

PYTHON FOR ACCOUNTING

A Modern Guide to Using
Python Programming
in Accounting

by **HORATIO BOTA**
with **ADRIAN GOSA**

Python for Accounting

A Modern Guide to Using
Python Programming in Accounting

by Horatio Bota
with Adrian Gosa

Library of Congress Control Number: 2021901161

ISBN: 978-973-0-33892-8

Version identifier: 115431b

Disclaimer

Although the authors have made every effort to ensure that the information in this book was correct at publication time, the authors do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause.

The authors have endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of nomenclature. However, the authors cannot guarantee the accuracy of this information.

Copyright

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the authors, except in the case of brief quotations embedded in critical articles or reviews. This work is registered with the U.S. Copyright Office.

Cover image by Phil Sheldon. Copyright © 2021 Horatio Bota.

About the author

Horatio Bota is a freelance Data Scientist with over seven years of experience in data analytics and data science. He has previously worked at Microsoft Research, J.P. Morgan Chase, and several startups in the U.K. He holds a B.Sc. and a Ph.D. in Computing Science from the University of Glasgow.

About the content reviewer

Adrian Gosa is a Senior Accountant at *Nike Europe*, with over seven years of business and finance experience. He has previously worked at PricewaterhouseCoopers and Deloitte, covering a wide variety of industries. Adrian holds an M.A. in Business Economics and an M.Sc. in Quantitative Finance from the University of Glasgow.

About the technical reviewer

Alexandra Vtyurina is a Research Scientist at *Kira Systems* in Toronto, where she uses Python to analyze user behavior. She holds a B.Sc. and M.Sc. in Computer Science from the Southern Federal University, and is about to get her Ph.D. in Computer Science from the University of Waterloo.

Contact

Did you like the book? Did you find it helpful? We'd love to add your name to our list of testimonials on the website! Please email us at contact@pythonforaccounting.com.

If you'd like to report any mistakes, typos, or offer suggestions on how we can improve this book, please email us at errata@pythonforaccounting.com.

Get updates via Twitter

If you'd like to be notified of updates to the book, follow [@pfabook](#) on Twitter.

Contents

Contents	v
1 Introduction	1
What this book is about	2
Whom this book is for	2
Why read this book	3
A quick tour of Python’s data tools	7
How to use this book	10
Roadmap	11
2 Getting set up	13
Installing Python on your computer	13
Setting up your local workspace	15
Using JupyterLab	16
Using Anaconda Navigator	25
PART ONE PYTHON ABCs	27
3 A quick look at Python code	29
4 Variables and operators	32
Everything is an object	32
Point me to the values	34
The cycle of a variable’s life	36
Operators	37
5 Python’s built-in data types	41
Integers	41
Floating-point numbers	42
Booleans	43
The None type	44
Strings	45
Associated built-in functions	47
6 Python’s built-in collections	49
Lists	49
Overthinking: Tuples and sets	55

Dictionaries	57
Membership operators	60
Associated built-in functions	60
7 Control flow	62
If-else statements	62
Loops	63
List comprehensions	66
8 Functions	68
Defining functions	68
Parameters and arguments	68
Return values	72
9 Modules, packages, and libraries	74
Modules and packages	74
Libraries	76
10 How to find help	79
Jupyter notebook helpers	79
Finding help online	80
11 Overthinking: code style	82
Naming variables and functions	82
Breaking long lines of code	83
Spacing around operators	84
Code comments	85
PART TWO WORKING WITH TABLES	87
12 Pandas in a nutshell	90
Getting started	90
Reading data from spreadsheets	91
Preparing and transforming data	92
Visualizing data	96
Writing data to a spreadsheet	97
13 Tables, columns, and values	98
Series	98
DataFrame	99
Index and axes	100

Values and types	103
14 Reading and writing Excel files	107
Reading Excel files	107
Inspecting data	109
Writing Excel files	111
15 Slicing, filtering, and sorting tables	115
Selecting columns	115
Removing columns	119
Selecting rows and columns	120
Filtering data	124
Sorting data	130
16 Project: Organizing sales data by channel	133
17 Adding and modifying columns	140
Adding columns	140
Renaming columns	142
Replacing values	144
Assigning new values to a table slice	145
Filter and edit	147
18 Summarizing data	150
Counting unique values	150
Averages and numerical summaries	152
High-level table information	154
19 Cleaning data	156
Dealing with missing values	156
Dealing with duplicate rows	163
Converting column data types	165
20 Project: Reading and cleaning a QuickBooks general ledger	168
21 Working with text columns	178
Revisiting Python strings	178
String methods in pandas	179
String data types in pandas	188
Overthinking: Regular expressions	191

22 Working with date columns	195
Revisiting Python dates	195
Date columns	197
Pandas date methods	202
Pandas date arithmetic	207
Overthinking: Timezones	217
23 Applying custom functions	220
Applying functions to columns	220
Overthinking: Functions without a name	223
Applying functions to rows	224
Overthinking: Other function parameters	226
24 Project: Mining product reviews	230
25 Concatenating tables	239
Row-wise concatenation	239
Column-wise concatenation	244
Appending rows to a DataFrame	245
26 Joining tables	247
How joins work	247
Joins in pandas	252
Inner, outer, left, and right joins	255
More joining options	259
27 Project: Filling missing product names in the sales data	262
28 Groups and pivot tables	267
How group operations work	267
Group operations in pandas	268
Stacking and unstacking	277
Pivot tables	279
29 Overthinking: Changing how DataFrames are displayed	284
Setting display options	284
Styling tables	287

30 Plotting with matplotlib	294
Elements of a matplotlib plot	294
Plotting basics	296
Adjusting plot details	302
Overthinking: Styles, colors, and fonts	311
31 Project: Making a waterfall plot from a cash flow statement	318
32 Other plotting libraries	330
Plotting with pandas	330
Plotting with seaborn	337
Using matplotlib, pandas and seaborn together	340
Overthinking: Interactive plots	344

33 Setting up your project	349
Files and folders	349
Documenting your analysis	351
34 Preparing data	355
Product and cost data	355
Sales data	358
35 Finding answers	366
Channel profits	367
Category profits	368
Channel and category profits	370
Product profits	371
Sharing results	375
Next steps	383

1

Introduction

If you work in accounting, you've probably heard the rumors: artificial intelligence and automation are set to "*reshape the accounting function*" over the next few years. When exactly the robots take over¹ is still uncertain, but how they'll do that is already clear: someone will program them. More likely than not, that someone is you.

You're best suited to automate the repetitive tasks in your job because you already know how: you have lots of domain knowledge and are used to working with computers and data.² You even have programming experience: most accounting work involves creating custom computer programs through point-and-click operations instead of code — these programs are called spreadsheets. You probably know what "*functions*" or other programming concepts are, and more importantly, you know how to *think* about data manipulation in powerful ways.

Unfortunately, the tools used in accounting today limit how expressive you can be when working with data. How many times have you found yourself digging through Excel's menus for a command that almost does what you want it to, but not entirely? How painless is it to work with large files in Excel? How simple is it to automate the boring parts of your workflow?

Unlike Excel, Python is a tool designed for expressing any data manipulation you can think of. On top of that, Python is fast, handles spreadsheets that Excel can't even open, and working with it always leaves a trace (i.e., code) that you can check and run whenever you need to. Even better, because you're already familiar with programming ideas and have the right mental models for working with data, all you need to start using Python is a guide on how to add it to your accounting toolkit.

This book is your guide. It starts with Python programming basics and goes all the way to making interactive data visualization using Python. On the way, it introduces many of the tools that have become a foundation for data science — all the while keeping its focus on accounting tasks and data (i.e., on using Python with spreadsheets, not instead of them).

The rest of this introduction explains how Python enables powerful data analysis practices that can strip away much of the boring parts of your work. Hopefully, by the end of this chapter, it will be clear why reading the rest of this book is worth your time.

1: The deadline keeps getting postponed since the 1950s.

2: And because you're reading this book, which means you want to use programming in your work.

What this book is about

This book is about computer programming more than it is about accounting practices: it teaches you how to use the Python programming language to work with tables. You'll use general ledgers, cash flow statements, and other accounting datasets throughout the book, but the goal is to teach you how to work with those datasets using Python code rather than introduce new accounting ideas on the way.

There's more than one kind of code: websites, smartphone apps, or the operating system on your computer are all built with code, but each is made with different programming languages and tools. Even when handling tables with Python, you can quickly glue spreadsheets together in a few lines of code, or you can code up an entire data analysis project. Both approaches use code, but larger projects employ more of Python's gears and tools.

As it happens, accounting tasks are a mix of quick data wrangling (e.g., moving rows between spreadsheets, cleaning text entries, reconciling values across files) and extensive data analysis projects (e.g., margin or costing analysis, inventory forecasts). You can easily use Python for both, but you need to use more of Python's machinery for larger projects. As we progress with the book, I'll introduce different parts of that machinery and go through several data wrangling projects using accounting datasets before attempting a Python-based sales analysis project. This way, you'll see how to use Python for the boring parts in your work first, and how to set up an entire data analysis project in Python towards the end of the book.

Whom this book is for

If you use a computer at work in any way, knowing how to write even a few lines of Python code will make you more productive (and you might even find coding fun). This book is for anyone who works with accounting data in Excel and wants to use Python to improve their workflow:

- ▶ Accounting, finance, or economics students
- ▶ Accountants working with anything from payroll to costing
- ▶ Business controllers
- ▶ Auditors
- ▶ Financial planners

If you are in any of these groups, you've likely experienced the frustration of using Excel with larger spreadsheets or of trying to get an Excel chart to look the way you want it to. Fortunately, Python can help with both of these — and more.

You don't need any Python or general coding knowledge to get started. However, you'll get the most out of this book if you know what a pivot table is and how to use at least some of Excel's functions (e.g., `SUM` or `LEN`). If you've been using Excel in your work for a while, you're good to go.

Some of you may have used R or Python³ before. In that case, the first part of the book will serve as a refresher for general programming concepts and guide you through setting up the tools you need to run Python data analysis code. If this is you, feel free to jump ahead and read the chapters you find most useful.

3: Both R and Python are programming languages used widely for data analysis and statistics.

Why read this book

Much has been written about coding as an essential skill, with everyone from Steve Jobs to Barack Obama encouraging people to learn how to code. While knowing how to code can be useful for anyone working with digital content, it's a crucial skill for accounting professionals because accounting is a type of programming that uses the wrong tools.

What's wrong with Excel for accounting

Python is essential in tech and science, but in accounting, Excel is king because most accountants are familiar with Excel, and it works well for certain tasks.⁴ However, there are genuine drawbacks to using Excel as your primary data analysis tool.

4: Data entry is one of those tasks.

The first drawback is that the work you do in Excel is not easily reproducible. With Excel, you don't have a record of all the steps you took in your analysis, so you can't rerun those steps if something goes wrong⁵ or if you get a new dataset that needs the same kind of analysis. Repeating the same actions over and over, every time the data changes or Excel crashes is not only time consuming and annoying, but exceedingly error-prone.

5: And Excel does have a fondness for crashing without saving your work, at the worst of times.

A related drawback is that data in spreadsheets just *is*: you don't always know how it was generated, where it was sourced from, or if anyone accidentally modified critical values in the email back-and-forth. Most analyses involve many cleaning or transformation steps, which can't be easily communicated over email chains, so often you can't check the assumptions that went into generating

a certain dataset. Because operations in Excel don't leave a trace, errors introduced in a spreadsheet get discovered too late, after reports have been published and decisions already taken.⁶

Lastly, Excel has hard limits on the size of data you can work with. An Excel spreadsheet can have at most 1 048 576 rows and 16 384 columns.⁷ Even though these limits seem reasonable, you don't need a massive business to gather data that go beyond these limits nowadays. Even if your data is within limits, how easy is it to open or work with a large dataset in Excel?⁸

All of the drawbacks mentioned above can be overcome by handling data using modern tools and practices (i.e., code, documentation, tests) that are open and explicit at every step (so you can easily find errors and fix them), scalable (that don't freeze your computer at over a million rows), and documented (so that anyone in your team can reproduce your analysis whenever they need to). Which is where Python comes in.

What Python is

Python is a programming language. A programming language is a set of rules⁹ for writing text such that your computer can understand what that text means. When you install Python on your computer, you install a program (just like you install Excel) that knows how to turn text (i.e., Python code) into instructions for your computer processor to run.

It's worth highlighting that the word "*Python*" refers to three distinct concepts:

- ▶ The *Python programming language* is the set of rules that defines how Python code can be written and interpreted by humans or computers (i.e., what symbols or keywords you can use when writing code, such as `for` or `if`, what they mean, etc.);
- ▶ The *Python interpreter* (or Python implementation) is the computer program that can read Python code¹⁰ and transform it into instructions that your computer's processor can understand and execute. Interpreters are often developed by the same community that decides what the Python language rules are. There are many Python interpreters available (just like there are many applications you can use to work with spreadsheets, not just Excel); the one we'll use is called IPython (short for *interactive Python*).¹¹ When you install Python on your computer, what you install is a Python interpreter and the Python standard library.
- ▶ The *Python standard library* is a collection of Python packages¹² that come with the Python interpreter. You can think of packages as extensions to the main application, which is the Python interpreter

6: "Growth in the Time of Debt" is a well-known academic paper written by two Harvard professors that made the case for austerity measures back in 2010. It is also one of the most famous cases of how consequential Excel data analysis errors can be.

7: There are more hard limits, not just on the number of rows or column. You can read more about them in the [Excel specifications and limits](#) article on the Microsoft Office support website.

8: You can find a recent example of how consequential Excel's limits are at: "[Excel: Why using Microsoft's tool caused Covid-19 results to be lost](#)".

9: Much like the English language is a set of rules of what different words mean and how you can put one after the other to form sentences.

10: In fact, Python code is just text that follows certain formatting rules. Python code files are plain text files.

11: IPython enables the style of interactive computing we'll be using throughout the book.

12: More details on what exactly these packages are in the following chapters.

(just like browser extensions or Excel add-ins). Most of the standard library packages are developed and maintained by the same community that develops the Python language. Python's standard library is commonly referenced as one of its main strengths; it provides packages that are useful for various tasks, such as creating graphical user interfaces, working with zip files, and many others.

Python isn't just any other programming language: it is *the* most popular programming language in the world right now.¹³ This popularity means that there's a lot of community support around the language: lots of Q&A content on the web related to Python, many tutorials, and plenty of books on how to use Python in different domains. One of these domains, where it has taken over, is data analysis: Python is now the most popular programming language for data science, machine learning, or scientific computing, making it an excellent tool for accounting.

Python is designed for code readability first: no unnecessary punctuation, no curly brackets, and English words instead of operators wherever possible (e.g., `and` instead of `&&`). Because of its emphasis on readability, it's easy to learn, easy to write, and easy on your fingers as well (for example, Python code is typically 3–5× shorter than Java¹⁴ code). Even if you haven't had any exposure to Python, you can already understand it, though you might not be comfortable expressing yourself with it yet:

```
def multiply_by_two(number):
    return number * 2

multiply_by_two(10)
```

20

A more familiar example might be the side-by-side comparison between two expressions of the same function, in VBA¹⁵ and Python, shown in listing 1.1 on the following page. You can see there how Python is designed to be readable and concise, in contrast to VBA, which has many confusing keywords and operators.

Python is free to use and runs on almost any computer you can think of (including your smartphone). The source code is entirely open, and the language was designed from the ground up around principles of openness (unlike, for example, Java, which is managed by Oracle). Python's creators describe it as "*powerful... and fast; plays well with others; runs everywhere; is friendly and easy to learn; is Open*", which covers most of the reasons behind its popularity.

13: In 2020, according to the [Popularity of Programming Language Index](#), which ranks programming languages based on how often people search for tutorials on Google.

14: Java is a programming language developed by the Oracle Corporation.

15: Visual Basic for Applications or VBA is the programming language integrated in Microsoft Office applications. As of 2020, it is "*the most dreaded*" programming language according to a [Stack Overflow developer survey](#).

```

1 Function Extract_Number_from_Text(Phrase
2     As String) As Double
3 Dim Length_of_String As Integer
4 Dim Current_Pos As Integer
5 Dim Temp As String
6 Length_of_String = Len(Phrase)
7 Temp = ""
8 For Current_Pos = 1 To Length_of_String
9 If (Mid(Phrase, Current_Pos, 1) = "-")
10 Then
11     Temp = Temp & Mid(Phrase, Current_Pos,
12         1)
13 End If
14 If (Mid(Phrase, Current_Pos, 1) = ".")
15 Then
16     Temp = Temp & Mid(Phrase, Current_Pos,
17         1)
18 End If
19 If IsNumeric(Mid(Phrase, Current_Pos, 1))
20     ) = True Then
21     Temp = Temp & Mid(Phrase, Current_Pos
22         , 1)
23 End If
24 Next Current_Pos
25 If Len(Temp) = 0 Then
26     Extract_Number_from_Text = 0
27 Else
28     Extract_Number_from_Text = CDbl(Temp)
29 End If
30 End Function

```

```

1 def extract_number_from_text(phrase):
2     return float(
3         "".join([
4             c for c in phrase
5                 if c.isdigit() or
6                     c in ["-", "."]
7             ]) or 0
8     )

```

Listing 1.1: A function that extracts a decimal number from some text given as input. The left version is written in VBA, the right version is in Python. Both examples are incomplete but illustrate the differences in readability between VBA and Python. VBA example is taken from www.automateexcel.com.

Why Python for accounting

For someone working in accounting right now, having to source data from various places (e.g., SAP or some other corporate database) and glue it together in Excel, learning Python should be extremely compelling. Just copy-pasting data from several large Excel files into a new spreadsheet is bound to generate a lot of frustration. And if you need to repeat this operation every few days, your only option is to do it all over again, manually. In Python, you can do this in a few lines of code, and once written, you can reuse the code repeatedly.

Using Python to glue data this way is a feature, not a side-effect. Python is designed for gluing things together (i.e., datasets, systems, software components), and this is one of the reasons it's suited for accounting. Through its libraries¹⁶ Python already works with many of the tools used in accounting today (e.g., QuickBooks, Tableau, Excel, etc.) and can be used to glue them together.

16: I'll describe these tools and libraries more in the following section.

Besides gluing things together, Python is also great at data analysis. Python's ecosystem of tools has exploded over the past decade, both in the number of data analysis libraries and their adoption in different domains (including finance). These tools have been optimized for handling large data and can boost your accounting workflow because they:

- ▶ Work well with the software you already use (e.g., SAP, Excel);
- ▶ Allow you to handle data in explicit and easy-to-check ways;
- ▶ Work with large datasets that Excel can't even open;
- ▶ Enable you to automate repetitive manual tasks.

Switching to Python for handling data addresses many of Excel's drawbacks I mentioned earlier. Unlike point-and-click operations, data analysis code is always explicit, stays reproducible, and scales with your data. If this is not enough to get you excited about the rest of the book, consider that Microsoft has begun to embrace the world of open-source software and, in particular, the world of Python.¹⁷ It's not unreasonable to imagine that in a few years, given its immense popularity, Python might replace VBA as the scripting language used in Excel.

In most analytics domains, accounting included, there's always some new tool to learn, which promises mythical productivity gains (e.g., Tableau, Looker, Alteryx, etc.) but ends up turning one kind of frustration into another. Python isn't one of these: it's a foundation on which you can build your own tools or glue those you're already using to make them work together.

