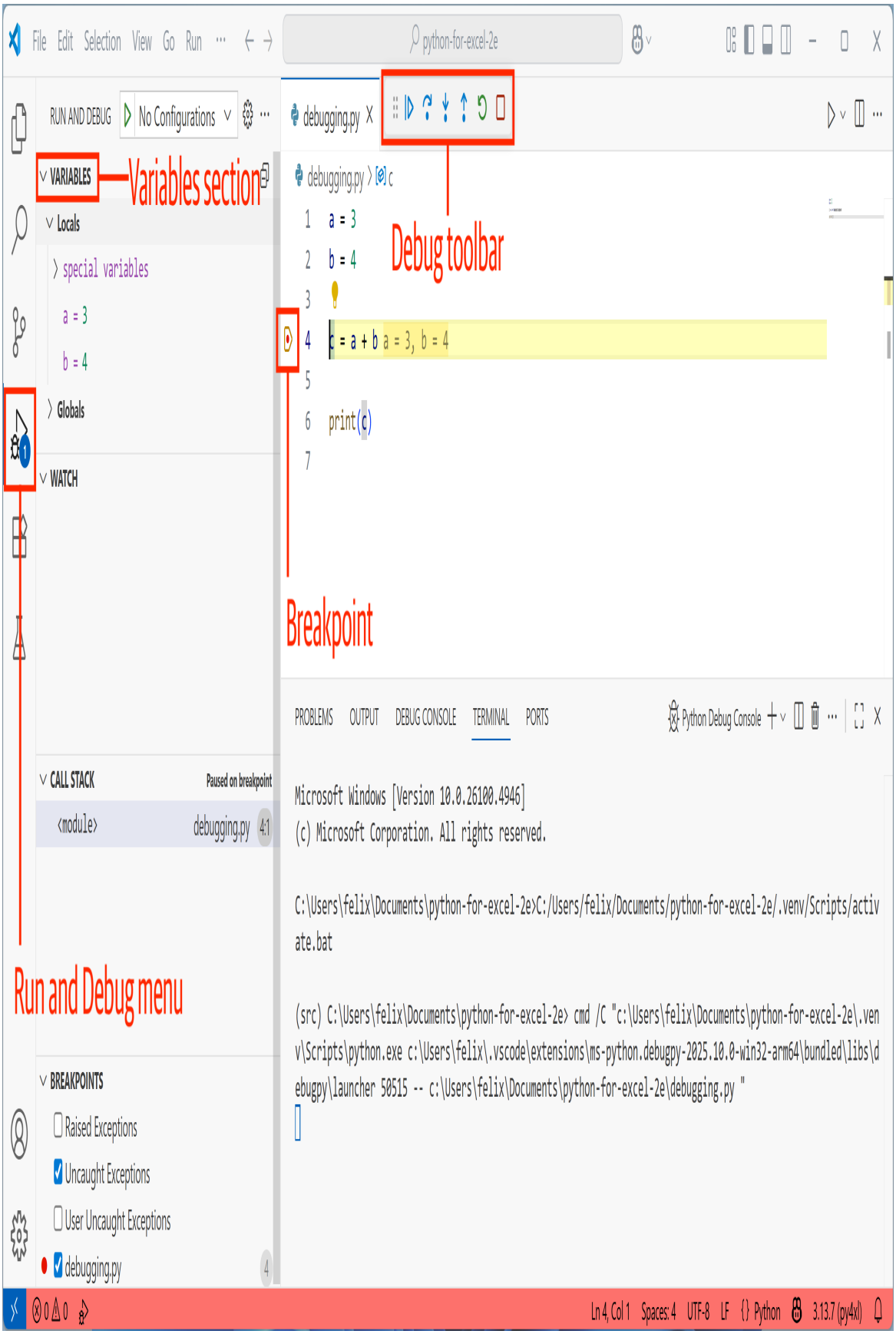
File Edit Selection View Go Run
python-for-excel-2e
RUN AND DEBUG
No Configurations
debugging.py
VARIABLES
Variables section
Locals
special variables
a = 3
b = 4
Globals
WATCH
debugging.py > c
1 a = 3
2 b = 4
3
4 c = a + b a = 3, b = 4
5
6 print(c)
7
Debug toolbar
Breakpoint
CALL STACK
Paused on breakpoint
<module>
debugging.py 4:1
Run and Debug menu
BREAKPOINTS
Raised Exceptions
Uncaught Exceptions
User Uncaught Exceptions
debugging.py
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
Python Debug Console
Microsoft Windows [Version 10.0.26100.4946]
(c) Microsoft Corporation. All rights reserved.
C:\Users\felix\Documents\python-for-excel-2e>C:/Users/felix/Documents/python-for-excel-2e/.venv/Scripts/activate.bat
(src) C:\Users\felix\Documents\python-for-excel-2e> cmd /C "c:\Users\felix\Documents\python-for-excel-2e\.venv\Scripts\python.exe c:\Users\felix\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-arm64\bundled\libs\debugpy\launcher 50515 -- c:\Users\felix\Documents\python-for-excel-2e\debugging.py "
Ln 4, Col 1
Spaces: 4
UTF-8
LF
Python
3.13.7 (py4xl)

Figure 2-6. VS Code with the debugger stopped at the breakpoint

If the Variables section doesn't show up automatically on the left, make sure to click on the Run and Debug menu to see the values of the variables. Alternatively, you can also hover over a variable in the source code and get a tooltip with its value. At the top, you will see the Debug Toolbar that gives you access to the following buttons from left to right: Continue, Step Over, Step Into, Step Out, Restart, and Stop. When you hover over them, you will see their keyboard shortcuts.

Let's see what each of these buttons does:

Continue

> This continues to run the program until it either hits the next breakpoint or the end of the program. If it reaches the end of the program, the debugging process will stop.

Step Over

> The debugger will advance one line. *Step Over* means that the debugger will not visually step through lines of code that are outside of your current scope. For example, it runs functions in one step instead of going through each line inside them.

Step Into

> If you have code that calls a function or class, etc., *Step Into* will cause the debugger to step into that function or class. If the function or class is in a different file, the debugger will open this file for you.

Step Out

> If you stepped into a function with Step Into, *Step Out* causes the debugger to return to the next higher level until eventually, you will be back on the highest level from where you called Step Into initially.

Restart

This will stop the current debug process and start a new one from the beginning.

Stop

This will stop the current debug process.

Now that you know what each button does, click on Step Over to advance one line and see how variable `c` appears in the Variables section, then finish this short debugging exercise by clicking on Continue.

At this point, you know how to create, run, and debug Python scripts in VS Code. However, VS Code can also run Jupyter notebooks—here's how this works.

Running Jupyter Notebooks with VS Code

Instead of spinning up a Jupyter notebook server in the Terminal, you can run notebooks directly in VS Code thanks to the Jupyter extension that we installed earlier. To create a new Jupyter notebook, open the Command Palette and search for "Create: New Jupyter Notebook." If you want to open an existing file instead, open it like any other file from the file explorer in VS Code. With Jupyter notebooks, VS Code shows the `py4xl` virtual environment on the top right of the notebook as the Python interpreter (in the context of a Jupyter notebook, this is called the *Jupyter kernel*), see Figure 2-7. If VS Code shows `Select Kernel` instead of the environment name, click on it to select the correct Python environment `py4xl`.

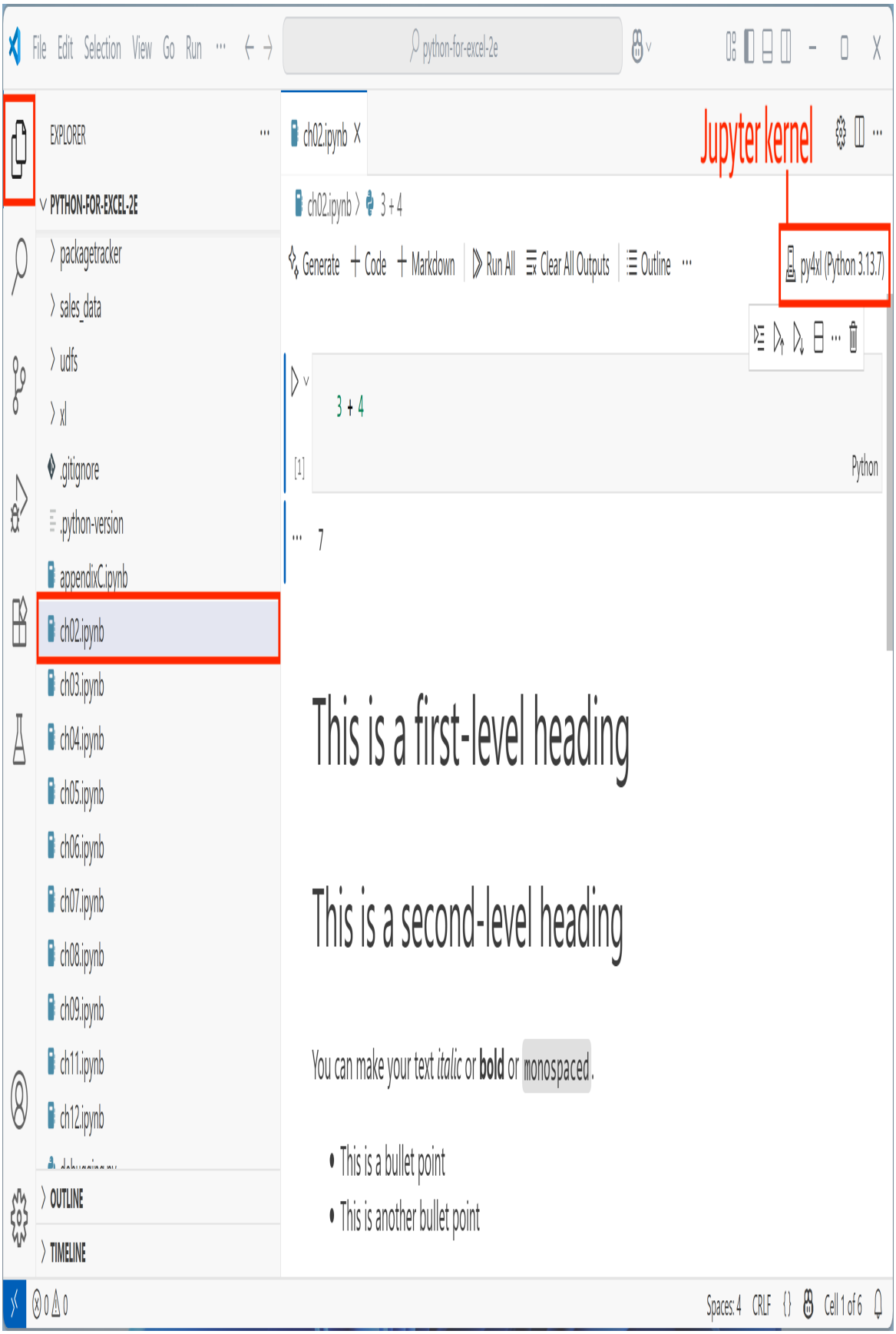
Jupyter kernel
EXPLORER
PYTHON-FOR-EXCEL-2E
packagetracker
sales_data
udfs
xl
.gitignore
.python-version
appendixC.ipynb
ch02.ipynb
ch03.ipynb
ch04.ipynb
ch05.ipynb
ch06.ipynb
ch07.ipynb
ch08.ipynb
ch09.ipynb
ch11.ipynb
ch12.ipynb
OUTLINE
TIMELINE
ch02.ipynb
Generate
Code
Markdown
Run All
Clear All Outputs
Outline
py4xl (Python 3.13.7)
3 + 4
[1]
Python
7
This is a first-level heading
This is a second-level heading
You can make your text italic or bold or monospaced.
This is a bullet point
This is another bullet point
Spaces: 4
CRLF
Cell 1 of 6

Figure 2-7. Jupyter notebooks in VS Code

Running Jupyter notebooks in VS Code makes things easier as you don't need to run the Jupyter notebook server manually in a Terminal. The main advantage, however, is that you get access to the tools that VS Code offers such as code completion, GitHub Copilot, and the debugger.

ALTERNATIVE TEXT EDITORS AND IDES

Tools are something individual, and just because this book is based on VS Code doesn't mean you shouldn't have a look at other options. Some of these options include:

- Zed is a free and fast text editor by the creators of the retired Atom editor.
- Sublime is a fast commercial text editor.
- Notepad++ is a free text editor for Windows.
- PyCharm is an IDE with a free community edition and a professional edition that adds support for scientific tools and web development.
- Spyder is similar to MATLAB's IDE and comes with a variable explorer.
- Positron is a free IDE for data science that supports both Python and R.
- JupyterLab is a web-based IDE developed by the Jupyter notebook team.
- Wing Python IDE is available in both free and paid versions.
- PyDev is originally based on the Eclipse IDE but nowadays also available as a VS Code extension.

Besides these solutions, there's a growing number of AI-centric IDEs such as Cursor. Given how rapidly this landscape is evolving, I recommend searching the web for the latest products.

Conclusion

In this chapter, apart from installing Python, I've introduced you to all the essential tools that we will work with throughout this book: the Terminal, uv, Jupyter notebooks, and VS Code. We also ran a tiny bit of Python code in a Python REPL, in a Jupyter notebook, and as a script in VS Code.

I do recommend you get comfortable with the Terminal, as it will give you a lot of power once you get used to it. Ultimately, the Terminal is where you run your Python code—whether directly, through VS Code's integrated Terminal, or via the VS Code play button.

With a working development environment, you are now ready to tackle the next chapter, where you'll learn enough Python to be able to follow the rest of the book.

Chapter 3. Getting Started with Python

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at mpotter@oreilly.com.

With Python and VS Code installed, you have everything in place to get started. Although this chapter doesn't go much further than the basics, it still covers a lot of ground. If you are at the beginning of your coding career, there may be a lot to digest. However, most concepts will get clearer once you use them in later chapters as part of a practical example, so there's no need to worry if you don't understand something fully the first time around. If you are a seasoned VBA developer, I will try to make your transition to Python a little easier by drawing parallels between the two languages and pointing out any caveats.

I'll start this chapter with Python's basic data types, such as integers and strings. After that, I will introduce indexing and slicing, a core concept in Python that gives you access to specific elements of a sequence. Up next are data structures like lists and dictionaries that can hold multiple objects. I'll continue with the `if` statement and the `for` and `while` loops before getting to an introduction of functions and modules that allow you to

organize and structure your code. To wrap this chapter up, I will show you how to format your Python code properly. As you have probably guessed by now, this chapter is as technical as it can get. Running the examples for yourself in a Jupyter notebook is therefore a good idea to make everything a bit more interactive and playful. Either type the examples yourself or run them by using the notebook *ch03.ipynb* from the companion repository.

Data Types

Python, like every other programming language, treats numbers, text, booleans, etc. differently by assigning them a different *data type*. The data types that we will use most often are integers, floats, booleans, and strings. In this section, I am going to introduce them one after another with a few examples. To be able to understand data types, though, I first need to explain what an object is.

Objects

In Python, *everything* is an object, including numbers, strings, functions, and everything else that we'll meet in this chapter. Objects can make complex things easy and intuitive by giving you access to a set of variables and functions. The variables hold the object's data (often called *state*), while the functions let you perform actions. So before anything else, let me say a few words about variables and functions!

Variables

In Python, a *variable* is a name that you assign to an object by using the equal sign. In the first line of the following example, the name `a` is assigned to the object `3`:

```
In [1]: a = 3
        b = 4
        a + b
Out[1]: 7
```

This works the same for all objects, which is simpler compared to VBA, where you use the equal sign for data types like numbers and strings and the `Set` statement for objects like workbooks or worksheets. In Python, you change a variable's type simply by assigning it to a new object. This is referred to as *dynamic typing*:

```
In [2]: a = 3
        print(a)
        a = "three"
        print(a)
3
three
```

Unlike VBA, Python is case-sensitive, so `a` and `A` are two different variables. Variable names must follow certain rules:

- They must start with either a letter or an underscore
- They must consist of letters, numbers, and underscores

After this short introduction to variables, let's see how we can make function calls!

Functions

I will introduce functions with a lot more detail later in this chapter. For now, you should simply know how to call built-in functions like `print` that we used in the previous code sample. To call a function, you add parentheses to the function name and provide the arguments within the parentheses, which is pretty much equivalent to the mathematical notation:

```
function_name(argument1, argument2, ...)
```

Let's now look at how variables and functions work in the context of objects!

Attributes and methods

In the context of objects, variables are called *attributes* and functions are called *methods*: attributes give you access to the data of an object, and methods allow you to perform an action. To access attributes and methods, you use the dot notation like this: `myobject.attribute` and `myobject.method()`.

Let's make this a bit more tangible: if you write a car racing game, you would most likely use an object that represents a car. The `car` object could have a `speed` attribute that allows you to get the current speed via `car.speed`, and you might be able to accelerate the car by calling the accelerate method `car.accelerate(10)`, which would increase the speed by ten miles per hour.

The type of an object and with that its behavior is defined by a *class*, so the previous example would require you to write a `Car` class. The process of getting a `car` object out of a `Car` class is called *instantiation*, and you instantiate an object by calling the class in the same way as you call a function: `car = Car()`. We won't write our own classes in this book, but if you are interested in how this works, have a look at Appendix C.

We will use a first object method in the next section to make a string—i.e., a piece of text—uppercase, and we will get back to the topic of objects and classes when we talk about `datetime` objects toward the end of this chapter. Now, however, let's move on with those objects that have a numeric data type!

Numeric Types

The data types `int` and `float` represent *integers* and *floating-point numbers*, respectively. To find out what data type an object has, use the built-in function `type`:

```
In [3]: type(4)
Out[3]: int
In [4]: type(4.4)
Out[4]: float
```

If you want to force a number to be a `float` instead of an `int`, it's good enough to use a trailing decimal point or the `float` *constructor* (a constructor is a function that creates a specific object):

```
In [5]: type(4.)
Out[5]: float
In [6]: float(4)
Out[6]: 4.0
```

The last example can also be turned around: using the `int` constructor, you can turn a `float` into an `int`. If the fractional part is not zero, it will be truncated:

```
In [7]: int(4.9)
Out[7]: 4
```

EXCEL CELLS ALWAYS STORE FLOATS

You may need to convert a `float` to an `int` when you read in a number from an Excel cell and provide it as an argument to a Python function that expects an integer. The reason is that numbers in Excel cells are always stored as floats behind the scenes, even if Excel shows you what looks like an integer.

Python has a few more numeric types that I won't discuss in this book: there are the `decimal`, `fraction`, and `complex` data types. If floating-point inaccuracies are an issue (see sidebar), use the `decimal` type for exact results. These cases are rare, though. As a rule of thumb: if Excel would be good enough for the calculations, use floats.

FLOATING-POINT INACCURACIES

By default, Excel often shows rounded numbers: type **`=1.125-1.1`** into a cell, and you will see `0.025`. While this might be what you expect, it is not what Excel stores internally. Change the display format to show at least 16 decimals, and it will change to `0.0249999999999999`. This is the effect of *floating-point inaccuracy*: computers live in a binary world, i.e., they calculate only with zeros and ones. Certain decimal fractions like `0.1` can't be stored as a finite binary floating-point number, which explains the result from the subtraction. In Python, you will see the same effect, but Python doesn't hide the decimals from you:

```
In [8]: 1.125 - 1.1
Out[8]: 0.02499999999999991
```

Mathematical operators

Calculating with numbers requires the use of mathematical operators like the plus or minus sign. Except for the power operator, there shouldn't be any surprise if you come from Excel:

```
In [9]: 3 + 4  # Sum
Out[9]: 7
In [10]: 3 - 4  # Subtraction
Out[10]: -1
In [11]: 3 / 4  # Division
Out[11]: 0.75
In [12]: 3 * 4  # Multiplication
Out[12]: 12
In [13]: 3**4  # The power operator (Excel uses 3^4)
Out[13]: 81
In [14]: 3 * (3 + 4)  # Use of parentheses
Out[14]: 21
```

COMMENTS

In the previous examples, I was describing the type of operation by using *comments* (e.g., `# Sum`). Comments help other people (and yourself a few weeks after writing the code) to understand what's going on in your program. My advice is to only comment those things that are not already evident from reading the code. AI-generated code often comes with a lot of comments—it doesn't hurt to remove those that aren't adding any value. Anything starting with a hash sign is a comment in Python and is ignored when you run the code:

```
In [15]: # This is a sample we've seen before.
         # Every comment line has to start with a #
         3 + 4
Out[15]: 7
In [16]: 3 + 4  # This is an inline comment
Out[16]: 7
```

Most editors have a keyboard shortcut to comment/uncomment lines. In Jupyter notebooks and VS Code, it is Ctrl+/ (Windows) or Command-/ (macOS). Note that Markdown cells in Jupyter notebooks won't accept comments—if you start a line with a # there, Markdown will interpret this as a heading.

With integers and floats covered, let's move straight to booleans!

Booleans

The boolean types in Python are `True` or `False`, exactly like in VBA. The boolean operators `and`, `or`, and `not`, however, are all lowercase, while VBA shows them capitalized. Boolean expressions are similar to how they work in Excel formulas, except for equality and inequality operators:

```
In [17]: 3 == 4  # Equality (Excel uses 3 = 4)
Out[17]: False
In [18]: 3 != 4  # Inequality (Excel uses 3 <> 4)
Out[18]: True
```

```
In [19]: 3 < 4  # Smaller than. Use > for bigger than.
Out[19]: True
In [20]: 3 <= 4  # Smaller or equal. Use >= for bigger or equal.
Out[20]: True
In [21]: # You can chain logical expressions
         # In VBA, this would be: 10 < 12 And 12 < 17
         # In Excel formulas, this would be: =AND(10 < 12, 12 <
17)
         10 < 12 < 17
Out[21]: True
In [22]: not True  # "not" operator
Out[22]: False
In [23]: False and True  # "and" operator
Out[23]: False
In [24]: False or True  # "or" operator
Out[24]: True
```

Every Python object evaluates to either `True` or `False`. The majority of objects are `True`, but there are some that evaluate to `False` including `None` (see sidebar), `False`, `0` or empty data types, e.g., an empty string or an empty list (I'll introduce strings and lists in a moment).

NONE

`None` is a built-in constant and represents "the absence of a value" according to the official docs. For example, if a function does not explicitly return anything, it returns `None`. It is also a good choice to represent empty cells in Excel as we will see in Part 3 and Part 4.