17: Microsoft provides their very own introduction to Python course: docs.microsoft.com/en-us/learn/modules/intro-to-python. More to the point, the creator of Python started working for Microsoft in 2020.

A quick tour of Python's data tools

Whether in Excel or Python, handling data usually means reading, preparing, transforming, and visualizing tables (spreadsheets, CSV files, or any other kind of table). The easiest way to do that in Python is by using its data analysis libraries.

Python libraries are collections of code that allow you to run different operations on your data without writing all the code for those operations. They are almost always free to use and open-source (i.e., you can see the code they run, unlike, for example, Excel, which is closed-source).¹⁸ Many are stored on github.com, a public repository of open-source libraries and projects (for various programming languages, not just Python).

18: They are usually developed by communities of programmers, engineers, scientists or enthusiasts who contribute their time and expertise because it benefits everyone (themselves and the wider community). You can contribute too!

To use libraries in your Python code, you need to install them on your computer (just like installing any other software) and import them into your Python code.

The main Python data analysis libraries you'll be using throughout this book are:

- ▶ **pandas** is *the* data analysis library for Python. It is designed for manipulating tabular data and the main tool you will use to work with Excel spreadsheets in Python. The `pandas` library aims to “*become the most powerful and flexible data analysis tool available in any (programming) language*”; given its widespread adoption, it has (arguably) already achieved this aim. `pandas` provides out-of-the-box code for reading, cleaning, and transforming data from various sources and is extremely fast even on large datasets (i.e., anything over 1 000 000 rows). A large part of this book is about using `pandas` together with Python (and other Python libraries) for accounting tasks.
- ▶ **NumPy** is a Python library for working with matrices and mathematical operations related to matrices.¹⁹ There is a close link between `pandas` and `NumPy`: `pandas` uses the `NumPy` library extensively in its internal code. The `NumPy` library also fills in certain gaps in `pandas`'s functionality, which is why you'll often see them used together.
- ▶ **matplotlib** and **seaborn** are two of the most popular data visualization libraries used in Python-based data analysis. While `matplotlib` is very flexible and allows you to customize plots down to the smallest of details, `seaborn` is built on top of `matplotlib` to generate complex plots quickly, often in one line of code. Even though `matplotlib` and `seaborn` cover most plotting use-cases (and are far more versatile than Excel), many other Python data visualization libraries are available online for you to explore.

We'll cover how to install and use libraries in your Python code in the following chapters, but here is a quick example of how you can import the `pandas` Python library and read data from an Excel file using the `read_excel` function²⁰ (which is part of `pandas`):

```
import pandas as pd

pd.read_excel('Q1Sales.xlsx')
```

Some of the libraries mentioned above depend on one another in fairly technical ways.²¹ Getting more clarity on how they work together and which one does what exactly requires using them for a while. For now, to make some sense of these libraries, you can view them as black-boxes: collections of code that you feed data to (in the form of tables), and they give you different data back (in the form of a pivot table, a plot or just a single number). The next chapter will guide you through setting up these libraries on your computer (which is far easier than it might seem right now).

19: `NumPy` is a Python library for working with multi-dimensional arrays, not just matrices, efficiently. You can think of a table as a matrix of sorts, which is why `NumPy` can help when working with spreadsheets.

20: We'll also review functions in the next chapter.

21: And non-technical ways: `pandas`, `matplotlib`, and `NumPy` (as well as others) are all part of the *SciPy ecosystem*, a collection of open-source software for scientific computing in Python. The term *SciPy* is also used to refer to the community of people who use and develop this software, a conference dedicated to scientific computing in Python, and an actual Python library called `SciPy`. Don't worry if you're confused; I'll go through the parts you actually need to know.

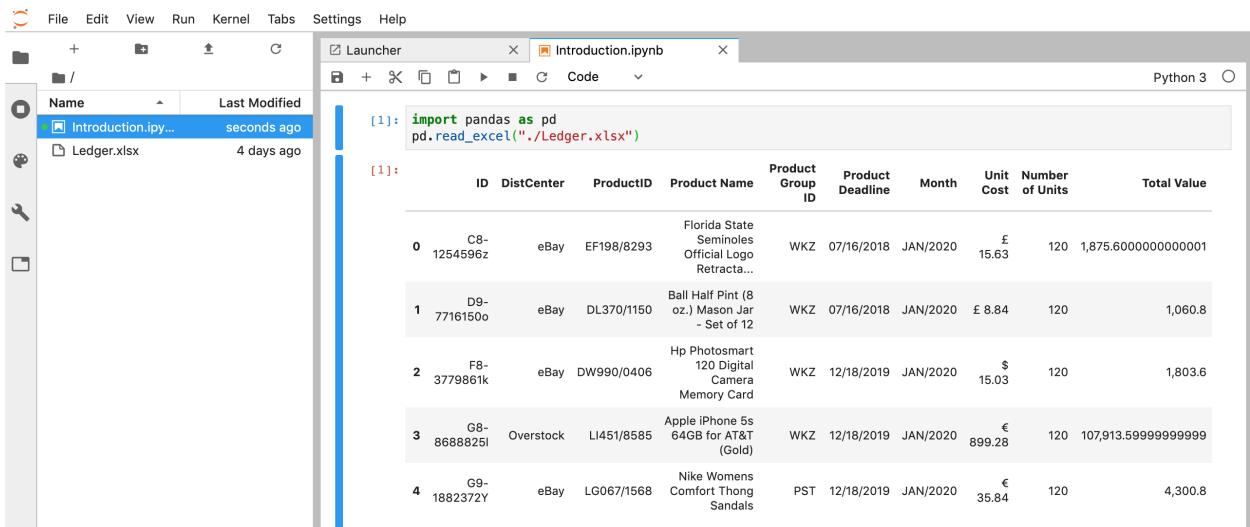


Figure 1.2: A screenshot of *JupyterLab*, the program you'll be using to write Python code. This example shows how to use `pandas` to read data from an Excel file, and what a table looks like in *JupyterLab*.

Two other tools — which are not Python libraries — will be essential in your Python adventure:

- ▶ **JupyterLab** is the software you'll use to write code. What Excel is to spreadsheets, JupyterLab is to data analysis files, which are called Jupyter notebooks (i.e., JupyterLab lets you create and edit Jupyter notebooks). Notebooks are designed to make coding by trial-and-error easy: you can run different pieces of Python code in a notebook and see what they do without having to design an entire program around them. Notebooks also allow you to mix different media types in the same document: code, tables, plots, images, and plain text. As you'll soon see, they're powerful tools for data analysis (and for learning). Just like the Python libraries I mentioned earlier, JupyterLab is open-source and free to use. The following chapter guides you through installing JupyterLab on your computer, but for now, you can see a screenshot of JupyterLab in figure 1.2.
- ▶ **Anaconda**²² is an installer that bundles all the open-source Python libraries and tools I mentioned so far into a single package that you can use to get up-and-running with Python data analysis code fast. What the Microsoft Office suite is to Excel, Anaconda is to the Python libraries for data analysis you'll be using throughout this book (except it's free to use and open-source).

22: Or the *Anaconda Distribution*.

These are just a few of Python's libraries and tools for data analysis but are perhaps the most relevant for accounting. There are many others online, and you'll soon discover them on your own.

How to use this book

This book has a lot of code in it, shown directly in the main text.

Try to read code examples, but also type them out by yourself in JupyterLab. You learn much faster when you type code by yourself, and you also get used to the mechanics of writing and running code (which will serve you well when you finish the book and have to navigate without a guide).

All code examples in the book look like this:

```
In [1]: print('hello') 1
print('python for accounting!') 2
```

```
Out [1]: hello
python for accounting!
```

The gray box at the top shows one or more Python *code statements* — the code here just prints some text, but all code examples will be similar in style to this one. When more than one line of code is listed in a code box, you'll see line numbers on the right of the box to make it easier to reference a particular line in the main text (e.g., line 1 prints a hello message).

The gray box code is what you need to run to produce the result shown in the blue box beneath it, which is the *code output*. Sometimes code doesn't produce any output (e.g., when you create a new variable), so there's no output box for that code. How exactly you run code and what the In [1]: and Out [1]: labels mean is covered in detail in the next chapter.

For the data analysis code you'll be writing, code output will often be a table. Output tables look like this:

```
In [2]: import pandas as pd 1
pd.read_excel('Q1Sales.xlsx') 2
3
```

```
Out [2]:   InvoiceNo      Channel      Product Name ... Unit Price Quantity Total
0       1532  Shoppe.com  Cannon Water Bom... ... 20.11      14 281.54
1       1533     Walcart    LEGO Ninja Turtl... ... 6.70        1  6.70
2       1534     Bullseye          Nan ... 11.67        5 58.35
3       1535     Bullseye  Transformers Age... ... 13.46        6 80.76
4       1535     Bullseye  Transformers Age... ... 13.46        6 80.76
...
14049    15581     Bullseye AC Adapter/Power... ... 28.72        8 229.76
14050    15582     Bullseye Cisco Systems Gi... ... 33.39        1 33.39
14051    15583 Understock.com Philips AJ3116M/... ... 4.18        1  4.18
14052    15584     iBay.com          Nan ... 4.78        25 119.50
14053    15585 Understock.com  Sirius Satellite... ... 33.16        2 66.32
```

[14054 rows x 12 columns]

The table above has 14 054 rows and 12 columns (you'll have to believe me), but you only see 10 of its rows and 6 of its columns (and an indicator at the bottom left that tells you how many rows and columns there are). I show tables in the book in this truncated form, with ... instead of actual rows or columns, to make them fit in the main text, and because tables in JupyterLab also get truncated like this — at first, you might find this annoying. You'll soon discover you can work with data even without seeing it all.

You'll be working with several accounting datasets, and for some of the chapters ahead, you'll have to download extra files. Each chapter has an associated web page (i.e., that's where the link in the bottom right margin below takes you) where you can download additional resources; more on getting set up in the next chapter.

Learning to code is like learning a new language: first, you learn the alphabet (i.e., what the different Python keywords are), then you learn common words and phrases (i.e., how to use Python with `pandas` and other Python libraries), and then you try to express yourself fluently (i.e., code an entire data analysis project). Just like learning a new language, you need practice to become good at coding: go through each example in the book, type it out by yourself, and you'll be fluent in Python before you know it.

Roadmap

There are four parts to this book:

- ▶ **PART ONE: PYTHON ABCs** introduces the building blocks of the Python programming language (i.e., how to define variables or functions, how to use lists and loops, and a few others). These building blocks are the foundation on which the data analysis tools used later in the book are built — and are also the glue that makes them all work together.
- ▶ **PART TWO: WORKING WITH TABLES** is where the rubber meets the road in using Python for accounting. This part of the book introduces the main features of `pandas`, the Python library you'll be using to work with tables — whether Excel spreadsheets, CSV files or any other kind of tabular data.
- ▶ **PART THREE: VISUALIZING DATA** shows you how to turn data into plots using some of Python's data visualization libraries (`matplotlib`, `seaborn`, and `hvplot`).
- ▶ **PART FOUR: SALES ANALYSIS PROJECT** guides you through an end-to-end data analysis project that uses Python, Jupyter notebooks, `pandas`, and a few other Python libraries. The project looks at a wholesale supplier's sales data and tries to identify which products

and sales channels are most profitable. This type of analysis is common in management accounting; even if your work doesn't involve analyzing sales or inventory, you'll find many ideas in this part that can help you structure your own data analysis projects. This last part brings together many of the tools you will have learned about and covers the areas of a typical analysis project that aren't related to code: setting and documenting specific questions for the project, organizing data and code files, finding answers, and sharing results.

The first three parts of the book have several chapters that introduce new programming ideas. These chapters are short and focused, and all build on top of each other; ideally you should try to read them one after the other. There are also five project chapters that don't introduce new programming ideas but guide you through applying what you've been learning in an accounting setting. These chapters will walk you through (in order): filtering and splitting a large Excel file into multiple sheets, reading and cleaning a general ledger exported from QuickBooks, mining product reviews, filling missing values in a sales dataset, and making a waterfall plot from a cash flow statement. The last part of the book is a sales analysis project in its entirety.

Throughout the book, you'll find several sections and even a few chapters whose titles start with the word "*Overthinking*". You can ignore these overthinking sections and chapters if you want to because they cover ideas that aren't strictly necessary in your learning journey. Still, you might find them interesting if you get bitten by the Python bug.

Good luck, and have fun!

2

Getting set up

This chapter guides you through getting set up with Python and its data analysis tools on your computer. You will likely want to use Python at work, so I'll show you how to install everything you need to get started on a computer running Windows 10 without any special privileges. If you have at least one folder on your work computer where you can copy files, you can install Python. If you're using another operating system (e.g., another version of Windows or macOS), the steps you need to take might be different, but you can easily adapt the instructions here for your setup.

Installing Python on your computer

There are a few different ways to install Python on your computer. However, my preferred method is using the (free) Anaconda installer that bundles Python with all the tools and libraries you need to get started. The Anaconda installer also includes an application called Anaconda Navigator that makes it easy to install and manage Python libraries on your computer.

To download the installer, head to www.anaconda.com/products/individual and find the download button. You will want the version of Anaconda that uses the most up-to-date Python (i.e., Python 3.8 or higher). If your computer is reasonably new (from the 2000s or later) and running Windows 10, download the 64-bit graphical installer for Windows.

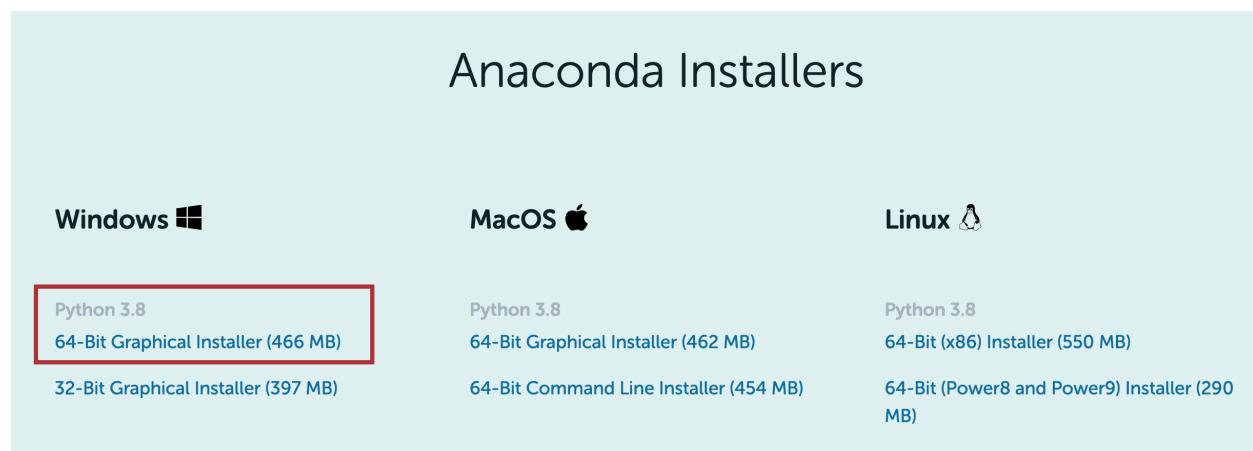


Figure 2.1: A screenshot of the Anaconda installer website, as of August 2020. To get the installer, click the download button at the top of the page and then select *64-bit Graphical Installer* under the Windows and Python 3.8 labels.

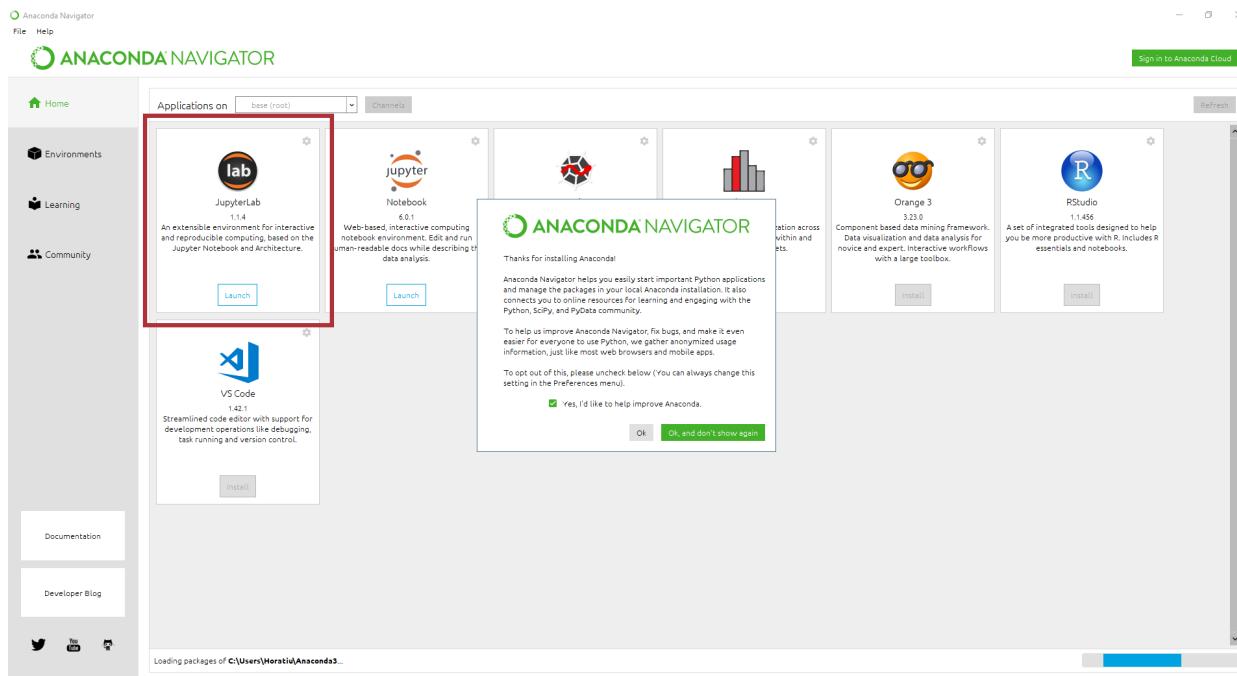


Figure 2.2: A screenshot of Anaconda Navigator. To get started, click “Launch” on the JupyterLab card – in this version of Anaconda Navigator, JupyterLab is the top left card.

When the download completes, run the installer; it will guide you through getting Python and the libraries you need up-and-running on your computer. The default options set things up so that you can run Python on your computer without any special privileges.

The installer will put all Python files in your local user directory (probably something similar to `C:\Users\<YOURUSERNAME>\Anaconda3`) — but you can change this location to any other folder when you get prompted. The entire installation takes up about 3GB of hard disk space, so make sure there is enough space on your drive, and the whole process takes a few minutes.

After your installation is complete, launch the Anaconda Navigator application, either by searching for it in your Start Menu or by going to your installation directory (e.g., `C:\Users\<YOURUSERNAME>\Anaconda3`) and double-clicking “*Anaconda Navigator.exe*”. Anaconda Navigator is the application you’ll be using to launch JupyterLab and manage your Python setup. You can also use it to install new Python libraries and update the ones already installed on your computer (we’ll use it to install one more Python library at the end of this chapter).

In the Navigator, click “*Launch*” on the JupyterLab card (your navigator window should look like figure 2.2, with JupyterLab in the top left corner). This will open JupyterLab as a new tab in your default web browser.