To double-check if an object is `True` or `False`, use the `bool` constructor:

```
In [25]: bool(2)
Out[25]: True
In [26]: bool(0)
Out[26]: False
In [27]: bool("some text")  # We'll get to strings in a moment
Out[27]: True
In [28]: bool("")
Out[28]: False
In [29]: bool(None)
Out[29]: False
```

With booleans in our pocket, there is one more basic data type left: text data, better known as *strings*.

Strings

If you have ever worked with strings in VBA that are longer than one line and contain variables and literal quotes, you probably wished it was easier. Fortunately, this is an area where Python is particularly strong. Strings can be expressed by using either double quotes (`"`) or single quotes (`'`). The only condition is that you have to start and end the string with the same type of quotes. You can use + to join ("concatenate") strings or `*` to repeat them. Since we were repeating strings in the previous chapter, here is a sample using the plus sign:

```
In [30]: "A double quote string. " + 'A single quote string.'
Out[30]: 'A double quote string. A single quote string.'
```

Depending on what you want to write, using single or double quotes can help you to easily print literal quotes without the need to escape them. If you still need to escape a character, you precede it with a backslash:

```
In [31]: print("Don't wait! " + 'Learn how to "speak" Python.')
Don't wait! Learn how to "speak" Python.
In [32]: print("It's easy to \"escape\" characters with a leading
\\.")
It's easy to "escape" characters with a leading \.
```

When you are mixing strings with variables, you usually work with *f-strings*, short for *formatted string literal*. Simply put an `f` in front of your string and use variables in between curly braces:

```
In [33]: # Note how Python allows you to conveniently assign
multiple
         # values to multiple variables in a single line
         first_adjective, second_adjective = "free", "open
source"
         f"Python is {first_adjective} and {second_adjective}."
Out[33]: 'Python is free and open source.'
```

As I mentioned at the beginning of this section, strings are objects like everything else, and they offer a few methods (i.e., functions) to perform an action on that string. For example, this is how you transform between upper and lowercase letters:

```
In [34]: "PYTHON".lower()
Out[34]: 'python'
In [35]: "python".upper()
Out[35]: 'PYTHON'
```

GETTING HELP

How do you know what attributes certain objects like strings offer and what arguments their methods accept? The answer depends on the tool you use: with classic Jupyter notebooks, hit the Tab key after typing the dot that follows an object, for example `"python".<Tab>`. This will make a dropdown appear with all the attributes and methods that this object offers. If your cursor is in a method, for example within the parentheses of `"python".upper()`, hit Shift+Tab to get the description of that function. VS Code will display this information automatically as a tooltip. If you run a Python REPL, use `dir("python")` to get the available attributes and `help("python".upper)` to print the description of the `upper` method. Other than that, it's always a good idea to get back to Python's online documentation. If you are looking for the documentation of third-party packages like pandas, it's helpful to search for them on PyPI, Python's package index, where you will find the links to the respective home pages and documentation. Nowadays, AI assistants like ChatGPT or Microsoft Copilot are also a great way to get support if you are lost, see Chapter 8.

When working with strings, you often need to select parts of a string: for example, you may want to get the `USD` part out of the `EURUSD` exchange rate notation. The next section shows you Python's powerful indexing and slicing mechanism that allows you to do this easily.

Indexing and Slicing

Indexing and slicing give you access to specific elements of a sequence. Since strings are sequences of characters, we can use them to learn how it works. In the next section, we will meet additional sequences like lists and tuples that support indexing and slicing too.

Indexing

Figure 3-1 introduces the concept of *indexing*. Python is zero-based, which means that the first element in a sequence is referred to by index `0`. Negative indices from `-1` allow you to refer to elements from the end of the sequence.

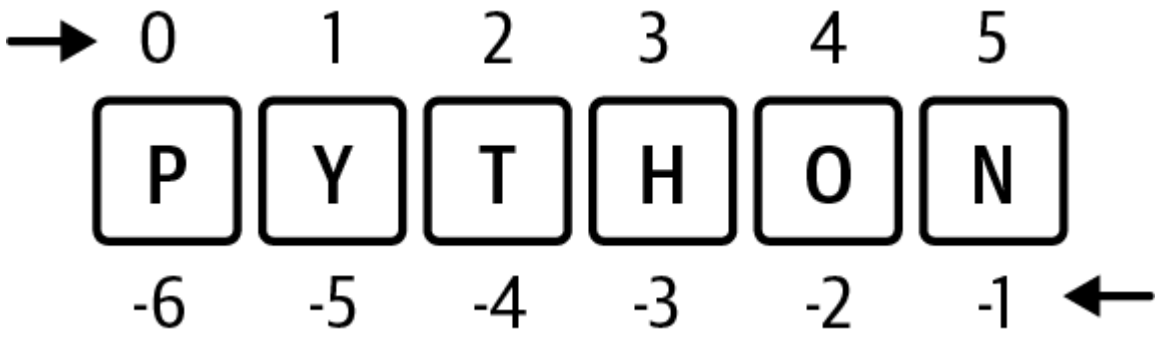

Figure 3-1. Indexing from the beginning and end of a sequence

> **COMMON ERROR TRAPS FOR VBA DEVELOPERS**
>
> If you are coming from VBA, indexing is a common error trap. VBA uses one-based indexing for most collections like sheets (`Sheets(1)`) but uses zero-based indexing for arrays (`MyArray(0)`), although that default can be changed. Another difference is that VBA uses parentheses for indexing while Python uses square brackets.

The syntax for indexing is as follows:

```
sequence[index]
```

Accordingly, you access specific elements from a string like this:

```
In [36]: language = "PYTHON"
In [37]: language[0]
Out[37]: 'P'
In [38]: language[1]
Out[38]: 'Y'
In [39]: language[-1]
Out[39]: 'N'
In [40]: language[-2]
Out[40]: 'O'
```

If you want to extract more than just a single character, you have to use slicing.

Slicing

If you want to get more than one element from a sequence, use the *slicing* syntax, which works as follows:

```
sequence[start:stop:step]
```

Python uses half-open intervals: the `start` index is included while the `stop` index is not. If you leave the `start` or `stop` arguments away, it will include everything from the beginning or to the end of the sequence, respectively. `step` determines the direction and the step size: for example, `2` will return every second element and `-3` will return every third element starting from the end of the sequence. The default step size is one:

```
In [41]: language[:3]  # Same as language[0:3]
Out[41]: 'PYT'
In [42]: language[1:3]
Out[42]: 'YT'
In [43]: language[-3:]  # Same as language[-3:6]
Out[43]: 'HON'
In [44]: language[-3:-1]
Out[44]: 'HO'
In [45]: language[::2]  # Every second element
Out[45]: 'PTO'
In [46]: language[-1:-4:-1]  # A negative step goes from end to
beginning
Out[46]: 'NOH'
```

So far we've looked at just a single index or slice operation, but Python also allows you to *chain* multiple index and slice operations together. For example, if you want to get the second character out of the last three characters, you could do it like this:

```
In [47]: language[-3:][1]
Out[47]: 'O'
```

This is the same as `language[-2]` so in this case, it wouldn't make much sense to use chaining, but it will make more sense when we use indexing and slicing with lists, one of the data structures that I am going to introduce in the next section.

Data Structures

Python provides powerful data structures that make it easy to work with collections of objects. In this section, I am going to introduce lists, dictionaries, tuples, and sets. While each of these data structures has slightly different characteristics, they are all able to hold multiple objects. In VBA, you may have used collections or arrays to hold multiple values. VBA even offers a data structure called dictionary that works conceptually the same as Python's dictionary. It is, however, only available on the Windows version of Excel. Let's get started with lists, the data structure that you will probably use most.

Lists

Lists are capable of holding multiple objects of different data types. They are so versatile that you will use them all the time. You create a list as follows:

```
[element1, element2, ...]
```

Here are two lists, one with the names of Excel files and the other one with a few numbers:

```
In [48]: file_names = ["one.xlsx", "two.xlsx", "three.xlsx"]
         numbers = [1, 2, 3]
```

Like strings, lists can easily be merged ("concatenated") with the plus sign. This also shows you that lists can hold different types of objects:

```
In [49]: file_names + numbers
Out[49]: ['one.xlsx', 'two.xlsx', 'three.xlsx', 1, 2, 3]
```

As lists are objects like everything else, they can also have other lists as their elements. I will refer to them as *nested lists*:

```
In [50]: nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

If you rearrange this to span over multiple lines, you can easily recognize that this is a very nice representation of a matrix, or a range of spreadsheet cells. Note that the square brackets implicitly allow you to break the lines (see sidebar). Via indexing and slicing, you get the elements you want:

```
In [51]: cells = [[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]]
In [52]: cells[1]  # Second row
Out[52]: [4, 5, 6]
In [53]: cells[1][1:]  # Second row, second and third column
Out[53]: [5, 6]
```

LINE CONTINUATION

Sometimes, a line of code can get so long that you will need to break it up into two or more lines to keep your code readable. Technically, you can either use parentheses or a backslash to break up the line:

```
In [54]: a = (1 + 2
              + 3)
In [55]: a = 1 + 2 \
             + 3
```

Python's style guide, however, prefers that you use *implicit line breaks* if possible: whenever you are using an expression that contains parentheses, square brackets, or curly braces, use them to create a line break without having to introduce an additional character. I will say more about Python's style guide toward the end of this chapter.

You can change elements in lists:

```
In [56]: users = ["Linda", "Brian"]
In [57]: users.append("Jennifer")  # Most commonly you add to the
end
         users
Out[57]: ['Linda', 'Brian', 'Jennifer']
In [58]: users.insert(0, "Kim")  # Insert "Kim" at index 0
         users
Out[58]: ['Kim', 'Linda', 'Brian', 'Jennifer']
```

To delete an element, use either `pop` or `del`. While `pop` is a method, `del` is implemented as a statement in Python:

```
In [59]: users.pop()  # Removes and returns the last element by
default
Out[59]: 'Jennifer'
In [60]: users
Out[60]: ['Kim', 'Linda', 'Brian']
In [61]: del users[0]  # del removes an element at the given
index
```

Some other useful things you can do with lists are:

```
In [62]: len(users)  # Length
Out[62]: 2
In [63]: "Linda" in users  # Check if users contains "Linda"
Out[63]: True
In [64]: print(sorted(users))  # Returns a new sorted list
         print(users)  # The original list is unchanged
['Brian', 'Linda']
['Linda', 'Brian']
In [65]: users.sort()  # Sorts the original list
         users
Out[65]: ['Brian', 'Linda']
```

Note that you can use `len` and `in` with strings as well:

```
In [66]: len("Python")
Out[66]: 6
In [67]: "free" in "Python is free and open source."
Out[67]: True
```

To get access to elements in a list, you refer to them by their index, i.e., position—that's not always practical. Dictionaries, the topic of the next section, allow you to get access to elements via a key (often a name).

Dictionaries

Dictionaries map keys to values. You will come across key/value combinations all the time. The easiest way to create a dictionary is as follows:

```
{key1: value1, key2: value2, ...}
```

While lists allow you to access elements by index, i.e., position, dictionaries allow you to access elements by key. As with indices, keys are accessed via square brackets. The following code samples will use the dictionary `exchange_rates`, which maps currency pairs ("key") to exchange rates ("value"):