You’ll be using JupyterLab to write and run Python code. What

Excel is to spreadsheets, JupyterLab is to the code files you'll be working with, which are called Jupyter notebooks.¹ Jupyter notebooks are interactive documents that contain code, tables, plots, and narrative text together (more on notebooks later).

Unlike Excel, JupyterLab's interface is browser-based — JupyterLab still runs on your computer, just like Excel does, but you interact with it through your browser and not a regular application window. There are several reasons for its browser-based interface, but perhaps the most influential one is that it enabled JupyterLab developers to build one interface that works on all platforms. Even though JupyterLab's interface is browser-based, by default, all the code and data you use with it remain on your computer and are never shared over the web or your local network in any way. If you want to learn more about JupyterLab and Jupyter notebook security and privacy, visit jupyter-notebook.readthedocs.io/en/stable/security.

¹: Jupyter stands for Julia, Python and R, which were the first programming languages Jupyter notebooks were designed to support. They now support all sorts of programming languages, including Javascript.

Setting up your local workspace

Before you start writing any code, I recommend creating a folder somewhere on your computer to store all the Python files you'll be working with as you go through this book (what I call your local workspace). A new folder called “*Python for Accounting*” in your “*Documents*” should do the trick — you can create this new folder through your usual file navigator (e.g., Windows Explorer) or directly in the JupyterLab interface.

The next step in setting up your local workspace is getting the data files you'll need later in the book. For that, you can go to www.pythonforaccounting.com/setup and click the download button. You'll get a zip file with all the datasets and notebooks we'll be using, organized by parts and chapters. If you extract this zip file in your local workspace and everything goes well, your JupyterLab interface should look like figure 2.3 above. I'll introduce each dataset as it's needed in the following chapters, but have a browse if you're curious.

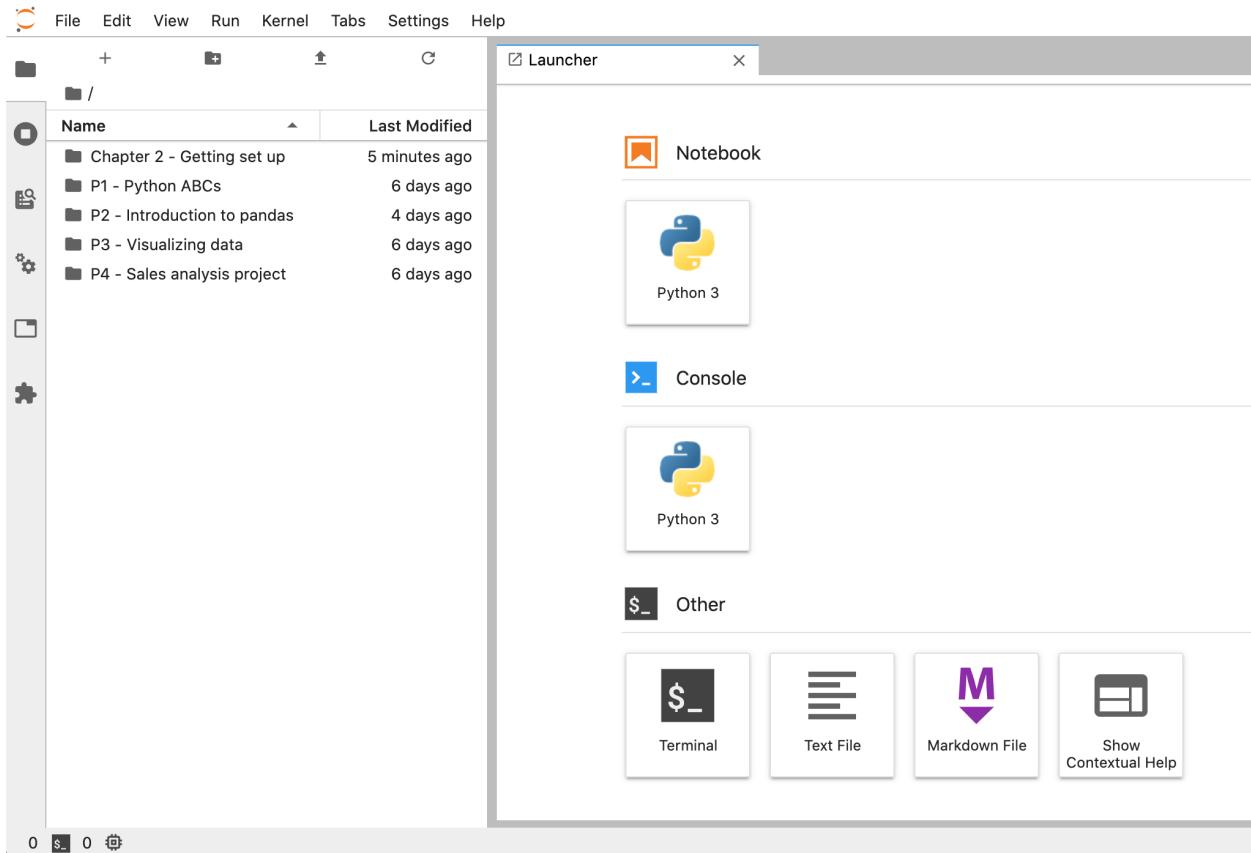


Figure 2.3: A screenshot of JupyterLab. This is the editor you'll be using to write Python code. It is to Python data analysis code files what Excel is to spreadsheets.

Using JupyterLab

Now that you have Python and JupyterLab installed on your computer, you're ready to write some code. For that, you need a Jupyter notebook: open the “*Chapter 2 - Getting set up*” folder in your workspace and click on the **File** → **New** → **Notebook** menu at the top of the interface, which will create and open a new notebook as a separate JupyterLab tab.

In this notebook (which is now called “*Untitled.ipynb*”), there is an empty text input area right at the top. This input area is a notebook *cell*, where you can write and run Python code.² In this cell, type the following code:

```
In [1]: print("hello, python for accounting!")
```

After you type the code, you need to run it. You can do that either by clicking the ► button right above the cell or pressing **Shift + Enter** (i.e., hold **Shift** and press **Enter**) on your keyboard to run the code in your current cell.³ After your code runs, you'll see its output right underneath the cell, as in figure 2.4 below.

If this is your first go at writing Python code, well done! You now qualify for nerd status.

2: I'll go into more details about notebook cells in just a bit. Notebook cells are similar to cells in Excel spreadsheets in that they can hold values but can also be used to run functions or code.

3: **Shift + Enter** is a really useful keyboard shortcut to remember: it runs the code in your selected notebook cell and displays its output underneath.

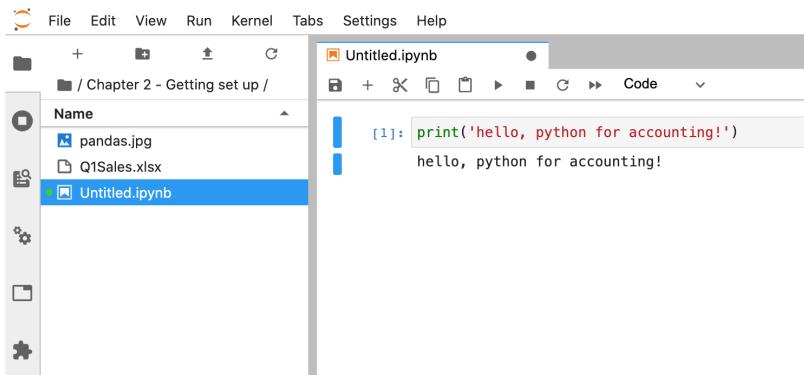


Figure 2.4: Your first line of Python code in a Jupyter notebook!

When you run a Jupyter notebook cell, your code is executed by the notebook kernel. In this case, because you’re running Python code, your notebook kernel is a Python interpreter (i.e., an application that knows how to turn Python code into instructions your computer can execute) — there are kernels available for various other programming languages, but we’re going to use just the Python kernel. This notebook kernel is a process running in the background on your computer, waiting for code.⁴ When you run a Jupyter notebook cell, JupyterLab sends your code to the notebook kernel, which translates it into instructions for your computer’s processor. After your code runs, the kernel collects any results from the processor and sends them back to the JupyterLab interface, where you can see them. Fortunately, you don’t have to worry about the details because JupyterLab handles them for you (i.e., it starts and stops kernels whenever you open or close JupyterLab), but it’s worth knowing how it all works.

Why are there so many parts in this machine? Because having a separate process that waits for instructions enables the trial-and-error, interactive style of writing code you’ll be using later. It might not seem like a big deal, but this kind of interactive computing is convenient for working with data because you can stop and check what is happening with your data at any point as you run your code. The alternative, writing an entire Python script or VBA macro, running it end-to-end, trying to figure out what went wrong with it and where, then editing and re-running your code is far more frustrating. This style of interactive coding is also helpful when learning how to code because you can quickly try different things and get instant feedback on what works and what doesn’t.

You’ll use Jupyter notebooks (with their Python kernels) and JupyterLab for all Python-related work in this book, so it’s worth spending some time getting familiar with these tools. The rest of this section walks you through some of their main features.

4: You can think of the kernel process as a watermill that keeps spinning, and the code you send it to process is the grain. When it has no grain to mill, it keeps spinning idly until you shut it down.

JupyterLab interface

JupyterLab is similar to Excel in that you have a main working area where you interact with data or other types⁵ of files. However, in JupyterLab you can't edit tables manually. To work with tables, you need to write and run some Python code. As a quick example, you can load the "Q1Sales.xlsx" file (which should be in your current folder) by running the following code:

```
In [2]: import pandas as pd
```

```
pd.read_excel('Q1Sales.xlsx')
```

	InvoiceNo	Channel	Product Name	Unit Price	Quantity	Total
0	1532	Shoppe.com	Cannon Water Bomb	20.11	14	281.54
1	1533	Walcart	LEGO Ninja Turtles	6.70	1	6.70
2	1534	Bullseye		11.67	5	58.35
3	1535	Bullseye	Transformers Age 0+	13.46	6	80.76
4	1535	Bullseye	Transformers Age 0+	13.46	6	80.76
...
14049	15581	Bullseye	AC Adapter/Power S...	28.72	8	229.76
14050	15582	Bullseye	Cisco Systems Giga...	33.39	1	33.39
14051	15583	Understock.com	Philips AJ3116M/37...	4.18	1	4.18
14052	15584	iBay.com		4.78	25	119.50
14053	15585	Understock.com	Sirius Satellite R...	33.16	2	66.32

[14054 rows x 12 columns]

Working with spreadsheets in Jupyter notebooks is what most of the book is about, you'll learn much more about how to do that in the following chapters. For now, let's unpack the JupyterLab interface a bit more.

The menu bar at the top of the interface (highlighted with a green border in figure 2.5) has several menus that list JupyterLab commands, together with their associated keyboard shortcuts. *File*, *Edit*, *View* are pretty standard menus across software tools, but in JupyterLab, you have some notebook-specific commands. Two menus that I use often are *Run* and *Kernel*. The *Run* menu has commands related to running cells in your current notebook. One of the most useful commands is *Run All Cells*, which does exactly what it says (it is useful when opening a notebook that already has a lot of code in it and running all the code at once, without having to run every cell separately).

Kernel is a new menu that you probably haven't seen elsewhere and contains commands related to managing the underlying Python interpreter process linked to your notebook. I mentioned before that each notebook has a Python process attached to it (that's what the kernel is). Sometimes this kernel process gets stuck, or you want to stop whatever it's doing because it is taking too long. When

5: At the time of writing, you can also view images, PDF, HTML, CSV or other plain text files. As JupyterLab evolves, it will likely allow you to open and edit different types of files as well.

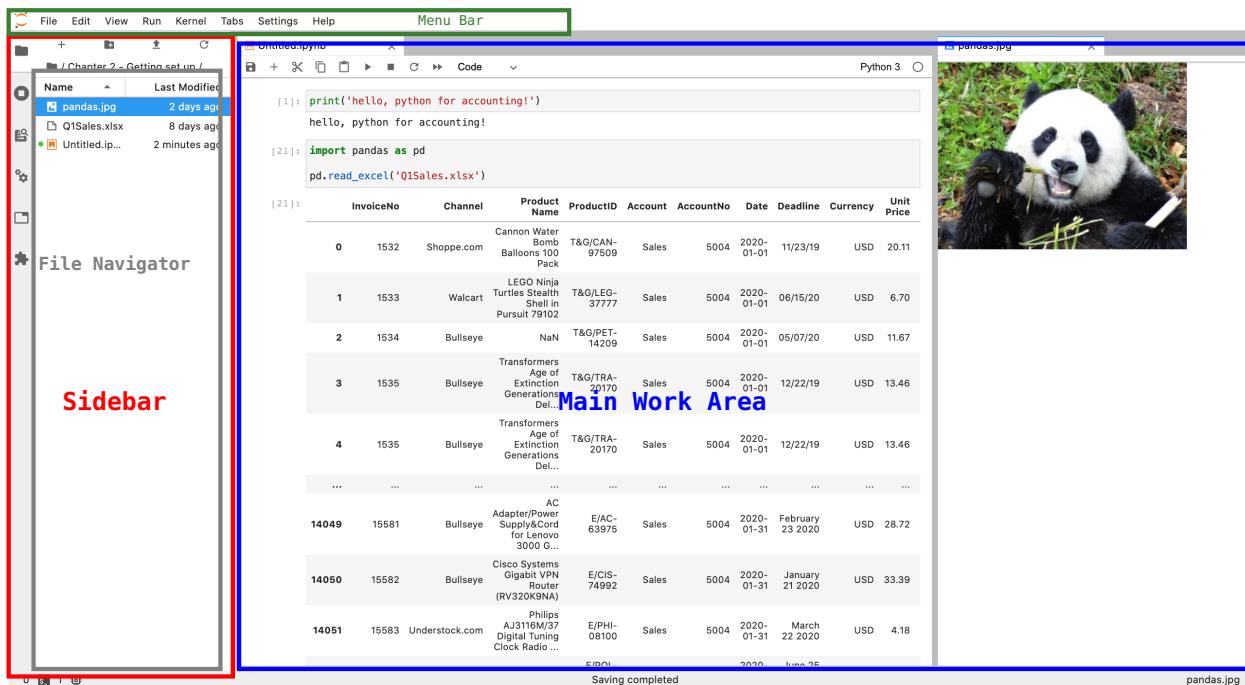


Figure 2.5: The JupyterLab interface. File navigator and sidebar on the left, main work area on the right. You can open notebooks, images, PDF or plain text files in the main work area and organize them in a grid layout using drag-and-drop.

that happens, you can use the *Interrupt Kernel* or *Restart Kernel...* commands from this menu.

How do you know your code is taking too long? There's an indicator in the top right of your notebook: it looks like an empty circle when the kernel is idle, and a full circle when the kernel is running some code. To the left of this indicator you can see several buttons that you can use to add or remove cells in your notebook (and some other similar controls).

The left sidebar (highlighted with a red border in figure 2.5) has some utilities, such as a file browser, a list of running Python processes (i.e., kernels) and terminals, the command palette where you can search for JupyterLab commands (e.g., *Restart Kernel*), and a list of open tabs. The left sidebar can be collapsed (to give you more screen space) or expanded by selecting "Show Left Sidebar" in the View menu.

The main work area (highlighted with a blue border in figure 2.5) is where you will be working with Jupyter notebooks. You can open notebooks, images, PDF or HTML files in the main work area, and you can arrange open tabs in a grid layout with drag-and-drop.

Jupyter notebooks

Jupyter notebooks are documents that keep all your data analysis work together. They allow you to combine text, images,

Python code, plots, and interactive visualizations in the same file to create a data story that is documented and reproducible.

Notebooks, like Excel spreadsheets, are just collections of cells. However, unlike Excel, notebook cells don't store tabular data — they can hold two types of content: Python code (examples of which you've seen already) or a kind of formatted text called Markdown. Markdown is just plain text enhanced with some styling commands you can use to indicate how text should look (it is a *markup* language, similar to HTML).

Regardless of type, notebook cells need to be *run*. What that means for code cells is that the code they contain gets executed (by the Python kernel attached to the notebook), and any code output gets displayed underneath the cell. Code output can be pretty much anything, from a single number to an entire table or an interactive plot, as you'll see later. Similarly, when you *run* Markdown cells, the text they contain is printed on the screen according to the styling commands you used. Let's take a look at code cells first.

Code cells

You've already used a code cell to `print('hello, python for accounting!')` earlier. Code cells are pretty straightforward — you use them to write code and run it.

Besides running code, these cells let you see what variables look like. We'll go over Python variables in more detail in the following chapter, but for now, consider the following example:

```
In [3]: message = "hello, python for accounting!"
```

This code cell creates a variable called `message` that stores the same text we've been printing so far. If you inspect `message` in a separate cell (i.e., type `message` in a new cell⁶ and run the cell), you'll see its value as the cell output:

```
In [4]: message
```

```
Out [4]: "hello, python for accounting!"
```

This style of quickly defining and checking variables makes notebooks powerful tools for learning how to code.

Not everything you put in a code cell gets displayed underneath when you run it, only what's on the last line in that cell. For instance, if you inspect several variables in the same cell, only the variable on the last line of the cell gets displayed:

6: Remember you can add new cells by clicking on the **+** button right above the notebook.

```
In [5]: welcome_message = "hello, python for accounting!"  
goodbye_message = "goodbye, python for accounting!"
```

```
welcome_message  
goodbye_message
```

```
Out [5]: "goodbye, python for accounting!"
```

You may have already noticed that code cells have a numbered label in front of them — similar to the `In [1]:` label shown before code examples in this book. These labels indicate the order in which you run code cells in your notebook.

```
[1]: print("hello, python for accounting!")  
hello, python for accounting!
```

Figure 2.6: Numbered Jupyter notebook code cell. The number in front of the cell indicates execution order for that cell.

Jupyter notebooks keep track of the order you run code cells because cell execution order is more important than the relative position of cells in a Jupyter notebook. The easiest way to understand this is through an example — consider the following cells:

```
In [1]: message = "hello, python for accounting!"
```

```
In [2]: print(message)
```

```
hello, python for accounting!
```

```
In [3]: message = "goodbye, python for accounting!"
```

Running each of these cells in a Jupyter notebook, one after the other, produces the output you see above.

But you can run notebook cells in any order (i.e., you can easily select any cell in your notebook and run it, regardless of its place in the notebook). In the example below, I ran the last cell in the notebook right after running the first one, then ran the middle cell — notice the cell execution counters on the left of each cell are not in order:

```
In [1]: message = "hello, python for accounting!"
```

```
In [3]: print(message)
```

```
goodbye, python for accounting!
```

```
In [2]: message = "goodbye, python for accounting!"
```

The middle cell above prints `goodbye, python for accounting!` because I assigned the value in the last cell to `message` before printing it (i.e., I didn't run the code cells in sequence, one after the other).

Running cells in an arbitrary order is a *feature* of Jupyter notebooks. It makes working with notebooks different from working with regular code, where everything runs in sequence, from top to bottom. This feature is useful for writing code in an interactive

way: you can add new cells anywhere in your notebook to inspect or modify variables, regardless of where you defined them initially. However, this feature can become confusing when you have a lot of code cells in your notebooks.

In general, even though you can create and run code cells in any order you want to, it is good practice to keep code cells in the same order as they need to run so that you can follow the sequence of steps in your notebooks.