```
In [68]: exchange_rates = {"EURUSD": 1.1152, "GBPUSD": 1.2454,
"AUDUSD": 0.6161}
In [69]: exchange_rates["EURUSD"]  # Access the EURUSD exchange
rate
Out[69]: 1.1152
```

To change existing values or add new key/value pairs, do the following:

```
In [70]: exchange_rates["EURUSD"] = 1.2  # Change an existing
value
         exchange_rates
Out[70]: {'EURUSD': 1.2, 'GBPUSD': 1.2454, 'AUDUSD': 0.6161}
In [71]: exchange_rates["CADUSD"] = 0.714  # Add a new key/value
pair
         exchange_rates
Out[71]: {'EURUSD': 1.2, 'GBPUSD': 1.2454, 'AUDUSD': 0.6161,
'CADUSD': 0.714}
```

To merge two or more dictionaries, use the pipe operator. If the second dictionary contains keys from the first one, the values from the first will be overridden. You can see this happening in the following example by looking at the `GBPUSD` exchange rate:

```
In [72]: exchange_rates | {"SGDUSD": 0.7004, "GBPUSD": 1.2222}
Out[72]: {'EURUSD': 1.2,
          'GBPUSD': 1.2222,
          'AUDUSD': 0.6161,
          'CADUSD': 0.714,
          'SGDUSD': 0.7004}
```

Many objects can serve as keys; the following is an example with integers:

```
In [73]: currencies = {1: "EUR", 2: "USD", 3: "AUD"}
In [74]: currencies[1]
Out[74]: 'EUR'
```

By using the `get` method, dictionaries allow you to use a default value in case the key doesn't exist:

```
In [75]: # currencies[100] would raise an exception. Instead of
100,
```

```
In [75]: # you could use any other non-existing key, too.
         currencies.get(100, "N/A")
Out[75]: 'N/A'
```

Dictionaries can often be used when you would use a `Case` statement in VBA. The previous example could be written like this in VBA:

```
ccy = 100
Select Case ccy
Case 1
    Debug.Print "EUR"
Case 2
    Debug.Print "USD"
Case 3
    Debug.Print "AUD"
Case Else
    Debug.Print "N/A"
End Select
```

Now that you know how to work with dictionaries, let's move on to the next data structure: tuples. They are similar to lists except for one big difference.

Tuples

Tuples are similar to lists with the difference that they are *immutable*: once created, their elements can't be changed. While you can often use tuples and lists interchangeably, tuples are the obvious choice for a collection that never changes throughout the program. Technically, tuples are created by separating values with commas:

```
mytuple = element1, element2, ...
```

However, using parentheses often makes it easier to read:

```
In [76]: currencies = ("EUR", "GBP", "AUD")
```

Tuples allow you to access elements the same way as lists, but they won't allow you to change elements. Instead, merging ("concatenating") tuples

will create a new tuple behind the scenes, then bind your variable to this new tuple:

```
In [77]: currencies[0]  # Accessing the first element
Out[77]: 'EUR'
In [78]: # Concatenating tuples will return a new tuple
         currencies + ("SGD",)
Out[78]: ('EUR', 'GBP', 'AUD', 'SGD')
```

I explain the difference between *mutable* vs. *immutable* objects in detail in Appendix C, but for now, let's have a look at the last data structure of this section: sets.

Sets

Sets are collections that have no duplicate elements. While you can use them for set theory operations, in practice they often help you to get the unique values of a list or a tuple. You create sets by using curly braces:

```
{element1, element2, ...}
```

To get the unique objects in a list or a tuple, use the `set` constructor like so:

```
In [79]: set(["USD", "USD", "SGD", "EUR", "USD", "EUR"])
Out[79]: {'EUR', 'SGD', 'USD'}
```

Other than that, you can apply set theory operations like union and intersection:

```
In [80]: portfolio1 = {"USD", "EUR", "SGD", "CHF"}
         portfolio2 = {"EUR", "SGD", "CAD"}
In [81]: # Same as portfolio2.union(portfolio1)
         portfolio1.union(portfolio2)
Out[81]: {'CAD', 'CHF', 'EUR', 'SGD', 'USD'}
In [82]: # Same as portfolio2.intersection(portfolio1)
         portfolio1.intersection(portfolio2)
Out[82]: {'EUR', 'SGD'}
```

For a full overview of set operations, see the official docs. Before moving on, let's quickly revise the four data structures we just met in Table 3-1. It shows a sample for each data structure in the notation I used in the previous paragraphs, the so-called *literals*. Additionally, I am also listing their constructors that offer an alternative to using the literals and are often used to convert from one data structure to another. For example, to convert a tuple to a list, do:

```
In [83]: currencies = ("USD", "EUR", "CHF")
         currencies
Out[83]: ('USD', 'EUR', 'CHF')
In [84]: list(currencies)
Out[84]: ['USD', 'EUR', 'CHF']
```

Table 3-1. Data structures

Data Structure	Literals	Constructor
List	`[1, 2, 3]`	`list((1, 2, 3))`
Dictionary	`{"a": 1, "b": 2}`	`dict(a=1, b=2)`
Tuple	`(1, 2, 3)`	`tuple([1, 2, 3])`
Set	`{1, 2, 3}`	`set((1, 2, 3))`

At this point, you know all important data types including basic ones like floats and strings, and data structures like lists and dictionaries. Next, you'll learn how to make your code respond to different conditions and how to write loops.

Control Flow

This section presents the `if` statement as well as the `for` and `while` loops. The `if` statement allows you to execute certain lines of code only if a condition is met, and the `for` and `while` loops will execute a block of code repeatedly. To use these control flow statements, however, you need to understand one of Python's most noteworthy particularities: significant white space. So let's start with this!

Code Blocks and the pass Statement

A *code block* defines a section in your source code that is used for something special. For example, you use a code block to define the lines over which your program is looping or it makes up the definition of a function. In Python, you define code blocks by indenting them, not by using keywords (like in VBA) or curly braces (like in most other languages). This is referred to as *significant white space*. The Python community has settled on four spaces as indentation, but you usually type them in by hitting the Tab key: both Jupyter notebooks and VS Code will automatically convert tabs into four spaces. Let me show you how code blocks are formally defined by using the `if` statement:

```
if condition:
    pass  # Do nothing
```

The line preceding the code block always terminates with a colon. Since the end of the code block is reached when you no longer indent the line, you need to use the `pass` statement if you want to create a dummy code block that does nothing. In VBA, this would correspond to the following:

```
If condition Then
    ' Do nothing
End If
```

Note that "Do nothing" is a comment in both versions, and could be left out. Now that you know how to define code blocks, I can properly introduce the `if` statement.

The if Statement and Conditional Expressions

To introduce the `if` statement, let me reproduce the example from "Readability Counts" in Chapter 1, but this time in Python:

```
In [85]: i = 20
         if i < 5:
             print("i is smaller than 5")
         elif i <= 10:
             print("i is between 5 and 10")
         else:
             print("i is bigger than 10")
i is bigger than 10
```

If you would do the same as we did in Chapter 1, i.e., indent the `elif` and `else` statements, you would get a `SyntaxError`. Python won't let you indent your code differently from the logic. Compared to VBA, the keywords are lowercase and instead of `ElseIf` in VBA, Python uses `elif`. Note that in Python, a simple `if` statement doesn't require any parentheses around it and to test if a value is `True`, you don't need to do that explicitly. Here is what I mean by that:

```
In [86]: is_important = True
         if is_important:
             print("This is important.")
         else:
             print("This is not important.")
This is important.
```

The same works if you want to check if a sequence like a list is empty or not:

```
In [87]: values = []
         if values:
             print(f"The following values were provided:
{values}")
         else:
             print("There were no values provided.")
There were no values provided.
```

Conditional expressions, also called *ternary operators*, allow you to use a more compact style for simple `if`/`else` statements:

```
In [88]: is_important = False
         print("important") if is_important else print("not
important")
not important
```

With `if` statements and conditional expressions in our pocket, let's move on to `for` and `while` loops.

The for and while Loops

If you need to do something repeatedly, a `for` loop can often do the work for you. `for` loops iterate over the items of a sequence like a list, a tuple, or a string (remember, strings are sequences of characters). As an introductory example, let's create a `for` loop that takes each element of the `currencies` list, assigns it to the variable `currency` and prints it—one after another until there are no more elements in the list:

```
In [89]: currencies = ["USD", "HKD", "AUD"]

         for currency in currencies:
             print(currency)
USD
HKD
AUD
```

As a side note, VBA's `For Each` statement is close to how Python's `for` loop works. The previous example could be written like this in VBA:

```
Dim currencies As Variant
Dim ccy As Variant  'currency is a reserved word in VBA

currencies = Array("USD", "HKD", "AUD")

For Each ccy In currencies
    Debug.Print ccy
Next
```

In Python, if you need a variable that keeps track of the number of iterations in a `for` loop ("counter variable"), the `range` or `enumerate` built-in functions can help you with that. Let's first look at `range`, which provides a sequence of numbers: you call it by either providing a single `stop` argument or by providing a `start` and `stop` argument, with an optional `step` argument. `start` is inclusive, `stop` is exclusive, and `step` determines the step size, with `1` being the default:

```
range(stop)   # Option 1
range(start, stop, step)   # Option 2
```

`range` evaluates lazily, which means that it generates numbers on the fly and without explicitly asking for it, you won't see the whole sequence. This lazy evaluation is memory-efficient, as `range` doesn't need to store all numbers at once:

```
In [90]: range(5)
Out[90]: range(0, 5)
```

Converting the range to a list makes the values visible:

```
In [91]: list(range(5))   # stop argument
Out[91]: [0, 1, 2, 3, 4]
In [92]: list(range(2, 5, 2))   # start, stop, step arguments
Out[92]: [2, 4]
```

Normally, you want to use `range` directly though, to take advantage of its memory-efficiency:

```
In [93]: for i in range(3):
             print(i)
0
1
2
```

If you need a counter variable while looping over a sequence, use `enumerate`. It returns a sequence of `(index, element)` tuples. By

default, the index starts at zero and increments by one. You can use `enumerate` in a loop like this:

```
In [94]: for i, currency in enumerate(currencies):
             print(i, currency)
0 USD
1 HKD
2 AUD
```

Looping over tuples and sets works the same as with lists. When you loop over dictionaries, however, Python will loop over the keys:

```
In [95]: exchange_rates = {"EURUSD": 1.1152, "GBPUSD": 1.2454,
"AUDUSD": 0.6161}
         for currency_pair in exchange_rates:
             print(currency_pair)
EURUSD
GBPUSD
AUDUSD
```

Use the `items` method to get the key and the value as a tuple:

```
In [96]: for currency_pair, exchange_rate in
exchange_rates.items():
             print(currency_pair, exchange_rate)
EURUSD 1.1152
GBPUSD 1.2454
AUDUSD 0.6161
```

To exit a loop immediately, use the `break` statement:

```
In [97]: for i in range(15):
             if i == 2:
                 break
             else:
                 print(i)
0
1
```

You skip the remainder of an iteration with the `continue` statement, which means that the loop continues with the next element `i`:

```
In [98]: for i in range(4):
             if i == 2:
                 continue
             else:
                 print(i)
0
1
3
```

When comparing for loops in VBA with Python, there is a subtle difference: in VBA, the counter variable increases beyond your upper limit after finishing the loop:

```
For i = 1 To 3
    Debug.Print i
Next i
Debug.Print i
```

This prints:

```
1
2
3
4
```

In Python, it behaves like you would probably expect it to:

```
In [99]: for i in range(1, 4):
             print(i)
         print(i)
1
2
3
3
```

Instead of looping over a sequence, you can also use *while loops* to run a loop while a certain condition is true:

```
In [100]: n = 0
          while n <= 2:
              print(n)
              n += 1
```

```
0
1
2
```

> **AUGMENTED ASSIGNMENT**
>
> I have used the *augmented assignment* notation in the last example: `n += 1`. This is the same as if you would write `n = n + 1`. This also works with the other mathematical operators that I've introduced earlier on; for example, `n = n - 2` can be written as `n -= 2`.

Quite often, you will need to collect certain elements in a list for further processing. In this case, Python offers an alternative to writing loops: list, dictionary, and set comprehensions.

List, Dictionary, and Set Comprehensions

List, dictionary, and set comprehensions are technically a way to create the respective data structure, but they often replace a `for` loop, which is why I am introducing them here. As an introductory example, in the following list of USD currency pairs, you need to select those currencies where USD is quoted as the second currency. You could write the following `for` loop:

```
In [101]: currency_pairs = [
              "USDJPY",
              "USDGBP",
              "USDCHF",
              "USDCAD",
              "AUDUSD",
              "NZDUSD",
          ]
In [102]: usd_quote = []
          for pair in currency_pairs:
              if pair[3:] == "USD":
                  usd_quote.append(pair[:3])
          usd_quote
Out[102]: ['AUD', 'NZD']
```

This is often easier to write with a *list comprehension*. A list comprehension is a concise way of creating a list. You can grab its syntax from the following example, which does the same as the previous `for` loop:

```
In [103]: [pair[:3] for pair in currency_pairs if pair[3:] ==
"USD"]
Out[103]: ['AUD', 'NZD']
```

If you don't have any condition to satisfy, simply leave the `if` part away. For example, to invert all the currency pairs so that the first currency comes second and vice versa, you would do:

```
In [104]: [f"{pair[3:]}{pair[:3]}" for pair in currency_pairs]
Out[104]: ['JPYUSD', 'GBPUSD', 'CHFUSD', 'CADUSD', 'USDAUD',
'USDNZD']
```

With dictionaries, there are dictionary comprehensions:

```
In [105]: exchange_rates = {"EURUSD": 1.1152, "GBPUSD": 1.2454,
"AUDUSD": 0.6161}
          {k: v * 100 for (k, v) in exchange_rates.items()}
Out[105]: {'EURUSD': 111.52, 'GBPUSD': 124.54, 'AUDUSD': 61.61}
```

And with sets, there are set comprehensions:

```
In [106]: {s + "USD" for s in ["EUR", "GBP", "EUR", "HKD",
"HKD"]}
Out[106]: {'EURUSD', 'GBPUSD', 'HKDUSD'}
```

At this point, you are able to write simple scripts, as you know most of the basic building blocks of Python. In the next section, you will learn how to organize your code to keep it maintainable when your projects start to get bigger.

Code Organization

In this section, we'll bring our code into a maintainable strucutre: I'll start by introducing functions with all the details that you will commonly need before I'll show you how to split your code into different Python modules. The knowledge about modules will then allow us to look into the `datetime` module that is part of the standard library.

Functions

Even if you will use Python for simple scripts only, you are still going to write functions regularly: they allow you to reuse the same lines of code from anywhere in your program. We'll start this section by defining a function before we see how to call it!

Defining functions

To write your own function in Python, you have to use the keyword `def`, which stands for function *definition*. Unlike VBA, Python doesn't differentiate between a function and a Sub procedure. In Python, the equivalent of a Sub procedure is simply a function without a return statement. Functions in Python follow the syntax for code blocks, i.e., you end the first line with a colon and indent the body of the function:

```
def function_name(required_argument,
optional_argument=default_value, ...):
    return result
```

Required arguments

Required arguments do not have a default value. Multiple arguments are separated by commas.

Optional arguments

You make an argument optional by supplying a default value. `None` is often used to make an argument optional if there is no meaningful default.

Return value

> The `return` statement defines the result that the function returns. If you leave it away, the function returns `None`.

To be able to play around with a function, let's define one that is able to convert the temperature from Fahrenheit or Kelvin to degrees Celsius:

```
In [107]: def convert_to_celsius(degrees, source="fahrenheit"):
              if source.lower() == "fahrenheit":
                  return (degrees - 32) * (5 / 9)
              elif source.lower() == "kelvin":
                  return degrees - 273.15
              else:
                  # Exceptions will be introduced in Chapter 11
                  raise ValueError(f"Don't know how to convert
from {source}")
```

I am using the string method `lower`, which transforms the provided strings to lowercase. This allows us to accept the `source` string with any capitalization while the comparison will still work. With the `convert_to_celsius` function defined, let's see how we can call it!

Calling functions

As briefly mentioned at the beginning of this chapter, you call a function by adding parentheses to the function name, enclosing the function arguments:

```
result = function_name(positional_arg, arg_name=value, ...)
```

Positional arguments

> If you provide a value as a positional argument (`positional_arg`), the values are matched to the arguments according to their position in the function definition.

Keyword arguments

> By providing the argument in the form `arg_name=value`, you're providing a keyword argument. This has the advantage that you can provide the arguments in any order. It is also more explicit to the reader and can make it easier to understand. For example, if the function is defined as `f(a, b)`, you could call the function like this: `f(b=1, a=2)`. This concept also exists in VBA under the name *named arguments* with the following syntax: `f(b:=1, a:=1)`.

Let's play around with the `convert_to_celsius` function to see how this works in practice:

```
In [108]: convert_to_celsius(100, "fahrenheit")  # Positional
arguments
Out[108]: 37.77777777777778
In [109]: convert_to_celsius(100)  # Will use the default source
("fahrenheit")
Out[109]: 37.77777777777778
In [110]: convert_to_celsius(source="kelvin", degrees=0)  #
Keyword arguments
Out[110]: -273.15
```

Now that you know how to define and call functions, let's see how to organize them with the help of modules.

Modules and the import Statement

When you work on bigger projects, you will have to split your code into different files to make it easier to maintain. As we have already seen in the previous chapter, Python files have the extension *.py* and you usually refer to your main file as a *script*. If you want your main script to access functionality from other files, you need to *import* that functionality first. In this context, Python source files are called *modules*. To get a better feeling for how this works and what the different import options are, open the file *temperature.py* (Example 3-1) in VS Code. You'll find it in the root directory of the companion repository. If you need a refresher on how to open files in VS Code, have another look at Chapter 2.

Example 3-1. temperature.py

```
TEMPERATURE_SCALES = ("fahrenheit", "kelvin", "celsius")


def convert_to_celsius(degrees, source="fahrenheit"):
    if source.lower() == "fahrenheit":
        return (degrees - 32) * (5 / 9)
    elif source.lower() == "kelvin":
        return degrees - 273.15
    else:
        # Exceptions will be introduced in Chapter 11
        raise ValueError(f"Don't know how to convert from
{source}")


print("This is the temperature module.")
```

To be able to import the `temperature` module, *temperature.py* needs to be in the same directory as the Jupyter notebook—this is already the case in the companion repository. To import, you only use the name of the module, without the *.py* ending. After running the `import` statement, you will have access to all the objects in that Python module via the dot notation. For example, use `temperature.convert_to_celsius()` to perform your conversion:

```
In [111]: import temperature
This is the temperature module.
In [112]: temperature.TEMPERATURE_SCALES
Out[112]: ('fahrenheit', 'kelvin', 'celsius')
In [113]: temperature.convert_to_celsius(120, "fahrenheit")
Out[113]: 48.88888888888889
```

Note that I used uppercase letters for `TEMPERATURE_SCALES` to express that it is a constant—I will say more about that toward the end of this chapter. When you execute the cell with `import temperature`, Python will run the *temperature.py* file from top to bottom. You can easily see this happening since importing the module will fire the print function at the bottom of *temperature.py*.

MODULES ARE ONLY IMPORTED ONCE

If you run the `import temperature` cell again, you will notice that it does not print anything anymore. This is because Python modules are only imported once per session. If you change code in a module that you import, you need to restart your Python interpreter to pick up all the changes. To restart a Jupyter notebook, click on the Restart button at the top of a notebook in VS Code, or go to Kernel > Restart Kernel in classic Jupyter notebooks.

In reality, you usually don't print anything in modules. This was only to show you the effect of importing a module more than once. Most commonly, you put functions and classes in your modules (for more on classes, see Appendix C). If you don't want to type `temperature` every time you use an object from the temperature module, change the `import` statement like this:

```
In [114]: import temperature as tp
In [115]: tp.TEMPERATURE_SCALES
Out[115]: ('fahrenheit', 'kelvin', 'celsius')
```

Assigning an alias such as `tp` to your module can make it easier to use while it's still always clear where an object comes from. Many third-party packages suggest a specific convention when using an alias. For example, pandas is using the alias `pd`. There is one more option to import objects from another module:

```
In [116]: from temperature import TEMPERATURE_SCALES,
convert_to_celsius
In [117]: TEMPERATURE_SCALES
Out[117]: ('fahrenheit', 'kelvin', 'celsius')
```

THE __PYCACHE__ FOLDER

When you import the `temperature` module, you will see that Python creates a folder called *__pycache__* with files that have the *.pyc* extension. These are bytecode-compiled files that the Python interpreter creates when you import a module. For our purposes, we can simply ignore this folder, as it is a technical detail of how Python runs your code.

When using the `from ... import ...` syntax, you import specific objects only. By doing this, you are importing them directly into the *namespace* of your main script: that is, without looking at the `import` statements, you won't be able to tell whether the imported objects were defined in your current Python script or Jupyter notebook or if they come from another module. This could cause conflicts: if your main script has a function called `convert_to_celsius`, it would override the one that you are importing from the `temperature` module. If, however, you use one of the two previous methods, your local function and the one from the imported module could live next to each other as `convert_to_celsius` and `temperature.convert_to_celsius`.

DON'T NAME YOUR SCRIPTS LIKE EXISTING PACKAGES

A common source for errors is to name your Python file the same as an existing Python package or module. If you create a file to play around with pandas, don't call that file *pandas.py*, as this can cause conflicts.

Now that you know how the import mechanism works, let's use it to import the `datetime` module! This will also allow you to learn a few more things about objects and classes.

The datetime Class

Working with date and time is a common operation in Excel, but it comes with limitations: for example, date-formatted cells don't support smaller units than milliseconds and time zones are not supported at all. In Excel, date and time are stored as a simple float called the *date serial number*. The Excel cell is then formatted to display it as date and/or time. For example, January 1, 1900 has the date serial number of 1, which means that this is also the earliest date that you can work with in Excel. Time gets translated into the decimal part of the float, e.g., `01/01/1900 10:10:00` is represented by `1.4236111111`.

In Python, to work with date and time, you import the `datetime` module, which is part of the standard library. The `datetime` module contains a class with the same name that allows us to create `datetime` objects. Since having the same name for the module and the class can be confusing, I will use the following import convention throughout this book: `import datetime as dt`. This makes it easy to differentiate between the module (`dt`) and the class (`datetime`).

Up to this point, we were mostly using *literals* to create objects like lists or dictionaries. Literals refer to the syntax that Python recognizes as a specific object type—in the case of a list, this would be something like `[1, 2, 3]`. However, the majority of objects have to be created by calling their class: this process is called *instantiation*, and objects are therefore also called *class instances*. Calling a class works the same way as calling a function, i.e., you add parentheses to the class name and provide the arguments as we did with functions. To instantiate a `datetime` object, you need to call the class like this:

```
import datetime as dt
dt.datetime(year, month, day, hour, minute, second, microsecond,
timezone)
```

Let's go through a couple of examples to see how you work with `datetime` objects in Python. For the purpose of this introduction, we'll ignore time zones and work with time-zone-naive `datetime` objects. If

you want to know more about time zones though, have a look at Appendix C for a short introduction.

```
In [118]: # Import the datetime module as "dt"
          import datetime as dt
In [119]: # Instantiate a datetime object called "timestamp"
          timestamp = dt.datetime(2020, 1, 31, 14, 30)
          timestamp
Out[119]: datetime.datetime(2020, 1, 31, 14, 30)
In [120]: # Datetime objects have attributes to access specific
components.
          # Here, we are extracting the day:
          timestamp.day
Out[120]: 31
In [121]: # The difference of two datetime objects returns a
timedelta object
          timestamp - dt.datetime(2020, 1, 14, 12, 0)
Out[121]: datetime.timedelta(days=17, seconds=9000)
In [122]: # Add or subtract timedelta objects to shift datetime
values
          timestamp + dt.timedelta(days=1, hours=4, minutes=11)
Out[122]: datetime.datetime(2020, 2, 1, 18, 41)
```

To *format* `datetime` objects into strings, you usually use f-strings. Alternatively, `datetime` objects have a `strftime` method that can do the same. I will show both options in the code examples below. To *parse* a string and convert it into a `datetime` object, use the `strptime` function (you can find an overview of the accepted format codes in the datetime docs):

```
In [123]: # Format a datetime object via f-string.
          # Note that the format specifier starts with a colon.
          f"{timestamp:%d/%m/%Y %H:%M}"
Out[123]: '31/01/2020 14:30'
In [124]: # This is equivalent to the f-string
          timestamp.strftime("%d/%m/%Y %H:%M")
Out[124]: '31/01/2020 14:30'
In [125]: # Parse a string into a datetime object
          dt.datetime.strptime("12.1.2020", "%d.%m.%Y")
Out[125]: datetime.datetime(2020, 1, 12, 0, 0)
```

After this short introduction to the `datetime` module, let's move on to the last topic of this chapter, which is about formatting your code properly and discovering bugs without even running your code.

Code Formatting, Linting, and Type Hinting

In this section, you'll learn how to format your code according to the official Python style guide. To do the formatting automatically, I will show you the Ruff package, which is both a code formatter and linter. While the formatter edits your code to e.g., always use the same amout of empty lines after a function, a linter checks your code for style and programming errors. To conclude, I'll introduce you to type hints: they help editors like VS Code provide better code autocompletion and allow type-checkers to catch type-related errors. To get started, let's find out what PEP 8 means!

PEP 8: Style Guide for Python Code

While going through the code samples in this chapter, you may have noticed that I was sometimes using variable names with underscores or in all caps. To explain my formatting choices, let me introduce you to Python's official style guide. Python uses so-called Python Enhancement Proposals (PEP) to discuss the introduction of new language features. One of these, the Style Guide for Python Code, is often referred to by its number: PEP 8. PEP 8 is a set of style recommendations for the Python community; if everybody who works on the same code adheres to the same style guide, the code becomes much more readable. Example 3-2 shows a short Python file that introduces the most important conventions.

Example 3-2. pep8_sample.py

```
"""This script shows a few PEP 8 rules."""  ❶

import datetime as dt  ❷


TEMPERATURE_SCALES = (  ❸
    "fahrenheit",
```

```
    "kelvin",
    "celsius",
)
❹

class TemperatureConverter: ❺
    pass  # Doesn't do anything at the moment ❻


def convert_to_celsius(degrees, source="fahrenheit"): ❼
    """This function converts degrees Fahrenheit or Kelvin
    into degrees Celsius. ❽
    """
    if source.lower() == "fahrenheit": ❾
        return (degrees - 32) * (5 / 9) ❿
    elif source.lower() == "kelvin":
        return degrees - 273.15
    else:
        # Exceptions will be introduced in Chapter 11
        raise ValueError(f"Don't know how to convert from
{source}")


celsius = convert_to_celsius(44, source="fahrenheit") ⓫
non_celsius_scales = TEMPERATURE_SCALES[:-1] ⓬

print("Current time: " + dt.datetime.now().isoformat())
print(f"The temperature in Celsius is: {celsius}")
```

❶ Explain what the script or module does with a *docstring* at the top. A docstring is a special type of string, enclosed with triple quotes. Apart from serving as a string for documenting your code, a docstring also makes it easy to write strings over multiple lines and is useful if your text contains a lot of double-quotes or single-quotes, as you won't need to escape them. They are also useful to write multiline SQL queries, as we will see in Chapter 11.

❷ All imports are at the top of the file, one per line. List the imports of the standard library first, then those of third-party packages, and finally those from your own modules. This sample only makes use of the standard library.

❸ Use capital letters with underscores for constants. Use a maximum line length of 79 characters. If possible, take advantage of parentheses,

square brackets, or curly braces for implicit line breaks.

4. Separate classes and functions with two empty lines from the rest of the code.
5. Despite the fact that many classes from the standard library, like `datetime`, are lowercase, your own classes should use `CapitalizedWords` (also called "CamelCase"). For more on classes, see Appendix C.
6. Inline comments should be separated by at least two spaces from the code. Code blocks should be indented by four spaces.
7. Functions and function arguments should use lowercase names with underscores if they improve readability. Don't use spaces between the argument name and its default value unless you are using type hints (I will introduce type hints at the end of this section).
8. A function's docstring should also list and explain the function arguments and its return value. I haven't done this here to keep the sample short.
9. Don't use spaces around the colon.
10. Use spaces around mathematical operators. If operators with different priorities are used, you may consider adding spaces around those with the lowest priority only.
11. Use lowercase names for variables. Make use of underscores if they improve readability. When assigning a variable name, use spaces around the equal sign. However, when calling a function, don't use spaces around the equal sign used with keyword arguments.
12. With indexing and slicing, don't use spaces around the square brackets.

This is a simplified summary of PEP 8, so it's a good idea to have a look at the original PEP 8. PEP 8 clearly states that it is a recommendation and that your own style guides will take precedence. After all, consistency is the most important factor.

In practice, I didn't format Example 3-2 by hand. Instead, I used a package called Ruff to do the work for me—let me show you how it works!

Ruff

In order to save you from manually applying the PEP 8 rules, let me introduce you to Ruff, a Python package that is both a code formatter and linter. Let's start with the formatter before looking at the linter!

Formatter

The Ruff formatter is an opinionated formatter, meaning it offers only a few options you can change. This has the nice effect that the code looks consistent across all projects that use it. The Ruff formatter largely follows PEP 8 with a few exceptions—for instance, it uses a default line length of 88 characters while PEP 8 uses 79. Apart from formatting the actual code, Ruff also sorts the import statements at the top of your file. Since Ruff is a drop-in replacement for Black (the package that made automatic code formatting popular in Python), you can also say that Ruff produces Black-style formatting.

To try out Ruff, open a Terminal and run `uv run ruff format` in the directory of your companion repository:

Windows (Terminal)

```
$ cd %USERPROFILE%\Documents\python-for-excel-2e
$ uv run ruff format pep8_sample.py
```

macOS (Terminal)

```
$ cd ~/Documents/python-for-excel-2e
$ uv run ruff format pep8_sample.py
```

It will show `1 file left unchanged`, as the file is already properly formatted. However, if you introduce a change to *pep8_sample.py*, for example, by inserting a few empty lines, they will be removed when you

run the command again. After this quick intorduction to the formatter, let's continue with the linter!

Linter

A linter checks your source code for programming and style errors without running your code. To run Ruff's linter, use `ruff check` (as usual, you can ignore the `cd` command if you are already in the directory of the companion repository):

Windows (Terminal)

```
$ cd %USERPROFILE%\Documents\python-for-excel-2e
$ uv run ruff check pep8_sample.py
```

macOS (Terminal)

```
$ cd ~/Documents/python-for-excel-2e
$ uv run ruff check pep8_sample.py
```

It will show `All checks passed!` as the file doesn't have any errors. However, if you, for example, insert `import os` at the top of *pep8_sample.py* and run the command again, Ruff will return the following error message: `os imported but unused`.

RUFF VS CODE EXTENSION

If you are working with VS Code, instead of running the Ruff commands manually, it's arguably more convenient to install the Ruff extension. Once installed, linting issues are marked directly in the editor with squiggly lines and you can format your code via the command palette: fire it up using the keyboard shortcut Ctrl+Shift+P (Windows) or Command-Shift-P (macOS) and search for "Ruff: Format Document".

To find errors that a linter can't catch, e.g., supplying the wrong data type as a function argument, you can use type hints together with a type checker. Another benefit of type hints is that they enable editors such as VS Code to offer much better code completion—reason enough to look at them in a little more detail!

Type Hints

In VBA, you often see code that prefixes each variable with an abbreviation for the data type, like `strEmployeeName` or `wbWorkbookName`. While nobody will stop you from doing this in Python, it isn't commonly done. You also won't find an equivalent to VBA's `Option Explicit` or `Dim` statement to declare the type of a variable. Instead, Python offers *type hints*. Type hints are also called *type annotations* and allow you to declare the data type of a variable. They are optional and have no effect on how the code is run by the Python interpreter. Third-party packages, however, can make use of these type hints during runtime. For example, in Chapter 12, we will see how xlwings makes use of type hints to convert Excel values into specific Python objects. The main ideas of type hints are:

- Improve autocompletion in editors like VS Code
- Let type checkers find type-related errors (like linters, type checkers don't need to run your code)

To understand how type annotations work in Python, here is the function signature of our familiar temperature converter with type hints:

```
def convert_to_celsius(degrees: float, source: str =
"fahrenheit") -> float:
    ...
```

As you can see, type hints are added by writing a colon and the expected type after each parameter, and the return type is specified after the closing parenthesis using `->`. Go ahead and edit the file *pep8_sample.py*: add type

hints and an additional line of code to print whether `degrees` is actually an integer, so that the function now starts as follows:

```
def convert_to_celsius(degrees: float, source: str =
"fahrenheit") -> float:
    """This function converts degrees Fahrenheit or Kelvin
    into degrees Celsius.
    """
    print(f"degrees is an int: {degrees.is_integer()}")
    ...
```

With type hints, VS Code shows a list of attributes, including `is_integer`, when you type the dot after `degrees`, see Figure 3-2. If you remove the `float` type annotation from the `degrees` parameter again, you won't get this code autocompletion anymore.

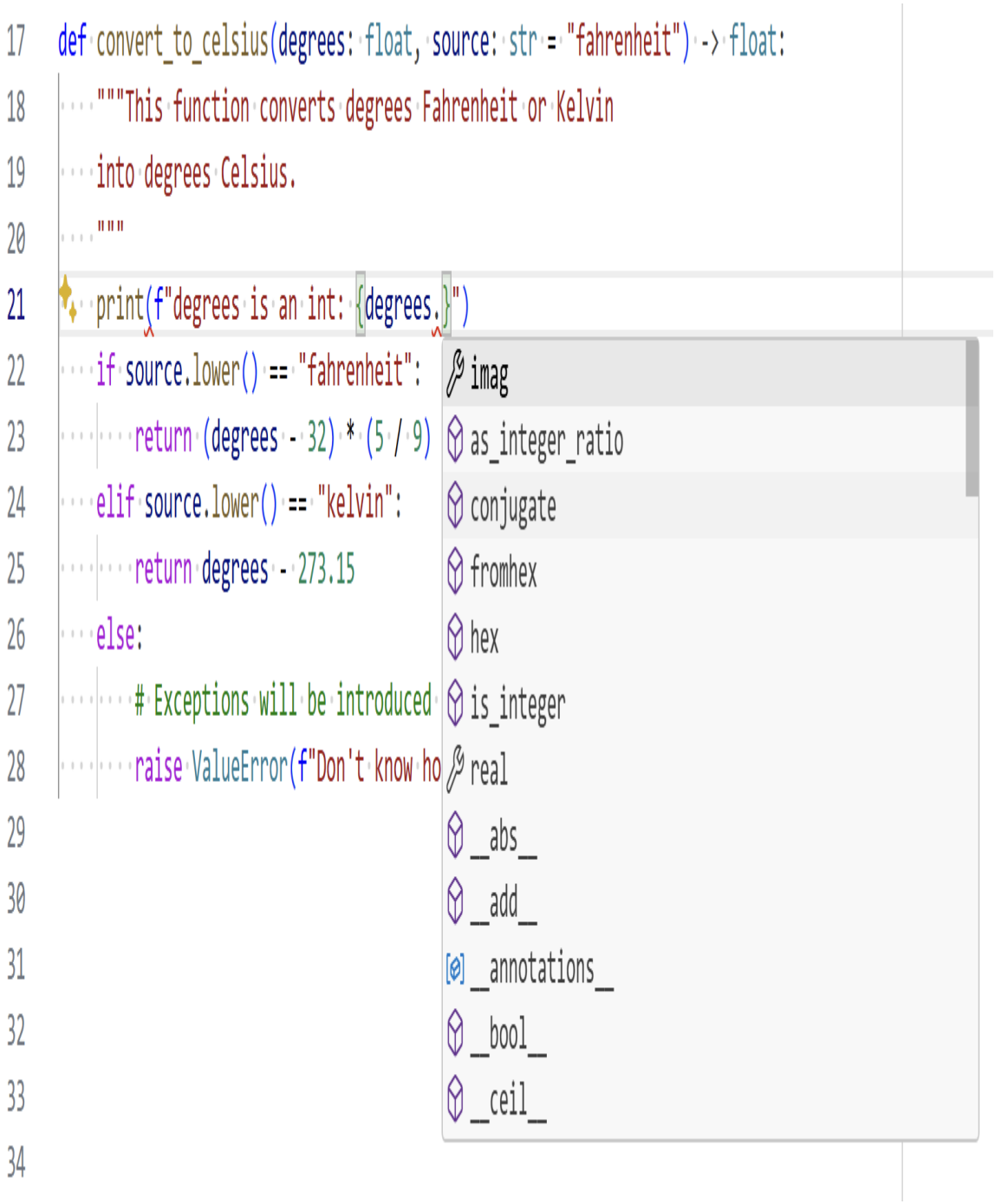

Figure 3-2. Code autocompletion after adding type hints

There are various type checkers available for Python code. For example, Pylance, a VS Code extension by Microsoft, has one included. Pylance got automatically installed during the installation of the Python extension in Chapter 2. However, by default, the Pylance type checker is off, so if you want to enable it, open the VS Code settings via Ctrl+, (Windows) or

Command+, (macOS), then search for `@id:python.analysis.typeCheckingMode`: this will pull up the respective page on the settings with instructions on how to enable the type checker. Another type checker is Ty, which is from the same team as Ruff. To try it out, open a Terminal and, if you are not already in the root directory of your companion repository, change into it first:

Windows (Terminal)

```
$ cd %USERPROFILE%\Documents\python-for-excel-2e
$ uv run ty check pep8_sample.py
```

macOS (Terminal)

```
$ cd ~/Documents/python-for-excel-2e
$ uv run ty check pep8_sample.py
```

You should get `All checks passed!`. However, if you change line 30 in *pep8_sample.py* from this:

```
celsius = convert_to_celsius(44, source="fahrenheit")
```

to this:

```
celsius = convert_to_celsius("forty-four", source="fahrenheit")
```

and then run the command again, Ty will complain with the following error: `Argument to function convert_to_celsius is incorrect`.

Despite the benefits that type hints bring, some Python programmers argue that they make the code harder to read. In this book, I will use type hints occasionally, where they make sense.

Conclusion

This chapter was a packed introduction to Python. We met the most important building blocks of the language, including data structures, functions, and modules. We also touched on some of Python's particularities like meaningful white space and code formatting guidelines, better known as PEP 8. To continue with this book, you won't need to know all the details: as a beginner, just knowing about lists and dictionaries, indexing and slicing, as well as how to work with functions, modules, `for` loops, and `if` statements will get you far already.

Compared to VBA, I find Python more consistent and powerful but at the same time easier to learn. If you are a VBA die-hard fan and this chapter didn't convince you just yet, I am pretty sure the next part will: there, I will give you an introduction to array-based calculations before starting our data analysis journey with the pandas library. Let's get started with Part 2 by learning a few basics about NumPy!

Chapter 4. NumPy Foundations

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at mpotter@oreilly.com.

As you may recall from Chapter 1, NumPy is the core package for scientific computing in Python, providing support for array-based calculations and linear algebra. As NumPy is a key dependency of pandas, I am going to introduce its basics in this chapter: after explaining what a NumPy array is, we will look into vectorization and broadcasting, two important concepts that allow you to write concise mathematical code and that you will find again in pandas. After that, we're going to see why NumPy offers special functions called universal functions before we wrap this chapter up by learning how to get and set values of an array and by explaining the difference between a view and a copy of an array. Even if we will hardly use NumPy directly in this book, knowing its basics will make it easier to learn pandas in the next chapter.

Getting Started with NumPy

In this section, you'll learn about one- and two-dimensional NumPy arrays and what's behind the technical terms *vectorization*, *broadcasting*, and

universal function.

NumPy Array

To perform array-based calculations with nested lists, you have to write some sort of loop. For example, you can use a nested list comprehension to add a number to every element in `matrix`:

```
In [1]: matrix = [[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]]
In [2]: [[element + 1 for element in row] for row in matrix]
Out[2]: [[2, 3, 4], [5, 6, 7], [8, 9, 10]]
```

This isn't very readable and more importantly, looping through each element is slow with big arrays. Depending on your use case and the size of the arrays, working with NumPy arrays instead of Python lists can make your calculations from a couple of times to around a hundred times faster. NumPy achieves this performance by using code written in C and Fortran—these are compiled programming languages that are much faster than Python. A NumPy array is an N-dimensional array for *homogenous data*. Homogenous means that all elements in the array need to have the same data type. Most commonly, you are dealing with one- and two-dimensional arrays of floats as schematically displayed in Figure 4-1.

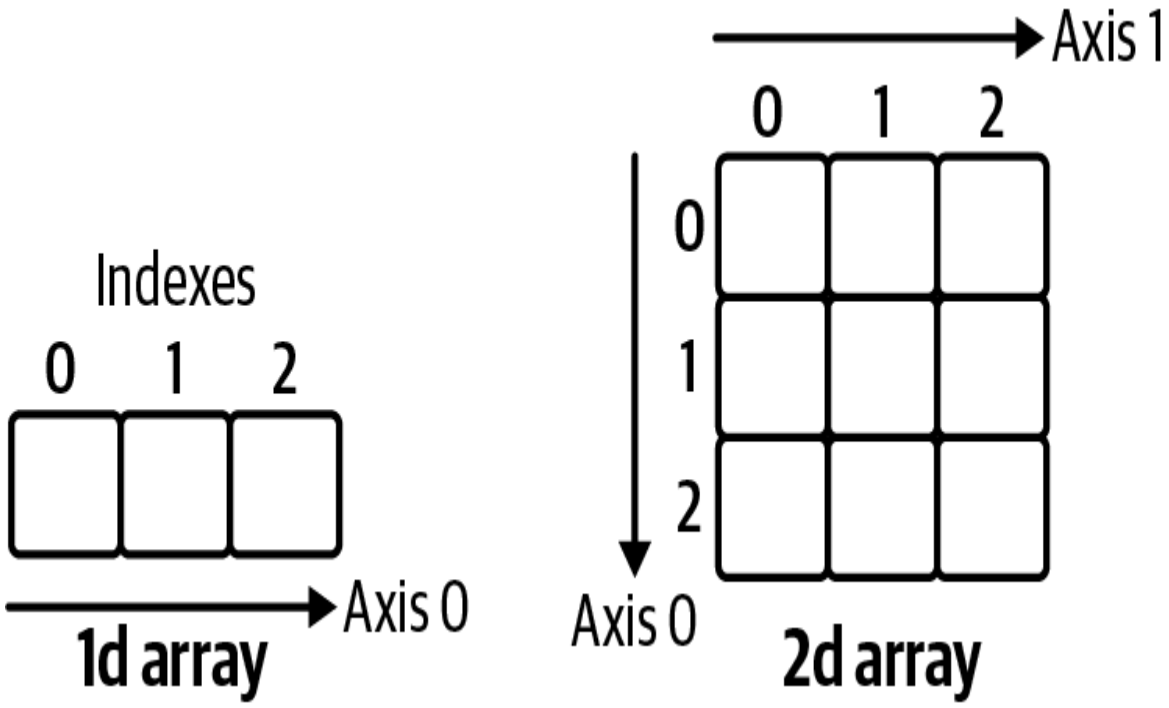

Figure 4-1. A one-dimensional and two-dimensional NumPy array

Let's create a one- and two-dimensional array to work with throughout this chapter:

```
In [3]: # Import NumPy with the "np" alias
        import numpy as np
In [4]: # Constructing an array with a simple list results in a
1d array
        array1 = np.array([10, 100, 1000.0])
        array1
Out[4]: array([  10.,  100., 1000.])
In [5]: # Constructing an array with a nested list results in a
2d array
        array2 = np.array([[1.0, 2.0, 3.0],
                           [4.0, 5.0, 6.0]])
        array2
Out[5]: array([[1., 2., 3.],
               [4., 5., 6.]])
```

ONE-DIMENSIONAL ARRAYS

A one-dimensional array has only one axis, so it does not have an explicit column or row orientation. While this behaves like arrays in VBA, it's different in MATLAB, a programming language used for similar use cases as NumPy. In MATLAB, one-dimensional arrays always have a column or row orientation.

Even if `array1` has two integers (10 and 100) and only one float (1000), the homogeneity of NumPy arrays forces the data type of the array to be `float64`, which is capable of accommodating all elements. To learn about an array's data type, access its `dtype` attribute:

```
In [6]: array1.dtype
Out[6]: dtype('float64')
```

Did you notice that `dtype` displays `float64` instead of the plain `float` type you saw in the last chapter? This is because NumPy uses its own numerical data types (`float64` stands for 64-bit floating-point number). This usually isn't an issue though, as in most cases you can use a NumPy float anywhere you would use a native Python float. If you ever need to explicitly convert from a NumPy data type to a Python data type, simply use the corresponding constructor (I will say more about accessing an element from an array shortly):

```
In [7]: array1[0]
Out[7]: np.float64(10.0)
In [8]: # Convert from a NumPy float64 to a native Python float
        float(array1[0])
Out[8]: 10.0
```

For a full list of NumPy's data types, see the NumPy docs. With NumPy arrays, you can write simple code to perform array-based calculations, as we will see next.

Vectorization and Broadcasting

If you build the sum of two NumPy arrays, NumPy will perform an element-wise operation so you don't have to loop through the elements yourself. This is called *vectorization*. It allows you to write concise code, practically representing the mathematical notation:

```
In [9]: array2 + array2
Out[9]: array([[ 2.,  4.,  6.],
               [ 8., 10., 12.]])
```

If you use two arrays with different shapes in an arithmetic operation, NumPy extends—if possible—the smaller array automatically across the larger array so that their shapes become compatible. This is called *broadcasting*. The first example illustrates how this works by summing up an array with a scalar, the second example multiplies a one-dimensional array with a two-dimensional array:

```
In [10]: array2 + 1
Out[10]: array([[2., 3., 4.],
                [5., 6., 7.]])
In [11]: array2 * array1
Out[11]: array([[  10.,  200., 3000.],
                [  40.,  500., 6000.]])
```

SCALAR

Scalar refers to a single-element data type like a float or an string. This is to differentiate them from data structures with multiple elements like lists, dictionaries, or NumPy arrays.

To perform matrix multiplications or dot products, use the `@` operator. This sample shows a matrix multiplication of `array2` with its transpose:

```
In [12]: array2 @ array2.T  # array2.T is a shorthand for
array2.transpose()
Out[12]: array([[14., 32.],
                [32., 77.]])
```

Don’t be intimidated by the terminology I’ve introduced in this section such as scalar, vectorization, or broadcasting! All this works the same in Excel if you use named ranges together with dynamic arrays. Figure 4-2 shows the previous examples together with the respective array-based Excel formula. As you can see, the formulas look the same with NumPy and Excel, with only the last one being slightly different. The screenshot is taken from *array_calculations.xlsx*, which you will find in the *xl* directory of the companion repository.

array1 (Named range)

10	100	1000

array2 (Named range)

1	2	3
4	5	6

- NumPy notation: array2 + 1
- Excel formula: =array2 + 1

2	3	4
5	6	7

- NumPy notation: array2 * array2
- Excel formula: =array2 * array2

1	4	9
16	25	36

- NumPy notation: array2 * array1
- Excel formula: =array2 * array1

10	200	3000
40	500	6000

- NumPy notation: array2 @ array2.T
- Excel formula: =MMULT(array2, TRANSPOSE(array2))

14	32
32	77

Figure 4-2. Array-based calculations in Excel

You know now that arrays perform arithmetic operations element-wise, but how can you apply a function on every element in an array? This is what universal functions are here for.

Universal Functions (ufunc)

Universal functions (ufunc) work on every element in a NumPy array. For example, if you use Python's standard square root function from the `math` module on a NumPy array, you will get an error:

```
In [13]: import math
In [14]: math.sqrt(array2)  # This will raise en Error
---------------------------------------------------------------
----------
TypeError                                 Traceback (most recent
call last)
Cell In[14], line 1
----> 1 math.sqrt(array2)  # This will raise en Error

TypeError: only length-1 arrays can be converted to Python
scalars
```

You could, of course, write a nested loop to compute the square root of every element and build a new NumPy array from the result:

```
In [15]: # We're using a nested list comprehension to do the
looping
         np.array([[math.sqrt(element) for element in row] for
row in array2])
Out[15]: array([[1.        , 1.41421356, 1.73205081],
                [2.        , 2.23606798, 2.44948974]])
```

This makes sense in cases where NumPy doesn't offer a ufunc and the array is small enough. However, if NumPy has a ufunc, you should use it, as it will be much faster with big arrays—apart from being easier to type and read:

```
In [16]: np.sqrt(array2)
```

```
Out[16]: array([[1.        , 1.41421356, 1.73205081],
                [2.        , 2.23606798, 2.44948974]])
```

Some of NumPy's ufuncs, like `sum`, are additionally available as array methods: if you want the sum of each column in an two-dimensional array, do the following:

```
In [17]: array2.sum(axis=0)  # Returns a 1d array
Out[17]: array([5., 7., 9.])
```

The argument `axis=0` refers to the axis along the rows while `axis=1` refers to the axis along the columns, as depicted in Figure 4-1. Leaving the `axis` argument away sums up the whole array:

```
In [18]: array2.sum()
Out[18]: np.float64(21.0)
```

You will meet more NumPy ufuncs in later chapters, as they can also be used with pandas DataFrames.

So far, we've always worked with the entire array. The next section shows you how to manipulate parts of an array and introduces a few helpful array constructors.

Creating and Manipulating Arrays

I'll start this section by showing you how to get and set specific elements of an array before introducing a few useful array constructors, including one to create pseudorandom numbers. To wrap it up, I'll explain the difference between a view and a copy of an array.

Getting and Setting Array Elements

In the last chapter, I showed you how to access specific elements of a list via indexing and slicing. As an example, to get the first element of the first row from a nested list called `matrix`, you write `matrix[0][0]`. NumPy

arrays, however, expect you to provide the index and slice arguments for both dimensions in a single pair of square brackets:

```
numpy_array[row_selection, column_selection]
```

For one-dimensional arrays, this simplifies to `numpy_array[selection]`. When you select a single element, you will get back a scalar; otherwise, you will get back a one- or two-dimensional array. Remember that slice notation uses a start index (included) and an end index (excluded) with a colon in between, as in `start:end`. By leaving away the start and end index, you are left with a colon, which therefore stands for all rows or all columns in a two-dimensional array. I have visualized a few examples in Figure 4-3, but you may also want to give Figure 4-1 another look, as the indices and axes are labeled there. Remember, by slicing a column or row of a two-dimensional array, you end up with a one-dimensional array, not with a two-dimensional column or row vector!

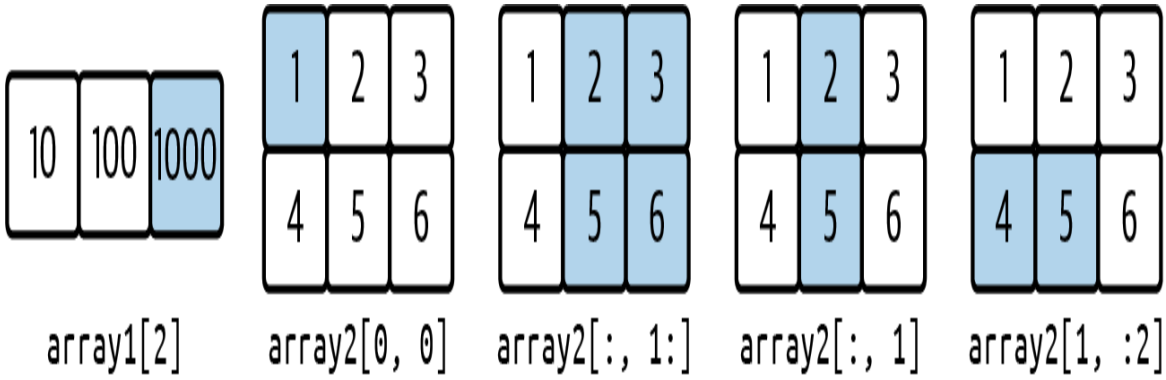

Figure 4-3. Selecting elements of a NumPy array

To play around with the examples shown in Figure 4-3, run the following code:

```
In [19]: array1[2]  # Returns a scalar
Out[19]: np.float64(1000.0)
In [20]: array2[0, 0]  # Returns a scalar
Out[20]: np.float64(1.0)
In [21]: array2[:, 1:]  # Returns a 2d array
Out[21]: array([[2., 3.],
                [5., 6.]])
```

```
In [22]: array2[:, 1]  # Returns a 1d array
Out[22]: array([2., 5.])
In [23]: array2[1, :2]  # Returns a 1d array
Out[23]: array([4., 5.])
```

So far, I have constructed the sample arrays by hand, i.e., by providing numbers in a list. In practice, though, NumPy offers several functions to create arrays more easily!

Useful Array Constructors

NumPy offers a few ways to construct arrays that will also be helpful to create pandas DataFrames, as we will see in Chapter 5. One way to easily create arrays is to use the `arange` function. This stands for *array range* and is similar to the built-in function `range` that we met in the previous chapter—with the difference that `arange` returns a NumPy array. Combining it with `reshape` allows us to quickly generate an array with the desired dimensions:

```
In [24]: np.arange(2 * 5).reshape(2, 5)  # 2 rows, 5 columns
Out[24]: array([[0, 1, 2, 3, 4],
                [5, 6, 7, 8, 9]])
```

Another common need, for example for Monte Carlo simulations, is to generate arrays of normally distributed pseudorandom numbers. NumPy makes this easy:

```
In [25]: rng = np.random.default_rng()  # Random number generator
         rng.standard_normal((2, 3))  # 2 rows, 3 columns
Out[25]: array([[-0.59754275,  1.00201726, -1.53665617],
                [ 0.04746927,  1.34369702,  1.2722003 ]])
```

Other helpful constructors are `np.ones` and `np.zeros` to create arrays with ones and zeros, respectively, and `np.eye` to create an identity matrix. We'll come across some of these constructors again in the next chapter, but for now, let's continue with the difference between a view and a copy of a NumPy array.

View vs. Copy

NumPy arrays return *views* when you slice them. This means that you are working with a subset of the original array without copying the data. Setting a value on a view will therefore also change the original array:

```
In [26]: # This is the original array
         array2
Out[26]: array([[1., 2., 3.],
                [4., 5., 6.]])
In [27]: # Slicing means that subset is a view
         subset = array2[:, :2]
         subset
Out[27]: array([[1., 2.],
                [4., 5.]])
In [28]: # Changing a value in subset...
         subset[0, 0] = 1000
In [29]: # ...changes the subset...
         subset
Out[29]: array([[1000.,    2.],
                [   4.,    5.]])
In [30]: # ...but also the original array
         array2
Out[30]: array([[1000.,    2.,    3.],
                [   4.,    5.,    6.]])
```

If that's not what you want, create your subset as a copy:

```
subset = array2[:, :2].copy()
```

Working on a copy will leave the original array unchanged.

Conclusion

In this chapter, I showed you how to work with NumPy arrays and explained the concepts of vectorization and broadcasting. Putting these technical terms aside, using arrays should feel quite intuitive given that they follow the mathematical notation very closely. While NumPy is an incredibly powerful library, there are two main issues when you intend to use it for data analysis:

Homogeneous data types

NumPy arrays require all elements to have the same data type. This, for example, means that you can't perform any arithmetic operations when your array contains a mix of text and numbers.

Missing labels

NumPy arrays don't provide labels for rows or columns, which makes it harder to keep track of what each column or row represents.

To solve these issues, pandas provides the data structures Series and DataFrame. The next chapter will teach you how to use them!