Markdown cells

Markdown cells are useful in adding text that describes your work to your notebooks. Documentation is essential when working with data because analyses often involve intricate steps that need to be explained in plain English.⁷

To add a Markdown cell to your notebook, click on the **+** button right above your code cell (you can also move cells around using drag-and-drop if you don't like where they are). By default, all new cells are code cells. You can change the type of your new cell by going to the notebook toolbar and selecting Markdown, as shown in figure 2.7.

7: You can use Markdown cells to describe the business context for your analysis (e.g., who the stakeholders are, what the business goals of your analysis are), or explain complex logic in your process (e.g., why a metric is computed using only a sample of the original data).



Figure 2.7: Changing cell type to Markdown using the notebook toolbar in JupyterLab.

You can now start writing Markdown text in this cell. For example, to give the notebook a title and some context, add the following content to this empty cell:

```
# My First Notebook

Notebooks are great! Use them to:
- write code
- make plots
- document your process
```

You can now either click the **▶** button in the notebook toolbar or press **Shift + Enter** to run the cell and render this text. You should see the title rendered in large, bold font. If you want to change this text, just double-click anywhere on the rendered text.

Figure 2.8 below shows a more complete reference for Markdown styles and formats you can use in Jupyter notebooks.

Whenever you want to use Markdown text but forget to change cell type, you will see the following error:

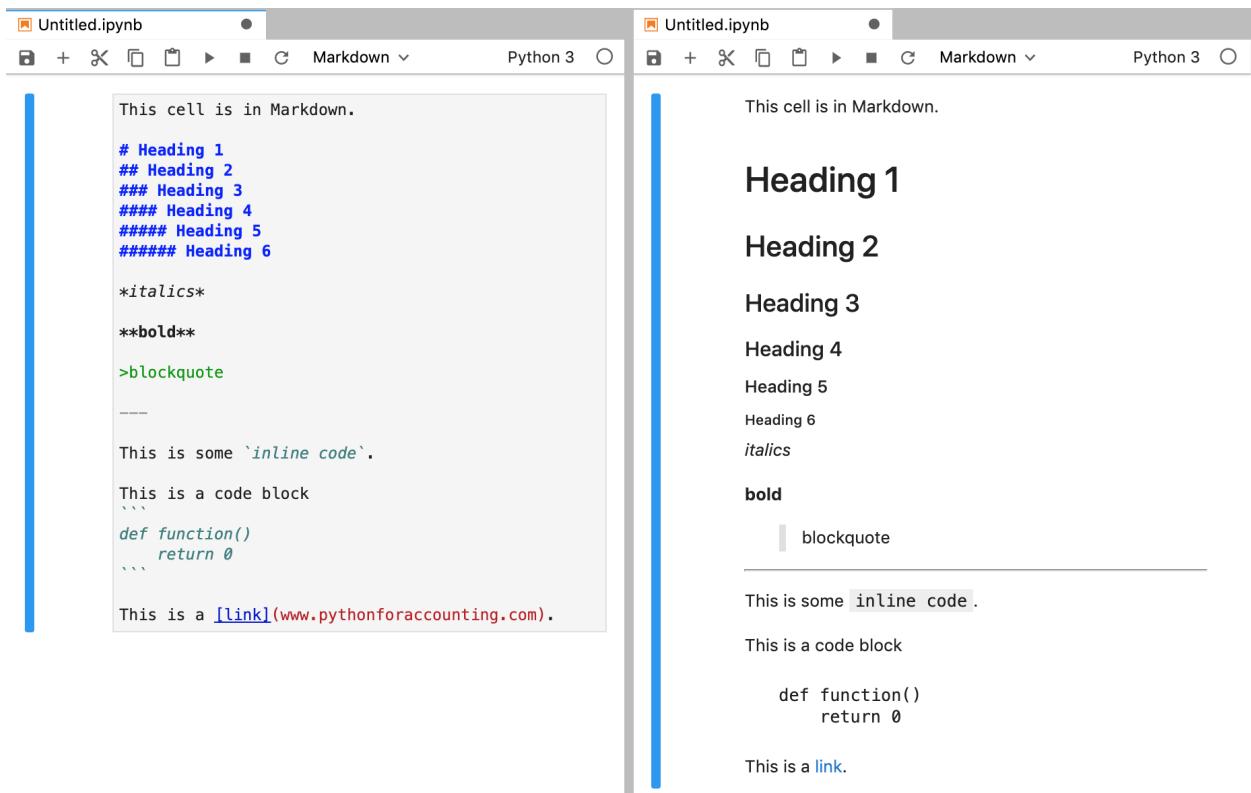


Figure 2.8: An almost complete Markdown reference for Jupyter notebooks. This reference shows side-by-side views of the same notebook in JupyterLab: the left view shows the Markdown text in edit mode, whereas the right view shows the rendered Markdown.

```

File "<ipython-input-257-68e6242d530e>", line 3
  Notebooks are awesome! Use them to:
  ^
SyntaxError: invalid syntax

```

The error above tells you that JupyterLab tried to run the contents of your cell as Python code but couldn't (because it's not code, it's just some text).

Command and edit modes

In Excel, if you want to edit a cell's contents, you need to double-click that cell. If you only click the cell once, you can use the keys on your keyboard to move around your spreadsheet or quickly issue commands instead of editing cell contents — this is Excel's *command mode*.

Similarly, when working with Jupyter notebooks, the keyboard does different things depending on what mode you are in: in edit mode, key presses change the contents of a cell, whereas, in command mode, you can use different key combinations (i.e., keyboard shortcuts) to issue commands. When you open a Jupyter

notebook in JupyterLab, you will see a mode indicator in the bottom right corner.



In contrast to Excel, when using Jupyter notebooks, if you click a cell once, you enter edit mode and can change that cell's contents. If you want to leave edit mode, you can press `Esc` or click outside any cell in the notebook, which switches back to command mode. Regardless of mode, your currently selected cell becomes highlighted with a blue border, as shown in figure 2.6.⁸

Both Excel and Jupyter notebooks have a command mode because in command mode you can use the keyboard to navigate the user interface and issue commands faster. These commands usually apply to your selected cell (e.g., pressing `c` in command mode copies your selected cell). Some of the most useful keyboard shortcuts you can use in command mode are listed in table 2.1 — you don't have to remember them all, but remembering **Shift + Enter** for running cells, **a** and **b** for adding cells above or below the current cell, and **dd** to delete the current cell will help you use notebooks faster.

Keyboard shortcut	Description
<code>Enter</code>	Enter edit mode
<code>Shift + Enter</code>	Run cell
<code>Up arrow or k</code>	Select cell above current cell
<code>Down arrow or j</code>	Select cell below current cell
<code>s</code>	Save notebook
<code>y</code>	Change cell type to code
<code>m</code>	Change cell type to markdown
<code>a</code>	Create new cell above current cell
<code>b</code>	Create new cell below current cell
<code>x</code>	Cut cell
<code>c</code>	Copy cell
<code>v</code>	Paste cell
<code>dd</code>	Delete cell
<code>z</code>	Undo previous cell operation
<code>Shift + z</code>	Redo previous cell operation
<code>Ctrl + Shift +]</code>	Go to next JupyterLab tab
<code>Ctrl + Shift + [</code>	Go to previous JupyterLab tab
<code>Alt + w</code>	Close current JupyterLab tab
<code>i</code>	Interrupt kernel
<code>00</code>	Restart kernel

Figure 2.9: JupyterLab notebook mode indicator shown in the bottom right corner of the interface.

8: In command mode, you can drag-and-drop cells to rearrange them in your notebook (you need to put your mouse cursor on top of the numbering in front of the actual cell for drag-and-drop to work). You can also select multiple cells at once and move them around.

Table 2.1: Jupyter notebook keyboard shortcuts in command mode. Double characters mean a quick succession of presses on the same key. You can edit these shortcuts and add more by going to “Settings”, “Advanced Settings Editor” then “Keyboard Shortcuts”.

Using Anaconda Navigator

Anaconda Navigator is the application you use to launch JupyterLab and manage Python libraries installed on your computer (without writing code or using the command-line). You already used it to launch JupyterLab. It comes with a bunch of Python-related application that we will not use in the book, but you might want to explore on your own (such as VSCode, which is a code editor similar to JupyterLab).

Besides launching applications, Anaconda Navigator can install or update Python libraries. Later in the book, we will be using a plotting library called `hvplot` that, as of writing, does not come with Anaconda, so you have to install it yourself.

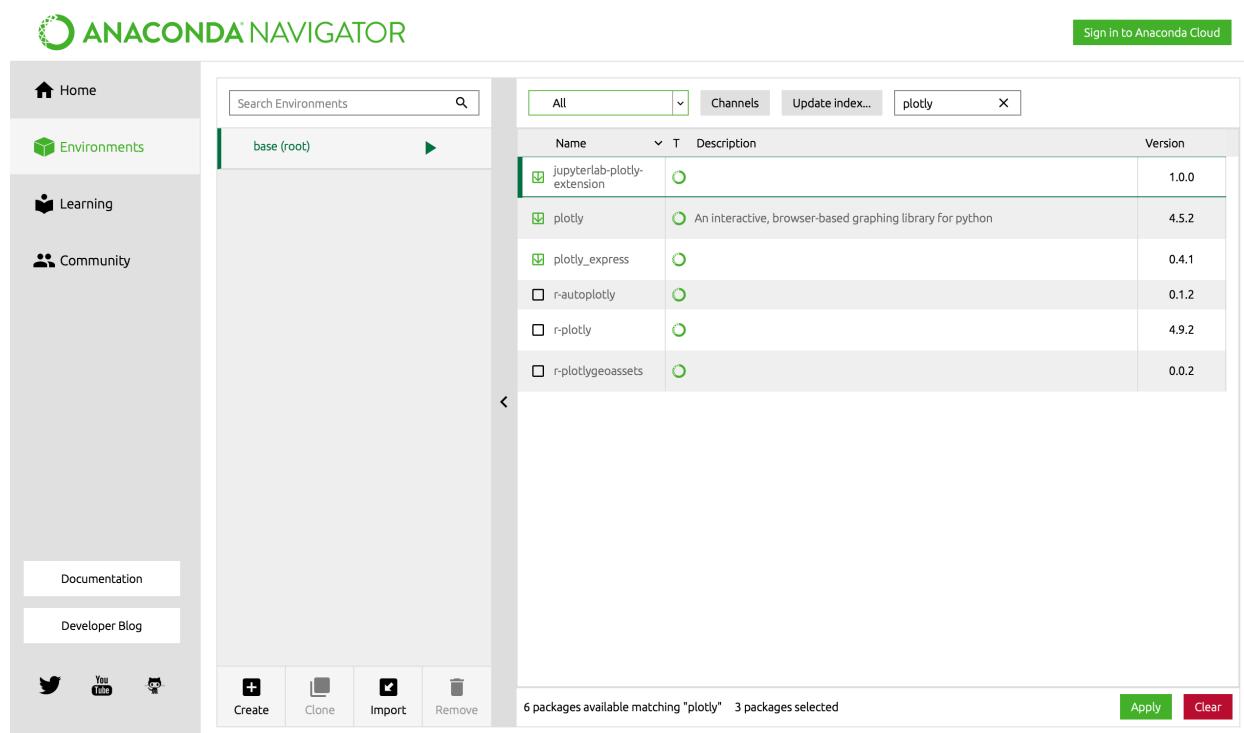


Figure 2.10: Using Anaconda Navigator to install `hvplot`.

To install Python libraries (also called packages), open the “*Environments*” tab (on the left side panel) and type “*hvplot*” in the packages search bar (the search bar on the right, not the “*Search Environments*” search bar).⁹ From the package type drop-down to the left of the package search bar, select “*All*”. You should see a list of packages that have the term “*hvplot*” in their name — select the library called just “*hvplot*” by clicking the checkbox in front of its name, and then click the “*Apply*” button at the bottom right of the screen. You will see a dialog showing you what packages will be installed; click “*Apply*” again and wait for the new library to get downloaded and installed on your computer.

9: More on libraries and packages in the following chapters.

One more thing you can do with Anaconda Navigator is quickly access various Python tutorials and reference documents. If you go to the “*Learning*” tab in the side panel (right below “*Environments*”), you will see links to different resources that can help you master Python and its libraries. Feel free to explore as many of these as you want, but the next chapters will go over everything you need to know.

Summary

This chapter went through getting up-and-running with Python and its data analysis tools on your computer. I introduced Jupyter-Lab as the editor you’ll be using to work with Jupyter notebooks. Jupyter notebooks are data analysis documents that allow you to write and run Python code but also display tables, plots, and styled text — all in the same file.

Now, let’s start learning Python.

PYTHON ABCs

PART ONE

Python has a large ecosystem of libraries and tools for data analysis. However, to tap into this vast ecosystem of tools, you first need to be familiar with the Python language itself.

This part of the book introduces the building blocks of the Python programming language (i.e., how to define variables or functions, how to work with lists and loops, and many more). These building blocks are the foundation on which the data analysis tools you'll be using later in the book are built — and they are also the glue that makes them all work together.

As you go through this part of the book, at times you'll feel overwhelmed with information. Keep in mind that you don't have to remember everything there is to know about Python to be productive with it; just reading about Python's core features is enough to get you going. Later in the book, as you start working with tables, we will revisit many of the concepts discussed in these early chapters — you'll have plenty of time to get used to them.

Remember that the term "*Python*" actually refers to three distinct concepts: a set of rules that define the Python programming language; a computer program that knows how to turn Python code into instructions for your computer to execute called the Python interpreter; and a collection of Python packages that you can use to simplify various coding tasks called the Python standard library. The following chapters focus on the Python programming language rules and show you how to write code that the Python interpreter can read and run without errors. If you followed the setup instructions from the previous chapter, and installed Python on your computer using Anaconda, you can run all the code examples in the following chapters in a Jupyter notebook, using JupyterLab.

When you're set, launch JupyterLab, go to your workspace, open the first notebook in part one, and let's get started.

3

A quick look at Python code

Python code is designed to read, as much as possible, like plain English. Even so, it has a few keywords and symbols that instruct your computer how to run the code you write that need some explanation. This collection of keywords and symbols and the rules defining how they can be used together correctly is called the Python language *syntax*.

Consider the following code example:

```
In[1]: # numbers to split
numbers = [-9, 1, -5, 4, -3, 0, 4, 10]

# define limit
limit = 0

# define two empty lists as placeholders
less = []; greater = [];

# go through numbers and split
# them based on limit value
for i in numbers:
    if (i < limit):
        less.append(i)
    elif (i > limit):
        greater.append(i)
    else:
        pass

# print results
print('Negative numbers:', less)
print('Positive numbers:', greater)

Negative numbers: [-9, -5, -3]
Positive numbers: [1, 4, 4, 10]
```

If you've never seen code before, there's a lot to digest in the example above. Try to read the code above as plain English — you'll be surprised how easy it is to understand. However, if this example doesn't make a lot of sense right now, don't worry; we'll take a closer look at the various parts of Python used above in more detail, starting with the following chapter.

In a nutshell, the code above defines a list of numbers and a limit value. It then goes through each number in the list and adds it to one of two placeholders, based on how each number compares to the limit value (less than the limit or greater than the limit). Finally, it prints the two placeholders on your screen.

This code example, while not very exciting, illustrates Python's essential syntax rules:

- ▶ **Indentation matters.** Python uses indentation to denote *code blocks* (VBA uses different keywords to denote code blocks, other programming languages, such as C or Java, use curly brackets).¹

In this example, the code under the `for` keyword is indented. This indentation marks a code block that is repeated for every number in the `numbers` list (you can have multiple lines of code in the same indented code block).

How much indentation should you use? It doesn't matter, as long as you always use the same amount of whitespace to indent blocks throughout your code, at all levels (e.g., one tab character, four space characters).² You'll probably use Tab to indent code blocks, but you can use spaces as well. If you want to indent code blocks within other code blocks, like the `if-else` statement on lines 13-17 in the example, you need to use extra indentation.

1: Python uses indentation (and not curly brackets) to denote code blocks because it makes code look more like plain English text.

2: By default, JupyterLab will automatically add an indent to your code when it thinks one is needed (i.e., after you type a statement followed by a colon and press Enter). Try it!

- ▶ **The line above an indented code block must end with a colon (i.e., :).** Notice the use of colons after each conditional statement in the `if-else` block or after the `for` statement on line 12 .
- ▶ **To end a statement, move to a new line.** On line 2 in the example above, you assign the value 0 to a variable called `limit`. The end of this assignment statement is marked by the end of the line. All other code statements are similarly terminated.
- ▶ **Semicolons can end a statement too.** Semicolons (i.e., ;) can also end Python statements, as in the example above on line 8 . The semicolon is useful when you want to put multiple statements on the same line, but most Python style guides discourage its use (you might sometimes see it in code examples online).
- ▶ **Whitespace on the same line does not matter.** The following lines of code are equivalent:

```
my_sum = 1+2+3
my_sum = 1 + 2 + 3
my_sum = 1 + 2 + 3
```

1
2
3

Most Python users would agree that the second line is easier to read than the other two — and, indeed, Python style guides recommend using a single space around operators to make code easier to read. However, all three lines have the same effect.

- ▶ **Comments are marked by #.** In Python, anything that comes after the hash character (i.e., #) on the same line is ignored when the code runs. This is how you can use plain English to explain parts of your code (as in the example above on lines 1, 4, 7, 10, 20).

The syntax rules we just reviewed are part of the Python programming language (much like punctuation rules are part of the English language). They describe *how* to write valid Python code, but not *what* different Python language constructs mean and what you can do with them. Next, let's take a look at variables and how you create them in Python.

4

Variables and operators

Variables are the building blocks of programming. They are *names* given to *pieces of data* that you reference throughout your code.

In Python, to create a new variable, you type a name and assign it a value using the equals sign.

```
In [1]: message = 'hello, python for accounting world!'
```

You just created a variable called `message` with the value of `'hello, python for accounting world!'`.

You can create different types of variables depending on what kind of data you need to work with (i.e., numbers, tables, images, etc.). We'll go over different variable types in the next chapter, but first, let's review some of the less obvious features of Python variables.

Everything is an object

In Python, every variable is an object.¹ This means that all Python variables have associated *attributes* and *methods* you can access by typing a period after their name. For instance, to make the `message` variable you created earlier uppercase, you can type a period after its name and use the `upper` method:

```
In [2]: message.upper()
```

```
Out [2]: 'HELLO, PYTHON FOR ACCOUNTING WORLD!'
```

Similarly, to add a number to a list² you can use the `append` method on a list variable:

```
In [3]: numbers = [10, 20, 30]          1
       numbers.append(100)            2
                                3
       numbers                      4
```

```
Out [3]: [10, 20, 30, 100]
```

Notice the dot after `numbers` and before `append` on line 2, the parentheses after `append` and the *method argument* (which is the number `100`) in between — this style of calling methods on objects is called *dot syntax*.

The parentheses at the end of a method name tell Python to run the method code. If you try to call a method but forget the parentheses

1: In fact, everything is an object in Python, even functions or libraries.

2: More on lists in chapter 6.

at the end, Python doesn't run the method code; instead, it gives you a description of the method.³

```
In [4]: numbers = [10, 20, 30]
numbers.append
```

```
Out [4]: <function list.append(object, /)>
```

Attributes are values associated with an object, whereas methods are functions associated with an object (i.e., attributes point to some information, methods do something).⁴ You can access object attributes the same way you access methods, the only difference being that you don't use parentheses at the end of an attribute name when you want to access it.

Even simple variables, such as `numbers`, have methods and attributes associated with them. For example, on a whole number variable (i.e., an integer) you can access the `real` and `imag` attributes or call the `bit_length` method:

```
In [5]: x = 5
```

```
print(x.real)
print(x.imag)
```

```
Out [5]: 5
0
```

```
In [6]: x.bit_length()
```

```
Out [6]: 3
```

This is just an example to show that even plain numbers have methods associated to them in Python; you will likely never use this method (I haven't).

Objects are a central concept in a type of computer programming called *object-oriented programming*. Python is an object-oriented programming language, which means everything you do in Python involves objects. I won't go into the details of how you define your own types of objects in Python because that's not necessary for the data analysis code we cover later.⁵

A type of object you'll be working with often in the next chapters is the `DataFrame`. `DataFrame` variables are how Excel spreadsheets are represented in Python — an entire table becomes a `DataFrame` variable in Python code. To create a `DataFrame` variable (i.e., a `DataFrame` object) from an Excel file called "Q1Sales.xlsx", you can use the following code:

```
In [7]: import pandas as pd
df = pd.read_excel("Q1Sales.xlsx")
```

3: When you see this kind of output after running a code cell, you forgot the parentheses after the method name.

4: *Method* and *function* are two terms that describe the same thing: a piece of code that does something. When used with objects, like in the examples above, they're called methods. When used by themselves, without being attached to a particular variable or object, they're called functions.

These are available in the unlikely case you need to work with complex numbers.

5: But you can read more about Python's objects at docs.python.org/tutorial/classes.

Here, the `df` variable contains *all* the information in the first sheet of “*Q1Sales.xlsx*” — this includes values in each column of the table, information about what types of data is stored in each column, the number of rows, table headers, and many more. It *encapsulates* many different pieces of data into a single variable and makes it easy to work with those data all at once. Methods that are available on the `df` object have access to its internal data (e.g., column names, individual values in the table) and often produce a result using some of them. For example, you can compute the column-wise sum of this table by calling the `sum` method on the `DataFrame` variable:

```
In [8]: df.sum()
```

We'll go into more detail about `DataFrame` objects in the next part of the book, and you'll get a clearer idea of how they work then. For now, remember that all variables in Python are objects, including the ones we've been using so far.

Point me to the values

In contrast to many other programming languages, Python is *dynamically typed*, which means you don't have to declare the type of variables you create when you create them; in Python, you can do things like this:

```
In [9]: value = 100
message = 'hello, python for accounting world!'
numbers = [1.1, 1.2, 1.3]
```

This style of writing code is what makes Python so easy to read and write. However, this lightness of coding comes with a caveat you should be aware of: Python variables *point* to values, they don't “*store*” values themselves.

I will go over Python lists in more detail later but, for now, consider this simple example that uses two list variables:

```
In [10]: a = [10, 20, 30]
b = a

print(a)
print(b)
```

```
Out [10]: [10, 20, 30]
[10, 20, 30]
```

This example creates two variables (i.e., `a` and `b`) that both *point* to the same sequence of values. If you modify this sequence by appending a new number to it, through either of the variables, the other variable “*changes*” as well:

```
In [11]: a = [10, 20, 30]
b = a

# add a new item to list
a.append(40)

print(a)
print(b)
```

```
Out [11]: [10, 20, 30, 40]
[10, 20, 30, 40]
```

This behavior might seem confusing at first, particularly if you think of variables as “*storing*” values. But variables in Python point to some data, and you use their name to access that data.⁶ If you create two (or more) variables and make them point to the same data, as I did in the example above and then change the data through either of the variables, both will be affected.

In Python, you make a variable point to some data by using the equals sign (i.e., assigning it a new value). In our example, if you change the data that variable `a` points to entirely (i.e., by using the equals sign again), `b` does not change because it still points to the original data:

```
In [12]: a = [10, 20, 30]
b = a

a = [10, 20, 30, 40]

print(a)
print(b)
```

```
Out [12]: [10, 20, 30, 40]
[10, 20, 30]
```

Variables that point to simpler data, such as a single number or a piece of text (not a list of numbers, as in the example above), are also pointers. However, simple data types like numbers or text cannot be modified in-place like the list variables above (i.e., they are *immutable*). If you want to modify a number or text variable in Python, you always have to use the equals sign, which means you always change the value they point to. For example:

```
In [13]: x = 10
y = x

x = x + 10

print(x)
print(y)
```

6: The data that variables point to is stored in a random location in your computer’s memory, where Python can access it.

Out [13]: 20
10

In the example above, there is no other way to increase the value of `x` by 10 except by using the equals sign again and making `x` point to a different value entirely. The same goes for text variables; as such working with numbers or text always behaves the way you expect it, in contrast to working with more complex objects (like lists or `DataFrame` objects) which may get tricky if you use multiple variables that point to the same data.

This variables-as-pointers feature is less of a problem than you may think it is: most of the time, you will intuitively use variables without problems. However, clarifying this pointer behavior now and developing the right mental model for Python variables will save some headache later on, which is why I mention it early in the book. If you start thinking of variables as just labels pointing to some data (data that is actually stored elsewhere on your computer), this behavior starts making sense as well.

The cycle of a variable's life

When you create a new variable in a notebook cell, you can use that variable in any of the other code cells in your notebook. It remains available for you to use until you shut down or restart your notebook (i.e., from the “*Kernel*” menu in JupyterLab).

Variables⁷ occupy space in your computer’s working memory (not on your actual drive, but in your computer’s *random access memory*), so the more variables you create, the less free working memory you will have on your computer.⁸ When you stop your notebook (e.g., by going to the “*Kernel*” menu and selecting “*Shut Down Kernel*”), Python erases all notebook variables from your computer memory, freeing it up for other programs that need it.

At times, you will want to delete a variable manually because you don’t need it anymore or because you created it by mistake (e.g., you made a typo in the variable name). To delete a variable from your notebook (and also remove the data it points to from your computer’s memory), you can use the `del` keyword:

In [14]: `message = 'hello, python for accounting!'`
`del message`

7: Variables and the data they point to. You can think of the data variables point to as a temporary files stored in your computer’s working memory: it has a name, a path and some content, just like a regular file. The only difference is that you cannot access it through a file navigator, only reference it through the variable name.

8: This means you can run out of working memory if you create a lot of variables in your notebooks. While this is not a problem for the tiny variable we will be working with in this chapter, as you start working with large tables loaded as `DataFrame` variables, you might reach the limits of your computer’s memory.

After you delete a variable using the `del` keyword, if you try to access it or use it in any way, you will see the following error:

In [15]: `message`

```
NameError                                     Traceback (most recent call last)
<ipython-input-25-1133e3acf0a4> in <module>
      1 message
NameError: name 'message' is not defined
```

NameErrors tell you that the name (of a variable or function) you are trying to use is not defined. When you see one, check that you defined a variable or function with that name in your notebook or that you haven't mistyped its name when trying to use it — often, you will find it is the latter.

Remember that all variables you create in a notebook get removed from your computer's working memory when you restart the Python process associated with a notebook (the kernel) or when you shut down the notebook entirely. However, your code doesn't get lost. All you need to do to get the same variables in your notebook is run the code cells that define them again (either by running individual cells or by going to the "Kernel" menu and selecting "Run All").

Operators

Now that you know how to create variables let's take a look at what you can do with them. Python has several operators⁹ you can use to work with variables — many of these are familiar mathematical operators.

9: Of the smooth variety.

Arithmetic operators

Table 4.1 below lists arithmetic operators available in Python. These operators can be used as you would expect, and you can combine multiple operators using parentheses:

```
In [16]: x = 5          1
         y = 2          2
         x * y + (x // y) 3
                           4

Out [16]: 12
```

You may have noticed already that Python supports order of operations (e.g., multiplication is performed before addition):

```
In [17]: 2 * 3 + 4
Out [17]: 10
```

Table 4.1: Arithmetic operators available in Python. Examples use $x=5$ and $y=2$.

Name	Example	Result	Description
Addition	<code>x + y</code>	7	Sum of x and y .
Subtraction	<code>x - y</code>	3	Difference of x and y .
Multiplication	<code>x * y</code>	10	Product of x and y .
Division	<code>x / y</code>	2.5	Division of x and y .
Modulus	<code>x % y</code>	1	The remainder of x divided by y .
Floor division	<code>x // y</code>	2	Division of x and y without fractional part.
Exponentiation	<code>x ** y</code>	25	x raised to the power of y .
Negation	<code>-x</code>	-5	Negative of x .

Something to be aware of is that operator precedence can be confusing at times (particularly in long formulas), so it's a good idea to use parentheses when you are unsure of which operator comes first. For example, negation has lower precedence than exponentiation:

```
In [18]: -10 ** 2
```

```
Out [18]: -100
```

```
In [19]: (-10) ** 2
```

```
Out [19]: 100
```

Comparison operators

Python comes with standard comparison operators, which are useful in conditional statements that can be evaluated as true or false. Table 4.2 lists the comparison operators available in Python.

Table 4.2: Comparison operators available in Python. Examples use $x=5$, $y=2$.

Name	Example	Result	Description
Equal	<code>x == y</code>	<code>False</code>	Checks x equal to y .
Not equal	<code>x != y</code>	<code>True</code>	Checks x not equal to y .
Less than	<code>x < y</code>	<code>False</code>	Checks x less than y .
Greater than	<code>x > y</code>	<code>True</code>	Checks x greater than y .
Less than or equal	<code>x <= y</code>	<code>False</code>	Checks x less than or equal to y .
Greater than or equal	<code>x >= y</code>	<code>True</code>	Checks x greater than or equal to y .

You can string multiple operators together if you need to evaluate a more complicated relationship between variables:

```
In [20]: x = 5
```

```
1 < x < 10
```

```
Out [20]: True
```

You can also combine comparison operators with arithmetic operators to express any possible check on numbers. A typical example is checking whether a number is even with the *modulus* operator:

```
In [21]: 24 % 2 == 0
```

```
Out [21]: True
```

There are a few other operators available in Python that you can use with conditional statements (e.g., `and` and `or` to connect multiple comparison operators). These operators are most useful in `if-else` statements, we'll come back to them in a few chapters.

Assignment operators

By now, you already know that the main assignment operator in Python is the equals sign:

```
In [22]: x = 5
```

You might come across enhanced versions of the equals sign that include one of the arithmetic operators I mentioned earlier. For instance, the following examples are equivalent:

```
In [23]: x = 5
```

```
x = x + 2
```

```
x
```

```
Out [23]: 7
```

```
In [24]: # equivalent to:
```

```
x = 5
```

```
x += 2
```

```
x
```

```
Out [24]: 7
```

The same style of assignment works with any of the operators listed in table 4.1:

```
In [25]: x = 5
```

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

```
# equivalent to:
```

```
x = 5
```

```
x **= 2
```

```
In [26]: x
```

Out [26]: 25

You will come across the more compact style in code examples elsewhere, which is why I briefly mention it here.

Summary

In this chapter, you saw that all Python variables are objects: they have attributes and methods associated with them. Calling methods on objects is how you'll do most of your work when using Python code. We also looked at Python's operators, which you'll use to work with numbers, either by themselves or in table columns. Let's see how you create different types of variables in Python next.

5

Python's built-in data types

Python has five simple types of data built-in — most of them you've already used. They are listed below, together with code examples for each.

Table 5.1: Simple built-in data types in Python.

Name	Keyword	Example	Description
Integer	<code>int</code>	<code>n = -10</code>	Whole numbers (i.e., integers).
Float	<code>float</code>	<code>n = 2.5</code>	Decimal numbers (i.e., floating point numbers).
String	<code>str</code>	<code>n = 'hello'</code>	Text or single characters.
Boolean	<code>bool</code>	<code>n = False</code>	<code>True</code> or <code>False</code> values.
NoneType	<code>None</code>	<code>n = None</code>	Special indicator to show the absence of a value.

These types of data are built into Python, which means they are always available for you to use — unlike, for instance, the `DataFrame` type, which is available only after you import the `pandas` library, as you will see later. All Python libraries and tools extend these simple types into more complicated ones.

You'll often forget what type your variables are. When that happens, you can use the built-in `type` function to find out:

```
In [1]: x = 10
        type(x)

Out[1]: int

In [2]: y = 'hello!'
        type(y)

Out[2]: str
```

Let's unpack each built-in data type some more.

Integers

In Python, and most programming languages, whole numbers (and their negatives) are called integers.

One thing that makes Python integers stand out is that they can be huge. For example, in Python you can easily compute:

```
In [3]: 2 ** 2000
```

```
Out [3]: 11481306952742545242328332011776819840223177020886952004776427368257662613923703138566594  
86316506269918445964638987462773447118960863055331425931356166653185391299891453122800006  
88779148240044871428926990063486244781615463646388363947317026040466353970904996558162398  
80894462960562331164953616422197033268134416890898445850560237948480791405890093477650042  
90027167066258305220081322362812917612678833172065989953964181270217798584040421598531832  
51540889433902091920554957783589672039160081957216630582755380425583726015528348786419432  
054508915275783882625175435528800822842770817965453762184851149029376
```

The numbers you work with likely aren't as big, but this feature of Python integers is available to you if you need it.

Floating-point numbers

As with whole numbers, decimal numbers in programming have a different name: *floating-point* numbers.¹

```
In [4]: x = 0.1  
type(x)
```

```
Out [4]: float
```

If you use integers and floating-point numbers together in a mathematical operation, the result will be a floating-point number:

```
In [5]: 10 + 1.0
```

```
Out [5]: 11.0
```

Something to be aware of when doing arithmetic with floating-point numbers is that results sometimes get rounded in strange ways. Consider the following example:

```
In [6]: 0.1 + 0.2
```

```
Out [6]: 0.3000000000000004
```

In Python, this type of rounding can become a problem in conditional statements:

```
In [7]: 0.1 + 0.2 == 0.3
```

```
Out [7]: False
```

This behavior is not unique to Python. It is due to the way decimal numbers get represented in your computer's memory. Excel faces the same issue as well, but it deals with this problem by rounding all decimal numbers to 15 digits for you — if you copy the decimal number above (which has 18 digits) in an Excel sheet, you will see it gets rounded down.

To deal with this issue in Python, you can round decimal numbers yourself (assuming you don't need this level of decimal precision) with the built-in `round` function:

```
In [8]: round(0.3000000000000004, 10)
```

1: The term *floating-point* refers to the way these numbers are represented inside your computer (i.e., their binary representation), which to some computer scientists resembles floating. Luckily, you can use floating point numbers even without sharing their sense of humor.

Out [8]: 0.3

The second number between parentheses above specifies how many digits from the original number to keep (in this case, the first ten digits after the decimal point).

However, when we start working with tables and entire columns of decimal numbers in Python, you will see that the decimal precision issue is less of a problem than it may seem right now. Even so, keep in mind that, in general, arithmetic with decimal numbers is approximate beyond the 15th decimal point in Python² — if you need that level of precision with your numbers, take a look at the `decimal` package that is part of the Python standard library.³

Booleans

Another simple built-in data type that is available in Python is the `boolean`⁴: `True` or `False` values. Comparison operations produce boolean values, which are particularly useful in conditional statements (i.e., `if-else` blocks)

In [9]:

```
if (10 < 20):
    print('hello, python for accounting!')
```

Out [9]: 'hello, python for accounting!'

Most often you do not work with booleans directly, they get produced by some other operation, but you can if you need to:

In [10]:

```
t = True
f = False
```

There are a few operators you can use with booleans in Python:

`and`, `or` and `not`:

In [11]:

```
x = 5
y = 2

(x < 10) and (y > 0)
```

Out [11]: True

In [12]: `t or (x == 5)`

Out [12]: True

In [13]: `(x == 5) and (not (y > 10))`

2: All software you use faces this issue and deals with it in different ways.

3: You can read more about high-precision arithmetic with decimal numbers in Python at docs.python.org/library/decimal.

4: Named after George Boole, who is the inventor of the true-false algebra that is at the core of digital electronics.

Out [13]: `True`

You can combine booleans — or comparison operations that produce boolean values — using these operators any way you want to. When you want to combine lots of comparison operations, using parentheses can prevent unwanted results.

Python's boolean values are just numbers in disguise: `True` is equal to 1 and `False` is equal to 0. You can use mathematical operators with them, as you would with numbers:

In [14]: `True == 1`

Out [14]: `True`

In [15]: `True + True + False`

Out [15]: `2`

This feature of boolean values is handy when working with tables in Python, as you will see in the next part of the book.

The None type

As the name suggests, the `None` type is used in Python to indicate the absence of a value. It has one value only: `None`.

In [16]: `type(None)`

Out [16]: `NoneType`

The `None` value is different from the `#N/A` value in Excel. In Excel, an `#N/A` value generally tells you that a formula couldn't run — it's a “cannot-compute” error message. In Python, `None` is used to indicate the absence of a value.

It may seem strange to have a value that indicates the absence of a value (which is a contradiction). Still, it's the most explicit way to indicate that a function or method does not produce a result (e.g., because it only displays something on the screen). For example, if you assign the output of the `print` function, which we've been using so far to display text, to a variable and print it, you'll see that the `print` functions doesn't produce a value:

In [17]: `value = print("hello, python for accounting!")` 1
`print(value)` 2

Out [17]: `hello, python for accounting!`
`None`

Something to keep in mind is that `None` doesn't show up in cell outputs if you don't explicitly print it. If you run the following cells in your notebook, you won't see any output:

```
In [18]: value = 1
no_value = None
```

```
In [19]: value
```

```
Out [19]: 1
```

```
In [20]: no_value
```

Strings

Even text values have a different name in programming: they're called *strings*. A string⁵ is a sequence of (one or more) characters that represents some text. Anything between quotes in Python is considered a string (both single and double quotes are valid):

```
In [21]: message = 'hello, python for accounting'
message = "hello, python for accounting"
```

You can also use three double quotes to define a multi-line string:

```
In [22]: message = """
hello,
python
for
accounting
"""
```

5: The term “string” seems to come from the printing industry, where a string was used to tie character blocks together as they were transported to the printing press after they were put together, but as with most terms in computer science, its origins are not very clear.

Python strings have many different methods you can use to manipulate them. For example, to make the first letter of each word in `message` uppercase, you can use:

```
In [23]: message = 'python for accounting'
message.title()
```

```
Out [23]: Python For Accounting
```

There are many other methods available on Python string — table 21.1 below lists a few of them.

Table 5.2: A few methods you can call on Python strings. A complete list of methods available for strings is documented at docs.python.org/library/stdtypes#string-methods.

Examples use `message = 'python for accounting'`.

Method name	Example	Output	Description
upper	<code>message.upper()</code>	PYTHON FOR ACCOUNTING	Changes all characters to uppercase.
lower	<code>message.lower()</code>	python for accounting	Changes all characters to lowercase.
capitalize	<code>message.capitalize()</code>	Python for accounting	Makes the first character uppercase.
title	<code>message.title()</code>	Python For Accounting	Makes first character in each word uppercase.
swapcase	<code>message.swapcase()</code>	PYTHON FOR ACCOUNTING	Swaps character case.

Clearing and formatting text values are recurrent operations when working with any data. We'll come back to handling strings in the following chapters, as I introduce more of the Python data analysis machinery.

Besides calling methods on strings, you can also combine different string variables with the `+` operator:

```
In [24]: first_name = "margaret"           1
         last_name = "hamilton"            2
                                          3
         first_name + " " + last_name      4
```

```
Out [24]: margaret hamilton
```

You can also call different methods on the strings you combine or mix-and-match methods and concatenation as you need to:

```
In [25]: first_name.capitalize() + " " + last_name.upper()
```

```
Out [25]: Margaret HAMILTON
```

```
In [26]: "Hello, " + (first_name + " " + last_name).title()
```

```
Out [26]: Hello, Margaret Hamilton
```

One thing you cannot do is combine numbers and strings using just the `+` operator:

```
In [27]: age = 83
          first_name + " " + last_name + " is " + age + " years old!"
```

```
Out [27]: TypeError                                     Traceback (most recent call last)
<ipython-input-3-9f830417fd96> in <module>
      1 print(first_name + " " + last_name + " you are " + age + " years old!")
      2
      3 TypeError: can only concatenate str (not "int") to str
```

The code above displays a `TypeError`. There are many kinds of errors in Python, don't be afraid of them; figuring out how to fix the errors you get as you're trying new things is what writing code is all about. The error above gets displayed because Python doesn't know how to combine the integer `83` with the other strings. If you look at the last line in the error message, you'll see why the error happened.

To fix this error, you can convert the `age` variable (or any other variable) to a string value by using the built-in `str` function:

```
In [28]: age = 83
          first_name + " " + last_name + " is " + str(age) + " years old!"
```

```
Out [28]: margaret hamilton is 83 years old!
```

Combining strings and other types of variables into a single piece of text is a typical operation. A simpler alternative to using the `+` operator with the `str` function is using an *f-string*:

```
In [29]: f'{first_name} {last_name} is {age} years old!'
```

```
Out [29]: margaret hamilton is 83 years old!
```

F-strings are regular Python strings that have an *f* character in front of them. Inside an f-string, you can write any variable name surrounded by curly brackets, and its value gets converted to a string. You can think of f-strings as templates that you fill in with different variable values.

The next section covers a few more built-in functions that you can use to convert Python variables from one type to another.

Associated built-in functions

Each of the simple data types I mentioned in this chapter (except the `NoneType`) has an associated built-in function you can use to convert variables to that type. You already used `str` to convert an integer to a string value, but you can use `int`, `float` or `bool` in the same way:

```
In [30]: int(10.5)
```

```
Out [30]: 10
```

```
In [31]: float(-10)
```

```
Out [31]: -10.0
```

```
In [32]: bool(10)
```

```
Out [32]: True
```

```
In [33]: bool(None)
```

```
Out [33]: False
```

```
In [34]: str(100)
```

```
Out [34]: '100'
```

Summary

This chapter introduced Python's built-in data types: integers, floating-point numbers or floats, booleans, strings, and the None type. Python's built-in data types are extended in various ways by the data analysis tools you'll use later in the book. Even as you start working with tables more, you'll still use these single value data types frequently, which is why I went over their features now. Next, let's take a look at how you can create collections of values in Python.

6

Python's built-in collections

Now that you're familiar with simple data types in Python, we can take a look at its built-in collection types, which are listed in table 6.1 below.

Table 6.1: Built-in collection data types in Python.

Name	Keyword	Example	Description
List	<code>list</code>	<code>x = [1, 4, 8]</code>	Sequence of ordered items.
Tuple	<code>tuple</code>	<code>x = (1, 4, 8)</code>	Sequence of ordered items that cannot be changed after you define it.
Set	<code>set</code>	<code>x = {1, 4, 8}</code>	Sequence of <i>unique</i> values in which the order of items does not matter (i.e., <code>{1, 4, 8}</code> is the same as <code>{4, 1, 8}</code>).
Dictionary	<code>dict</code>	<code>x = { 'first': 1, 'second': 4, 'third': 8 }</code>	Mapping of keys to values. As with <code>set</code> above, and unlike <code>list</code> or <code>tuple</code> , the order of items in a dictionary does not matter (i.e., you access items by key, rather than position).

Lists

Lists¹ are collections of items stored together under the same name. Instead of giving each item a separate name (i.e., assigning each item to a different variable), you can give the entire collection a name and access individual items through that common name. They are useful for working with related data: you can think of a table column as a list of values and of an entire table as a list of columns (i.e., a list of lists) — lists are surprisingly ubiquitous.

Lists in Python are defined using square brackets around a sequence of values separated by commas:

```
In [1]: prices = [15, 19, 7, 40] 1  
accounts = ['Assets', 'Equity', 'Income', 'Expenses'] 2
```

Lists usually contain more than one item² so it's a good idea to name them using a plural (e.g., `prices`, `accounts`).

1: You may be familiar with *arrays* from Excel's VBA. Lists in Python are similar to arrays, the difference being that, in Python, you do not have to declare how large your lists are up-front, or what type of values they contain.

Accessing items in a list

Lists are ordered sequences, which means you can access elements in a list by specifying their position (i.e., by their index). To access

2: You can mix data types in the same list if you need to (e.g., strings and integers), but I recommend holding items of the same type in your lists.

elements in a list, type the list variable name followed by the element index you want to access, surrounded by square brackets:

```
In [2]: accounts = ['Assets', 'Equity', 'Income', 'Expenses']  
        1  
        2  
        3  
accounts[0]
```

```
Out [2]: Assets
```

This gives you the first value in the `accounts` list. Notice that the first element in the list is at position 0, not 1, which is the default way items in a sequence get counted in most programming languages (this is called zero-based indexing). The second item has index 1, and so on until the last item in the list, which has index equal to the number of elements in the list minus one.

To find out how many elements there are in a list, you can use the built-in `len`³ function:

3: Short for *length*.

```
In [3]: len(accounts)
```

```
Out [3]: 4
```

Once you access an element from a list, you can use it as you would any other variable of the same type:

```
In [4]: accounts[0]
```

```
Out [4]: 'Assets'
```

```
In [5]: print("Account Name: " + accounts[0].upper())
```

```
Account Name: ASSETS
```

Something available in Python, but not many other programming languages, is using negative indexes to access list elements from the end of the list. The following example prints elements from `accounts` starting from the end of the list:

```
In [6]: accounts = ['Assets', 'Equity', 'Income', 'Expenses']  
        1  
        2  
        3  
        4  
        5  
        6  
  
print(accounts[-1])  
print(accounts[-2])  
print(accounts[-3])  
print(accounts[-4])
```

```
Expenses  
Income  
Equity  
Assets
```

Python also provides a powerful way to access multiple elements in a list through *slicing*. Given the `accounts` list above, you can get a sub-list containing just the first two elements by slicing it using:

```
In [7]: accounts = ['Assets', 'Equity', 'Income', 'Expenses']  
        1  
        2  
        3  
accounts[0:2]
```

```
Out [7]: ['Assets', 'Equity']
```

Slicing is similar to how you reference a range of cells in Excel. The exception is that when slicing a list in Python, the limit specified on the right side of the colon is non-inclusive, meaning that if you slice elements `[0:2]`, you will get a sub-list with the items at positions 0 and 1, but not the element at position 2 from the original list.

If you leave out the first number in the slice (i.e., the number before the colon), it defaults to 0, meaning you will get everything in the list up to the number on the right of the colon:

```
In [8]: accounts[:2]
```

```
Out [8]: ['Assets', 'Equity']
```

Similarly, you can omit the number after the colon:

```
In [9]: accounts[1:]
```

```
Out [9]: ['Equity', 'Income', 'Expenses']
```

When you omit the position on the right of the colon, you get all the items in the list after (and including) the position on the left of the colon.

You can even use negative indexes in the same way. For example, to get the last two elements in the `accounts` list:

```
In [10]: accounts = ['Assets', 'Equity', 'Income', 'Expenses']      1  
          accounts[-2:]           2  
                                3
```

```
Out [10]: ['Income', 'Expenses']
```

Slicing always gives you a new list, so you can assign a slice of your original list to another variable, without worrying about the variables-as-pointers issue I mentioned in chapter 4:

```
In [11]: a = [1, 2, 3]  
b = a[:2]  
  
b.append(10)  
  
print(a)  
print(b)
```

```
Out [11]: [1, 2, 3]  
[1, 2, 10]
```

Adding or removing items from a list

You will often need to modify the contents of a list by adding or removing items. To add a new element at the end of the `accounts` list you've been working with so far:

```
In [12]: accounts = ['Assets', 'Equity', 'Income', 'Expenses']      1
          accounts.append('Liabilities')                            2
                                                               3
          accounts                                         4
```

```
Out [12]: ['Assets', 'Equity', 'Income', 'Expenses', 'Liabilities']
```

You have already seen the `append` method in action earlier. It adds a new element at the end of a list. However, if you need to insert an element at a certain position in your list, you can use the `insert` method, which allows you to specify the index at which you want to insert the new element:

```
In [13]: accounts = ['Assets', 'Equity', 'Income', 'Expenses']      1
          accounts.insert(1, 'Liabilities')                           2
                                                               3
          accounts                                         4
```

```
Out [13]: ['Assets', 'Liabilities', 'Equity', 'Income', 'Expenses']
```

Here, you inserted the new item at position 1 in the list and all elements to the right of it are shifted by one position.

Removing elements from lists follows a similar logic:

```
In [14]: accounts = ['Assets', 'Equity', 'Income', 'Expenses']      1
          accounts.remove('Income')                                2
                                                               3
          accounts                                         4
```

```
Out [14]: ['Assets', 'Equity', 'Expenses']
```

The `remove` method removes only the *first* occurrence of an item, so if you have multiple items in a list with the same value (e.g., two 'Income' accounts in the `accounts` list), you will have to loop through the list and remove each of them — more on loops in the following chapters.

When you don't know which element you want to remove from your list, but you know its position in the list, you can call the `pop` method on your list variable and specify the index of the item you want to remove:

```
In [15]: accounts = ['Assets', 'Equity', 'Income', 'Expenses']      1
          accounts.pop(2)                                       2
                                                               3
          accounts                                         4
```

```
Out [15]: ['Assets', 'Equity', 'Expenses']
```

By default, if you do not specify an item index, the `pop` method removes the *last item* in the list. It also gives you a reference to the item you removed (i.e., it returns the removed item) so you can keep working with it after it is removed from the list.

```
In [16]: accounts = ['Assets', 'Equity', 'Income', 'Expenses']      1
removed = accounts.pop()                                         2
                                                               3
removed                                                       4
```

```
Out [16]: Expenses
```

```
In [17]: accounts
```

```
Out [17]: ['Assets', 'Equity', 'Income']
```

Modifying items in a list

Sometimes you don't want to remove or add elements to a list, but change the ones it contains. For instance, to change the '`Income`' account to '`Revenue`', you can assign a new value to the item at position 2 in the `accounts` list (remember that indices in a list start with 0):

```
In [18]: accounts = ['Assets', 'Equity', 'Income', 'Expenses']      1
accounts[2] = 'Revenue'                                           2
                                                               3
accounts                                                       4
```

```
Out [18]: ['Assets', 'Equity', 'Revenue', 'Expenses']
```

If you don't know the position of the item you want to change, but you know its value, you can use the `index` method on the `accounts` variable to find its index first:

```
In [19]: accounts = ['Assets', 'Equity', 'Income', 'Expenses']      1
accounts.index('Income')                                         2
```

```
Out [19]: 2
```

Then you can use the index value to change the element at that position in the list:

```
In [20]: accounts = ['Assets', 'Equity', 'Income', 'Expenses']      1
accounts[accounts.index('Income')] = 'Revenue'                      2
                                                               3
accounts                                                       4
```

```
Out [20]: ['Assets', 'Equity', 'Revenue', 'Expenses']
```

Organizing a list

One of the most common operations performed on lists is sorting them. To sort a list, you can use the `sort` method on the list variable you want to re-order:

```
In [21]: accounts = ['Income', 'Assets', 'Expenses', 'Equity']      1
accounts.sort()                                              2
                                                               3
accounts                                              4
```

```
Out [21]: ['Assets', 'Equity', 'Expenses', 'Income']
```

The `sort` method sorts the list by comparing list items to each other: string items are sorted alphabetically, in ascending order (i.e., from A to z); numbers are sorted from low to high. If you want to sort the list in descending order (from high to low), you can pass a `reverse=True` argument⁴ to the `sort` method:

```
In [22]: numbers = [9, 1, 16, 4, 36, 25]      1
numbers.sort(reverse=True)                  2
                                                               3
accounts                                              4
```

```
Out [22]: [36, 25, 16, 9, 4, 1]
```

Similarly, you can use the `reverse` method to — you guessed it — reverse the order of items in a list:

```
In [23]: accounts = ['Assets', 'Equity', 'Income', 'Expenses']      1
accounts.reverse()                                              2
                                                               3
accounts                                              4
```

```
Out [23]: ['Expenses', 'Income', 'Equity', 'Assets']
```

Note that both these methods modify the original list you start with (i.e., they modify the `accounts` list in-place), so after you use these methods, you lose the initial ordering of the list. If the initial order is important, and you need to keep track of it, you can use the built-in `sorted` function instead. The `sorted` function gives you a sorted copy of the original list to work with, and does not modify the original list — similarly, you can use the `reversed` function to get a new list with items in reverse order without modifying the original list:

```
In [24]: accounts = ['Income', 'Assets', 'Expenses', 'Equity']      1
sorted_accounts = sorted(accounts)                                2
reversed_accounts = reversed(accounts)                            3
                                                               4
print(accounts)                                              5
print(sorted_accounts)                                         6
print(reversed_accounts)                                       7
```

4: More on functions and arguments
in the following chapters.

```
['Income', 'Assets', 'Expenses', 'Equity']
['Assets', 'Equity', 'Expenses', 'Income']
['Equity', 'Expenses', 'Assets', 'Income']
```

Overthinking: Tuples and sets

Python tuples are very similar to lists, the only difference being that after you define a tuple, you can't change its contents (i.e., you cannot add, remove, or change its items in any way). Tuples are useful when the relative order of items in your collection is important, and you don't want to modify it accidentally (e.g., by calling `sort` on it). You define tuples the same way you define lists in Python, the difference being that you use parentheses rather than square brackets:

```
In [25]: values = (1, 4, 5, 2)
          1
          2
          values
          3
Out [25]: (1, 4, 5, 2)
```

As with lists, you can access elements in a tuple using their position:

```
In [26]: values[0]
Out [26]: 1
```

But you can't assign new values to tuple items once the tuple is defined:

```
In [27]: values[0] = 9
          TypeError: Traceback (most recent call last)
          <ipython-input-50-e353267790ad> in <module>
          ----> 1 values[0] = 9
          TypeError: 'tuple' object does not support item assignment
```

Because the name “*tuple*” starts with the sound *two*, I used to think tuples can only hold two items, but they can hold as many items as you need them to.

Sets are sequences of unique items — while Python lists can contain duplicate items, sets can only have unique items. Sets are a bit trickier to work with (i.e., you can't access elements in a set using their position), but they're useful when you need to perform set arithmetic (i.e., union, intersection, or difference of items).

You define sets using curly brackets, instead of square ones:

```
In [28]: accounts = {'Assets', 'Liabilities', 'Revenue', 'Assets'}
          accounts
```

```
Out [28]: {'Assets', 'Liabilities', 'Revenue'}
```

Notice that even if you try to include duplicates (the 'Assets' item is duplicated above), they will be discarded from the set.

If you need to perform set arithmetic, you can use the `union`, `difference` or `intersection` methods, the output of which will be another set:

```
In [29]: accounts = {'Assets', 'Liabilities', 'Revenue'}      1
ledgers = {'Income', 'Assets', 'Equity'}                      2
                                                       3
accounts.union(ledgers)                                     4
```

```
Out [29]: {'Assets', 'Equity', 'Income', 'Liabilities', 'Revenue'}
```

```
In [30]: accounts.intersection(ledgers)
```

```
Out [30]: {'Assets'}
```

```
In [31]: accounts.difference(ledgers)
```

```
Out [31]: {'Liabilities', 'Revenue'}
```

Sets are handy when you want to remove duplicates from a list, or when you want to count the number of unique items in a Python list:

```
In [32]: values = [1, 2, 2, 3, 3, 3]      1
                                                       2
set(values)                                         3
```

```
Out [32]: [1, 2, 3]
```

Even though you can't access items in a set by their index, you can easily convert lists to sets and back to lists (which are much easier to work with):

```
In [33]: values = [1, 2, 2, 3, 3, 3]      1
unique_values = list(set(values))          2
                                                       3
unique_values                               4
```

```
Out [33]: [1, 2, 3]
```

Index out of range errors

When working with lists or tuples, a common error you'll see is `IndexError: list index out of range`. A code example that triggers this error is shown below — if you want to access the last element in the `accounts` list, you might write something like this:

```
In [34]: accounts = ['Assets', 'Equity', 'Income', 'Expenses']
print(accounts[4])
```

```
IndexError                                     Traceback (most recent call last)
<ipython-input-68-5d6e698433b9> in <module>
      1 print(accounts[4])
----> 1 print(accounts[4])

IndexError: list index out of range
```

However, this doesn't work. Python attempts to print the item at position 4 in the list, but because list positions are numbered starting with 0, the `accounts` list doesn't have an item at the fourth position (i.e., the fourth and last item in the list is at position 3). Another common trigger for this error is when you try to access elements in an empty list, so if you see this error, check that your lists contain any items:

```
In [35]: accounts = []
accounts[-1]                                1
                                2
```

```
-----
```

```
IndexError                                     Traceback (most recent call last)
<ipython-input-69-fa8b578b7c9b> in <module>
      1 accounts[-1]
----> 1 accounts[-1]

IndexError: list index out of range
```

Getting an `IndexError` is typical when working with lists because of the off-by-one numbering of elements that Python lists use. When you see this error, remember that everyone who has worked with Python has seen this error message many times over, and inspect your list to see how many elements it contains. You might find that your list looks different from what you thought it did.

Dictionaries

Dictionaries are another useful Python data structure that can store multiple items. Just like real-world dictionaries that map words to their description (or translation), Python dictionaries map a *key* to a *value*. Dictionary keys are often string values, but they can be numbers or boolean values as well; dictionary values can be anything, including lists or other dictionaries.

You can create Python dictionaries by specifying key-value pairs separated by commas and wrapped around curly brackets:

```
In [36]: account = {'name': 'Assets', 'value': 12000}          1
                                2
account                         3
```

```
Out [36]: {'name': 'Assets', 'value': 12000}
```

A dictionary can have as many key-value pairs as you need it to, but each key must be unique (i.e., you can't have entries with the same key in the same dictionary).

To access values from a dictionary, type the name of the dictionary variable and the key associated with the value you want to access, surrounded by square brackets:

```
In [37]: account['name']
```

1

```
Out [37]: Assets
```

```
In [38]: account['value']
```

1

```
Out [38]: 12000
```

If you try to access a key-value pair that doesn't exist in the dictionary, you will get prompted with a `KeyError`:

```
In [39]: account['updated_at']
```

1

```
Out [39]: KeyError
```

Traceback (most recent call last)

```
<ipython-input-90-8fb099c272cb> in <module>
----> 1 account['updated_at']
```

```
KeyError: 'updated_at'
```

To add a key-value pair to the `account` dictionary defined above, you can assign a value to a new key in the dictionary:

```
In [40]: account['created_at'] = '09/03/2020'
```

1

```
account
```

2

```
Out [40]: {'name': 'Assets', 'value': 12000, 'created_at': '09/03/2020'}
```

3

Or to remove a key and its value, you can use the `pop` method and specify the key you want to remove as its argument:⁵

5: Similar to using `pop` with a list.

```
In [41]: account.pop('created_at')
```

1

```
account
```

2

```
Out [41]: {'name': 'Assets', 'value': 12000}
```

3

To modify a dictionary in-place, you can assign new values to any of its existing keys:

```
In [42]: account = {'name': 'Assets', 'value': 12000}
```

1

```
account['name'] = 'Liabilities'
```

2

```
account['value'] = -19300
```

3

```
account
```

4

```
Out [42]: {'name': 'Liabilities', 'value': -19300}
```

5

You can define an empty dictionary by using an empty set of curly brackets and then adding key-value pairs as you need them:

```
In [43]: account = {}

account['name'] = 'Income'
account['value'] = 20100
account['created_at'] = '10/03/2020'

account
```

1
2
3
4
5
6
7

```
Out [43]: {'name': 'Income', 'value': 20100, 'created_at': '10/03/2020'}
```

Remembering all the keys in a dictionary can become a problem when you have a lot of them. You can inspect a dictionary's keys or values with the `keys` or `values` methods:

```
In [44]: account.keys()

Out [44]: dict_keys(['name', 'value', 'created_at'])

In [45]: account.values()

Out [45]: dict_values(['Income', 20100, '10/03/2020'])
```

Unlike dictionary keys, which can only be strings, numbers, or booleans⁶ dictionary values can be anything, including lists or other dictionaries:

```
In [46]: account = {}

account['name'] = 'Income'
account['value'] = 20100
account['created_at'] = '10/03/2020'

account['info'] = {'title': 'Main income account', 'description': 'Account description'}
account['ins'] = [12000, 2300, 5800]

print(account)

{'name': 'Income', 'value': 20100, 'created_at': '10/03/2020', 'info': {'title': 'Main income account', 'description': 'Account description'}, 'ins': [12000, 2300, 5800]}
```

1
2
3
4
5
6
7
8
9
10

You can access values in a nested dictionary by using multiple sets of square brackets and key names:

```
In [47]: account['info']['title']

Out [47]: Main income account
```

Much of Python's power comes from the fact that you can easily manipulate its building blocks into data structures that work for you and your use-cases. Dictionaries can be powerful tools, but as with most programming concepts, it takes practice to build intuition around when and how to use them effectively. We'll come back to Python dictionaries when we start working with Excel spreadsheets, mostly to replace values in a column (by specifying mappings from old values to new ones), or to rename table columns.

6: To be more precise, any *immutable* value can be used as a key in a Python dictionary. There are other immutable types besides strings, numbers or booleans, however, these three types are most often used as dictionary keys.

Membership operators

Python has two keyword operators you can use to check whether a certain value is in a sequence: `in` and `not in`. These operators work with all the collections we have covered so far: lists, tuples, sets, or dictionaries. For example, you can use them to check whether a value is in a list:

```
In [48]: accounts = ['Assets', 'Equity', 'Income', 'Expenses']      1
          'Income' in accounts                                     2
          'Revenue' not in accounts                                3

Out [48]: True

In [49]: 'Revenue' not in accounts

Out [49]: True
```

With Python dictionaries, these operators can be used to check whether a key is already in the dictionary:

```
In [50]: account = {'name': 'Income', 'value': 20100, 'created_at': '10/03/2020'}    1
          'name' in account                                         2
          'updated_at' in account                                  3

Out [50]: True

In [51]: 'updated_at' in account

Out [51]: False
```

You can use these operators together with boolean operators (i.e., `and` and `or`) and comparison operators to construct complex conditional statements about item membership:

```
In [52]: account = {'name': 'Income', 'value': 20100, 'created_at': '10/03/2020'}    1
          'value' in account and account['value'] > 10000               2
          'value' in account and account['value'] > 10000                3

Out [52]: True
```

Associated built-in functions

Just as simple data types can be converted from one type to another using their associated built-in functions, you can convert collection types using the `list`, `set`, `tuple` or `dict` functions. You've already seen an example earlier in this chapter of converting a list to a set to remove duplicate items:

```
In [53]: values = [1, 2, 2, 3, 3, 3]
          set(values)

Out [53]: {1, 2, 3}

In [54]: tuple(values)
```

```
Out[54]: (1, 2, 2, 3, 3, 3)
```

```
In [55]: list(set(values))
```

```
Out[55]: [1, 2, 3]
```

When you convert a dictionary to a list, using the `list` function, you will only get its keys in the list (i.e., without the associated values).

```
In [56]: account = {'name': 'Income', 'value': 20100, 'created_at': '10/03/2020'}
```

1
2
3

```
list(account)
```

```
Out[56]: ['name', 'value', 'created_at']
```

Conversely, you can convert a list of sequences (i.e., a list of lists or a list of tuples) to a dictionary using the `dict` function:

```
In [57]: values = [('name', 'Income'), ('value', 20100)]
```

1
2
3

```
dict(values)
```

```
Out[57]: {'name': 'Income', 'value': 20100}
```

Summary

This chapter introduced Python's built-in collections: lists, tuples, sets, and dictionaries. These collections are useful when working with related data — you'll use them often with Python's data analysis tools. Let's take a look at Python's loops and `if-else` statements next.

7

Control flow

In this chapter, we'll take a closer look at `if-else` statements and loops — both of which you've already seen in the example at the beginning of chapter 3.

If-else statements

One of the most well-known concepts in computer programming is the `if-else` statement — also known as a conditional statement. Conditional statements in Python use the `if`, `elif` and `else` keywords, like in the following example:

```
In [1]: x = 42
1
2
3
4
5
6
7
8
9
10
11
if x > 0:
    print('x is positive')
elif x < 0:
    print('x is negative')
elif x == 0:
    print('x is zero')
    print('x is neither negative nor positive')
else:
    print('not sure what x is')

x is positive
```

The code above prints a message based on `x`'s value. Notice the use of indentation and colons after each branch of the `if-else` statement. There can be zero or more `elif`¹ branches, and the final `else` is optional (i.e., you can have just a single `if` in your conditional statement).

1: Short for *else if*.

```
In [2]: if x > 0:
    print('x is positive')
```

```
Out [2]: x is positive
```

As you start working with indented blocks of code more (in `if` statements or elsewhere), a common error message you'll come across is `IndentationError`:

```
In [3]: x = 42
1
2
3
4
if x < 0:
    print('x is negative')
```

```
File "<ipython-input-1-ce78d3683b44>", line 2
    print("x is negative")
    ^
IndentationError: expected an indented block
```

IndentationError messages tell you that somewhere in your code, indentation is not following Python's syntax rules. Notice in the example above that the error message is pointing you to the line of code missing the indent — here, the line immediately after the `if` statement needs to be indented.

Indentation is just a rule of Python, much like the English language has rules about what punctuation symbols you can use at the end of a sentence. Unfortunately, while you can have exceptions in the English language, computer code is less flexible, so you need to make sure you follow Python's rules strictly.

Loops

Loops are the part of computer programming that automates repetitive tasks. In Python, you can use `for` loops to repeat a part of your code as many times as you need to:

```
In [4]: for i in [1, 2, 3, 4, 5]:
    if i % 2 == 0:
        print(f'{i} is even')
    else:
        print(f'{i} is odd')
```

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
```

```
In [5]: accounts = ['Assets', 'Equity', 'Income', 'Expenses']

for account in accounts:
    print(account)
```

```
Assets
Equity
Income
Expenses
```

To start a for loop, after the `for` keyword, you specify a variable name², the `in` keyword and a sequence you want to loop over. You can even read the second example above as “*For every account in the accounts list, print the account*”. Python code is not that far from plain English.

2: Which can be any name you want, as long as it does not start with a number or has spaces in it.

Notice that, as with `if` statements, the indentation and colon after the `for` keyword are both required. If you need to do more work on the same iteration of the loop, you can indent multiple lines right under the `for` keyword:

```
In [6]: company_accounts = [
    {'name': 'Assets', 'value': 12000},
    {'name': 'Income', 'value': 20100},
    {'name': 'Expenses', 'value': -10300},
]

for current_account in company_accounts:
    print("Account name: " + current_account['name'])
    print("Account value: " + str(current_account['value']))
    print()

print("Printed all accounts!")

1
2
3
4
5
6
7
8
9
10
11
12
```

```
Account name: Assets
Account value: 12000

Account name: Income
Account value: 20100

Account name: Expenses
Account value: -10300

Printed all accounts!
```

Let's unpack this example some more:

- ▶ Line 1 defines the `company_accounts` list, which contains several Python dictionaries representing individual accounts. Each dictionary in the list has two keys: `name` and `value`. Remember that lists can hold any data, not just strings or numbers.
- ▶ Line 7 defines the `for` loop. In this case, the *iteration variable* is called `current_account` — you can use any name for this variable, as long as it's a valid variable name in Python (i.e., it doesn't start with a number and doesn't contain any invalid characters, such as spaces or emojis).
- ▶ Line 9 uses the `str` function to transform a numerical value into a string so that you can concatenate it to another string.
- ▶ Line 10 prints an empty line after each account. Notice that lines 8, 9, 10 are all indented at the same level — this tells Python those lines are part of the `for` loop and should be executed at each step in the iteration.
- ▶ Line 11 prints a completion message after the `for` loop ends. This `print` instruction runs only after the `for` loop has completed going through the `company_accounts` list.

Overthinking: while loops

Besides `for` loops, Python also has `while` loops. This type of loop is useful when you don't know up-front how many times you need to repeat an action (i.e., they run until a *condition* is met).

In [7]:

```
counter = 0

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

Out [7]:

```
0
1
2
3
4
```

The `while` loop checks the conditional statement in each iteration. When it becomes `False`, the loop stops — something needs to happen inside the looping code that makes the condition false, or your loop will keep on running forever. As with the `for` loop before, indentation and the colon are required.

You won't use `while` loops anywhere in this book, but you might come across them elsewhere, which is why I briefly mentioned them here.

Overthinking: break

Once started, loops keep running until they finish all their work (i.e., while loops keep running until the condition they check becomes `False`, for loops until they run out of items to go over). However, you will sometimes want to end a loop early. In those cases, you can use the `break` keyword to stop a loop:

In [8]:

```
accounts = ['Assets', 'Equity', 'Income', 'Expenses']

for account in accounts:
    if account.startswith('I'):
        print('Account name starts with I: ' + account)
        break

    print('Account name does not start with I: ' + account)
```

```
Account name does not start with I: Assets
Account name does not start with I: Equity
Account name starts with I: Income
```

In this example, you loop through each item in the `accounts` list and check if the item starts with the letter '`I`'. If it does not, you print a message that says that. If it *does* start with '`I`', you print

the account and then run the `break` command, which ends the `for` loop early, before its last iteration (i.e., the one before the 'Expenses' item in the `accounts` list).

You can use the `break` keyword to stop your computer from doing unnecessary work, once you know the result you wanted was computed. The `break` keyword can be used with both `for` and `while` loops.

List comprehensions

One of the most loved features of Python is the *list comprehension*. It allows you to quickly create lists of items — which sounds less impressive than it is.

A list comprehension is an expression followed by a `for` statement, and its result is always a new list. You can use list comprehensions to apply transformations on the items in an existing list, or use them with an `if` statement to filter lists. For example, you can filter a list of numbers and get its negative values in a separate list using a list comprehension:

```
In [9]: numbers = [-11, 9, 3, -2, 0, -3, 18]
[n for n in numbers if n < 0]
```

```
Out [9]: [-11, -2, -3]
```

You can also apply transformations to the values you want to put in a list comprehension (remember that the `**` operator squares numbers):

```
In [10]: squares = [i**2 for i in [2, 3, 4, 5]]
squares
```

```
Out [10]: [4, 9, 16, 25]
```

```
In [11]: fresh_fruit = ['apple', 'banana', 'blueberry', 'kiwi', 'orange']
[fruit.capitalize() for fruit in fresh_fruit]
```

Or you can include an `if-else` statement in there to filter an existing list:

```
In [12]: [fruit for fruit in fresh_fruit if fruit.startswith('b')]
```

```
Out[12]: ['banana', 'blueberry']
```

I find it easier to write list comprehensions when I first type out the `for` part and surround it with square brackets. After you have the `for` part typed out, it's usually easier to see how to expand the list comprehensions with either an item transformation or an `if-else` statement (or both).

Summary

This chapter went over `if-else` statements, loops, and list comprehensions. Loops let you repeat parts of your code, while `if-else` statements let you choose which parts of your code run and which don't. These constructs are often used in custom functions to define bespoke operations on data — let's look at how you create a custom function in Python next.

8

Functions

Programming is powerful because you can write a sequence of actions once and repeat it as many times as you need to. You do that with loops and *functions*.

Functions are named blocks of code designed to do one specific thing (e.g., compute a sum, read a file, draw a plot). Whenever you need to run the same code multiple times, you can define a custom function and *call* that function by typing its name.

In general, functions do one of two things: print some information on the screen (e.g., a table, a plot, or just some text, as in the examples you've seen so far) or process some data and *return* one or more values. Sometimes functions do both — regardless of what a function does, it is defined the same way in Python.

Defining functions

To define a function in Python, you need to use the `def` keyword, followed by a *function name*, a set of parentheses, and a colon:

```
In [1]: def print_book_info():  
        print('Python for Accounting')
```

The colon and line indentation you're already familiar with from `if` and `for` statements are required for function definitions too.

To run the function defined above (i.e., *call* the function), you need to type its name followed by an empty set of parentheses; you can call this function as many times as you need to:

```
In [2]: print_book_info()  
print_book_info()  
print_book_info()
```

```
Python for Accounting  
Python for Accounting  
Python for Accounting
```

Parameters and arguments

Most often, you will want to pass some values to a function when calling it. The values you pass can then be used by the function code — for instance, to pass a book title to `print_book_info`, you can re-define the function above and add a *parameter* to its definition:

```
In [3]: def print_book_info(title):
    print("Book title: " + title)

print_book_info("Python for Accounting")
```

Book title: Python for Accounting

Parameters and arguments are terms used to describe two sides of the same coin. *Parameters* are the variable names your function works with inside its code (their names are declared inside the parentheses when you define the function), whereas *arguments* are the values associated with these variables when you call the function.

In the example above, `title` is the parameter and '`Python for Accounting`' is the argument used when calling the function (i.e., the value associated with `title` when calling it).

Most functions will need more than one parameter to be useful. You can use *positional arguments* or *keyword arguments* to call functions with multiple parameters; let's take a quick look at both.

Keyword arguments

When using keyword arguments, you specify, by name, which parameters get which values when calling a function:

```
In [4]: def print_book_info(title, subtitle):
    print("Book title: " + title)
    print("Book subtitle: " + subtitle)

print_book_info(title="Python for Accounting", subtitle="A modern guide to Python!")
```

Book title: Python for Accounting
Book subtitle: A modern guide to Python!

On line 5, when calling the `print_book_info` function, you name each parameter explicitly and assign it a value. If you want to use keyword arguments when you call a function, you have to use the same parameter names you used when you defined the function. The order of keyword arguments doesn't matter because you are explicit about which value is associated with which parameter — for example, the following calls are identical:

```
In [5]: print_book_info(title="Python for Accounting", subtitle="A modern guide to Python!")
print_book_info(subtitle="A modern guide to Python!", title="Python for Accounting")
```

Book title: Python for Accounting
Book subtitle: A modern guide to Python!
Book title: Python for Accounting
Book subtitle: A modern guide to Python!

Notice the `+` operator used to concatenate string values.

Positional arguments

If you want to type less, you can use positional instead of keyword arguments when calling functions. For example, you can call the `print_book_info` function above with:

```
In [6]: def print_book_info(title, subtitle):
    print("Book title: " + title)
    print("Book subtitle: " + subtitle)

print_book_info("Python for Accounting", "A modern guide to Python!")
```

1
2
3
4
5

```
Book title: Python for Accounting
Book subtitle: A modern guide to Python!
```

Notice the order of arguments when you call the function is the same order in which you specified function parameters when you defined it. These arguments are called *positional* because the order in which they are specified (i.e., their position in the sequence of arguments) is how Python links them to their respective parameter names inside the function code. If you want your function to work as you designed it, you need to be mindful of the order you pass arguments to it.

You can use as many positional arguments as you need to with your Python functions. However, if you have lots of parameters in your function, it becomes difficult to remember their order when you call the function. In those cases, using keyword arguments can be much easier.

Default values

You can assign default parameter values when defining a function. If a parameter has a default value, it can be omitted when calling the function, in which case the code inside the function will use the default value for that parameter:

```
In [7]: def print_book_info(title, subtitle="Default subtitle"):
    print('Book title: ' + title)
    print('Book subtitle: ' + subtitle)

my_book_title = "Python for Accounting"
print_book_info(my_book_title)
```

1
2
3
4
5
6

```
Book title: Python for Accounting
Book subtitle: Default subtitle
```

Parameters that have default values need to appear at the end of the parameter list in the function definition — that is, after all the other parameters that don't have default values (you can still have as many as you want). For example, the code below is not valid:

```
In [8]: def print_book_info(title='Python for Accounting', subtitle):
         print('Book title: ' + title + ', book subtitle: ' + subtitle) 1
                                                               2

File '<ipython-input-22-d183596b536c>', line 1
    def print_book_info(title='Python for Accounting', subtitle):
    ^
SyntaxError: non-default argument follows default argument
```

In this case, the `SyntaxError` message informs you that the argument that does not have a default value needs to come before the one that does have a default value.

Because functions are so flexible in Python, you have many equivalent ways of calling them. For the `print_book_info` function defined below, all of the following function calls are equivalent and print the same text:

```
In [9]: def print_book_info(title, subtitle='A modern guide to Python!'):
         print('Book title: ' + title + ', book subtitle: ' + subtitle) 1
                                                               2

print_book_info("Python for Accounting") 3
print_book_info("Python for Accounting", "A modern guide to Python!") 4
print_book_info("Python for Accounting", subtitle="A modern guide to Python!") 5
print_book_info(title="Python for Accounting", subtitle="A modern guide to Python!") 6
print_book_info(subtitle="A modern guide to Python!", title="Python for Accounting") 7
                                                               8
```

Which calling style you use is a matter of preference — use whichever one you find easiest to read, understand, and remember (and which keeps you from having to type a lot).

Argument errors

When working with functions, you'll often see an error message related to *missing required arguments*. For the `print_book_info` function above, calling it without passing any arguments leads to the following error:

```
In [10]: print_book_info() 1

TypeError                                 Traceback (most recent call last)
<ipython-input-29-857f10f150e2> in <module>
      1 print_book_info()

TypeError: print_book_info() missing 1 required positional argument: 'title'
```

If your function is defined with parameters (and they don't all have default values assigned), you need to pass values for those parameters when calling it; otherwise, you get the error message above. Python uses parameter names to indicate which arguments are missing in the error messages it shows you — which is a good reason to keep your parameter names descriptive because it will help you understand Python's error messages. If you provide too

many arguments to a function call, you'll see a similar message, indicating which arguments are in excess.

Return values

Functions don't always print something on the screen, like the `print_book_info` function you saw earlier. Indeed, most often, functions process some data and *return* one or more values. Return values are computed within the function code and, on the last statement of the function definition, are sent back to the line of code that called the function, using the `return` keyword.

Let's modify the `print_book_info` function to `return` a string containing book details instead of printing book details directly. We'll also give the function a new name because it no longer prints anything but instead returns a value:

```
In [11]: def get_book_info(title, subtitle):
    return "Book title: " + title + ", book subtitle: " + subtitle

get_book_info("Python for Accounting", "A modern guide to Python!")
```

```
Out [11]: 'Book title: Python for Accounting, book subtitle: A modern guide to Python!'
```

Even though you changed the function to `return` a value, you can still see its output directly under the code cell above. This is because Jupyter notebooks allow you to inspect variables and function outputs interactively (which is the main reason we're using them). Notice the quotes around the message above, which tells you that calling `get_book_info` returned a string value (i.e., when using `print`, you don't see these quotes).

If you want to keep working with a function's return value, you can assign it to a variable:

```
In [12]: book_info = get_book_info("Python for Accounting", "A modern guide to Python!")

print(book_info)
```

```
Book title: Python for Accounting, book subtitle: A modern guide to Python!
```

You can also return more than one value from your functions:

```
In [13]: def get_book_info(title, subtitle):
    return title, subtitle
```

If you inspect the return value of this function, you will see that Python wraps the two values it returns inside a tuple:

```
In [14]: get_book_info('Python for Accounting', 'A modern guide to Python!')
```

```
Out[14]: ('Python for Accounting', 'A modern guide to Python!')
```

To assign return values to separate variables when you call the function above, you can *unpack* them using:

```
In [15]: book_title, book_subtitle = get_book_info(  
    'Python for Accounting',  
    'A modern guide to Python!'  
)  
  
print(book_title)  
print(book_subtitle)
```

```
Python for Accounting  
A modern guide to Python!
```

You'll see this style of unpacking multiple return values when making plots with Python code later in the book.

Summary

This chapter showed you how to define custom Python functions. When we start working with tables, you'll see that you can easily apply custom functions to all the values in the rows or columns of a table. You can also keep your custom functions in a separate file outside your notebook if you want to share functions between notebooks — let's see how you do that next.

Modules, packages, and libraries

Modules, packages, and libraries are different ways to organize Python code into separate files and folders.¹ They're useful when you want to share functions or variables across notebooks or with other Python users. To access code from modules, packages, or libraries inside your notebooks, you need to `import` them:

```
In [1]: import pandas as pd
```

This line of code gives you access to functions from the `pandas` library, which is a collection of Python files stored somewhere on your computer (i.e., you installed it with Anaconda earlier).

There's nothing complicated about modules or packages; they're just Python files (i.e., files that end with a `.py` extension). Let's take a closer look at how you create them.

Modules and packages

Modules are text files that have some Python code in them. The best way to understand Python modules is to create one by yourself: in JupyterLab, under the “*File*” menu, open the “*New*” sub-menu and then select “*Text File*”. This should create a new file in your workspace folder called “*untitled.txt*”. To turn this empty file into a Python module, first, you need to rename it: right-click it in the file navigator and select “*Rename*”. Change its name to “*my_module.py*” (make sure the file extension is changed to `.py` instead of `.txt`).²

The second step in turning this file into a Python module is adding some code to it. Open “*my_module.py*” in JupyterLab and define a simple function that adds two numbers:

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

Save the file and close it. In this chapter's Jupyter notebook (or any other notebook that's in the same folder as “*my_module.py*”) you can import the module using its filename:

```
In [3]: import my_module
```

You now have access to the `add_numbers` function in your notebook. To call it, you can use the following code:

```
In [4]: my_module.add_numbers(2, 5)
```

1: They enable Python's vast ecosystem of tools, which is driven by developers sharing their code files online.

2: Any filename works, as long as it doesn't have spaces, but the file extension must be `.py`.

Out [4]:

7

You just created your own Python module. The main takeaway from this simple example is that modules are just text files which contain re-usable Python code. Whenever you use the `import` keyword — and we'll use it often in the following chapters — remember that all it does is give you access to some code written in a different file.

Note that whenever you change code inside a module file (by adding a new function, for example), you have to restart your notebook kernel and run the `import` statement again to get access to your module's latest changes. This is related to how Jupyter notebooks work more than it is to Python modules. As an example, let's extend `my_module` with a `subtract_numbers` function — edit your `my_module.py` file to contain the following functions:

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

def subtract_numbers(a, b):
    return a - b
```

Save "`my_module.py`" and go back to your notebook. In the menu bar at the top click "*Kernel*" and select "*Restart Kernel*" to restart the notebook kernel. You will be prompted with a confirmation dialog: click "*Restart*" (you can also restart the kernel if you are in *Command* mode by pressing the `Ø` key twice, in quick succession).

After the kernel restarts, run the `import my_module` cell again. You should be able to run the following code now:

```
In [6]: import my_module

my_module.subtract_numbers(2, 5)
```

Out [6]:

-3

There are different ways of using the `import` keyword to make your coding experience with modules easier. Notice that in the examples above you need to repeat the module name in front of the function call whenever you use a function from `my_module`. However, you can import functions directly into your notebook so that you don't have to type the module name every time you want to use a function:

```
In [7]: from my_module import add_numbers, subtract_numbers

add_numbers(2, 5)
```

Out [7]: 7

Another useful way of working with modules and their functions is by assigning them an alias. We'll be using this style of importing modules frequently in the following chapters:

In [8]: `import my_module as mm`

```
mm.subtract_numbers(2, 5)
```

Out [8]: -3

Here, you gave `my_module` an alias called `mm`, which you can then use throughout your notebook to refer to `my_module` and its functions. You've already seen that this style of importing used with the `pandas` library:

In [9]: `import pandas as pd`

Now you know what this `import` statement does. Aliases are usually short identifiers you can use to access module functions without having to type too much while also making it clear in your code what module a certain function is from. In the example above, the `pd` alias is used to refer to the `pandas` library — `pd` is commonly used for `pandas`, but you can use any alias you want as long as it is a valid Python name (i.e., doesn't start with a number or contains spaces).

Similar to how Python modules are just text files (with a `.py` extension), Python packages are folders which contain many Python modules. Packages can also be organized hierarchically, meaning that Python packages can contain sub-packages (i.e., other folders containing Python modules) and regular Python modules.

Python modules and packages are tools for organizing and sharing code across multiple notebooks or with other Python users; they're just files and folders with Python code. In practical terms, you'll use both modules and packages in the same way, by importing them into your notebooks and calling their functions. Even so, it's a good idea to know what they are so you don't get confused when you read about them online or when you see more intricate `import` statements in other code examples.

Libraries

The term library doesn't have a specific definition³ but it is used to refer to a collection of modules and packages that provide some functionality (e.g., making plots, working with tabular data). For example, `pandas` is commonly referred to as a library⁴ but it's just a collection of Python packages and modules (i.e., files and folders with some Python code).

3: Python modules and packages do: they are described in the official Python documentation, you can read more about them at docs.python.org/3/tutorial/modules.html.

4: It is a *library* rather than an *application* because it provides very specific functionality: you can't really *do* anything with `pandas` without writing additional Python code.

When you install a Python library on your computer (through Anaconda Navigator), you copy some Python files from a remote computer used to distribute Python libraries to your computer. Anaconda keeps everything organized (i.e., creates various folders on your computer to store these files separately, makes sure Python “knows” where these files are stored so you can load them in your notebook using the `import` keyword, keeps them updated when new versions become available, etc.).

Another library that you will use frequently is the Python standard library. It is a collection of unrelated modules and packages that provide all sorts of functionality: from working with large lists efficiently, to accessing web pages directly in your code. It is *standard* because it comes with most Python interpreters (so when you install Python, you already get these modules and packages).

There are many different modules in the Python standard library, many of which are designed for specific and technical use-cases (e.g., tools for working with HTML or XML files). Below is a quick tour of some of the modules you might find useful, but you can read about all of them in Python’s documentation.⁵

- ▶ The **os module** is helpful when you want to access local files. For example, to list all the files in your current folder.⁶

In [10]:

```
import os  
  
os.listdir()
```

Out [10]:

```
['my_module.py',  
 '__pycache__',  
 'Python ABCs.ipynb',  
 '.ipynb_checkpoints']
```

You can use the output of `listdir` to get a list of files and then go through it (e.g., using a `for` loop) to process each file. You can also pass a folder path as an argument to `listdir` to get all files at that path (e.g., `os.listdir('C:/Documents/Python workspace')` would list all files and folders at that path).

In addition to `listdir`, the `os` module has a submodule called `path` you can use for various operations with file or folder paths (e.g., check if a file exists on your drive). We will use the `os` module and its `path` submodule later in the book, to read multiple Excel files at once.⁷

- ▶ The **datetime module** is helpful when working with date and time values (e.g., subtracting a date from another, adding a number of days to a date, etc.). As an example:

5: At docs.python.org/library.

6: In the code output, there are two names you might not recognize. The `__pycache__` folder is created by the Python interpreter and is not in any way useful to you. The `.ipynb_checkpoints` is where JupyterLab auto-saves temporary copies of your notebooks, in case anything happens and your work is not saved (similar to how Excel saves temporary copies of your spreadsheets in case it crashes).

7: You can read more about `os.path` at docs.python.org/library/os.path.

```
In [11]: from datetime import date, timedelta  
  
today = date.today()  
days_of_holiday = timedelta(days=14)  
  
today + days_of_holiday
```

```
Out [11]: datetime.date(2020, 9, 3)
```

You can also use it to convert string values to date values and back. We'll take a closer look at date values when we start working with tables, in the next part of the book.⁸

- ▶ The **json module** is useful when you want to convert Python dictionaries to JavaScript object notation (i.e., JSON) or vice-versa. Many software tools or online services use JSON for various features (e.g., often JSON is used to store configuration options), so you might find the json module useful when you want to interact with such services. If you are not quite sure what JSON is, don't worry, we won't use it anywhere in this book.

8: More about the `datetime` module at docs.python.org/library/datetime.

Finding useful libraries

Besides the libraries I mentioned so far, and the ones we'll be using later in the book (e.g., `pandas`, `matplotlib`, `seaborn`), Python's ecosystem of libraries is vast (that's one of Python's main strengths). It can also be overwhelming at times: there are so many libraries, many of them with similar features, it can be hard to figure out which one to use.

You can use Anaconda Navigator to search for libraries and read more about each one. Besides Anaconda Navigator, the Python Package Index (PyPI)⁹ or GitHub¹⁰ are online homes for many Python libraries. Whenever you need more information about a library or want to know if it can help with your problem, check these resources.

9: pypi.org.

10: github.com.

Summary

This chapter showed you how to make your own Python module. Modules, packages, and libraries are Python features that help you share code between notebooks or with other Python users.

Python's vast ecosystem of libraries is one of its main strengths. But when you find a library that seems to solve your problem, how do you figure out its features and use it in your own code? JupyterLab has built-in tools that can help you navigate Python's universe – let's take a closer look at some of these tools next.

How to find help

If you made it so far, you might be wondering “*How do people remember all this stuff?!*”. The answer is they don’t — everyone searches for help on Google¹ all the time. Finding information about functions and modules is as essential a skill as knowing Python basics.

All the tools you’ll be using in this book have extensive documentation available online. When you get stuck, that’s where you’ll find how to fix your issues. For instance, Python’s functions, modules, and packages are documented at docs.python.org — searching for anything Python-related online will often send you there. The other libraries we’ll use later in the book have similar documentation websites that I’ll reference as we progress.

Even though documentation websites are comprehensive, they can often be cryptic and hard to navigate. Fortunately, there are a few helpers built-into Jupyter notebooks that make accessing docs more straightforward than sifting through official websites or plain googling. Let’s take a closer look at these helpers.

Jupyter notebook helpers

Jupyter notebooks enable you to view documentation directly as cell outputs in your notebooks: to access documentation, type a variable or function name, add the `?` character at the end, and then run the code cell as you usually do (i.e., by pressing the Shift + Enter keys); for example:

```
In [1]: message = 'hello, python for accounting!'

message.title?

Out[1]: Signature: message.title()
Docstring:
Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining
cased characters have lower case.
Type:      builtin_function_or_method
```

The output above is the documentation for the `title` method, which is available on string variables. It explains what the method does (the same developers that wrote the method code wrote this documentation).

1: Or their search engine of choice.

You can use the `?` character with any variable or function name, regardless of what library or module it comes from. If there's documentation available, JupyterLab will show it in your notebook.

Whenever you get stuck or can't remember what a function does, type its name followed by `?` and run that in a cell. Even though documentation can be puzzling at times, libraries like `pandas` typically include many code examples in their documentation, which can help you figure out how they work.

But what if you don't know what functions or methods are available? In that case, you can use the autocomplete feature in Jupyter notebooks. If you type the following code:

```
In [2]: message = 'hello, python for accounting!'

message.
```

And instead of running the code cell, you press the `TAB` key right after the dot character in your code cell, you should see the JupyterLab autocomplete menu:

```
message = 'hello, python for accounting!'

message.
```

f	capitalize	function
f	casefold	function
f	center	function
f	count	function
f	encode	function
f	endswith	function
f	expandtabs	function
f	find	function
f	format	function
f	format_map	function

Figure 10.1: Pressing the `TAB` key in a code cell opens the autocomplete menu. In this example, notice the blue `f` before each menu option, which indicates a *function* instead of an *attribute*. Attributes are marked with an orange `i` (which stands for *instance*).

The autocomplete menu above lists all attributes and methods associated with the `message` variable (which is a string object, so the options here include the `capitalize` method and many others). You can navigate this menu using the arrow keys and select one of the items by pressing `Enter` — and if you don't know what a method does exactly, but the name sounds interesting, you can add a `?` at the end of its name and run that in a code cell to find out.

You can press `TAB` to autocomplete any snippet of code (including variable names), and JupyterLab will show you a list of options (i.e., methods or attributes) that are available in your context.

Finding help online

The two online resources you'll make the most use of as you continue on your Python journey are Google² and Stackoverflow.

2: DuckDuckGo or Bing are good alternatives.

Google needs no introduction, and you'll use it frequently — even to find Python's documentation website.

Stackoverflow³ is a question-answering platform where Python users (and users of other programming languages) ask questions or contribute answers about all sorts of coding problems they face. You will sometimes get cryptic error messages that make it hard to decipher what went wrong in your Python code. In those cases, a good idea is to quickly search the web (using the error message as your query) to see if anyone else has an explanation or suggested a fix on Stackoverflow. At times, quickly searching the web for an answer to your problem is what makes you most productive with Python.

3: stackoverflow.com.

However, there is a catch: if you always have to search the web to get unstuck, you'll quickly become frustrated, and using Python will be a nuisance instead of a benefit. You'll forget method names or what arguments to use because everybody forgets the details, but if you develop the mental models and learn the right vocabulary and concept names⁴, you can solve anything with Python, Google and Stackoverflow; unfortunately, all the googling in the world can't help without knowing what to look for.

4: You can do that by finishing this book!

Summary

Python and all its libraries are well documented. This short chapter showed you how to access documentation directly in your Jupyter notebook, either by using the `?` character or through the autocomplete menu.

Overthinking: code style

This overthinking chapter covers coding style (i.e., how to name variables, how much whitespace between operators to use, how to keep code comments useful). As it happens, Python has unambiguous and widely adopted style guidelines — these guidelines are explicitly laid out in a document called *Python Enhancement Proposal 8* (also known as PEP 8).¹

Writing code using best practices adopted by other Python users can make your code easier to understand (by others and by yourself, when you revisit it). Even more, because you'll often read other people's code when looking for examples online, adopting these best practices for your code will make it easier for you to understand their code as well.

PEP 8 is a lengthy document, covering all sorts of formatting issues related to Python code (e.g., how many empty lines between function definitions, how many whitespace characters after opening parentheses, and many others). However, in this chapter, I'm going to introduce the essential style rules. You don't have to follow all these practices for your code to run without errors, but knowing about these practices will help you write cleaner code.

The Python examples you find in this book follow PEP 8 guidelines so if you write your code in the same style, it will follow tried-and-tested practices that most Python users have adopted. As a general rule, if you have to choose between code that is easier (or faster) to write, and code that is easier to read, always choose the latter.

Naming variables and functions

You can be as creative as you want with variable or function names, but there are a few rules you should follow. Some of these naming rules keep your code from throwing errors, while others make your code easier to read:

- ▶ **Don't use reserved keywords as variable or function names.** For example, don't call your variables `print`, `for` or `str`. You will quickly notice if a term is reserved because it gets highlighted in JupyterLab (i.e., certain reserved terms stand out in a different color, usually green, like in the examples you've seen so far in the book). If a term turns green, don't use it as a variable or function

¹: You can read it all at python.org/dev/peps/pep-0008.

Python Enhancement Proposals (commonly known as *PEPs*) are documents created by developers in the Python community in which new Python features are discussed. PEPs are the primary mechanism through which the Python programming language evolves and is adapted to meet the ever-changing requirements of modern technology. All PEPs are publicly available at python.org/dev/peps.

name. Not following this rule won't cause any apparent errors at first but it will make your code behave in strange ways.

- ▶ **Use only letters, numbers, and underscores in your variable or function names — spaces are not allowed.** Your variable names can start with letters or underscore(s), but not with numbers. For example, `__variable__1` is a valid name, whereas `1_variable__` or `variable 1` are not. Not following this rule throws an error.
- ▶ **Try to keep your variable or function names descriptive but short.** For example, `account_number` is better than `acc_n` and better than `number_of_the_account`.

It's also a good idea to keep all your variable or function names lowercase, with distinct words separated by underscores, such as `account_name`. This style of writing compound words is known as *snake case*.² Python uses snake case almost everywhere (in variable names, function names, module or package names), so it makes sense to write your code in the same style.

2: Another instance of programmer humor: Python uses snake case.

Most people use English as their coding language (i.e., their variable or function names use English words). This is because almost all Python resources (i.e., official documentation, tutorials, Q&A content) and most programming tools (e.g., JupyterLab, Python itself) are available in English only. You don't have to make the same choice, particularly if your code is for your eyes only, but I recommend you try to use characters from the Latin alphabet in your code — so no ñ, ô or any other accented characters.

Naming variables is probably one of the most challenging aspects of computer programming, yet it seems very simple. Reading other people's code (e.g., examples you find online) is a good source of inspiration for what naming style to use in your own work. It takes some practice to come up with good variable names; as you write and read more code, you'll develop a strong intuition for choosing descriptive names.

Breaking long lines of code

Sometimes you will need to write a long line of code that involves many different variables:

```
In [1]: income = (gross_wages + taxable_interest + (dividends - qualified_dividends) -
    ira_deduction - student_loan_interest)
```

Having very long code lines can make it hard to understand what the code does exactly (in the example above, it's difficult to follow which variables get added and which get subtracted to get the `income` variable). To improve readability, you should break lines *before* mathematical operators and indent each subsequent line:

```
In [2]: income = (gross_wages
                  + taxable_interest
                  + (dividends - qualified_dividends)
                  - irs_deduction
                  - student_loan_interest)
```

Notice that breaking lines this way requires the use of parentheses around the formula. Leaving the parentheses out will prompt you with an indentation error:

```
In [3]: income = gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - irs_deduction
          - student_loan_interest
```

```
File "<ipython-input-13-9e189a5adfa9>", line 2
      + taxable_interest
      ^
IndentationError: unexpected indent
```

Spacing around operators

Try to surround Python operators with a single space on either side. The Python operators that should be surrounded by whitespace are: assignment (=), comparisons (==, <, >, !=, <=, >=, in, not in), booleans (and, or, not).

```
In [4]: # Yes:
i = i + 1

# No:
i=i+1
```

However, when calling functions, don't use spaces around the = sign if you are using keyword arguments (or when indicating default parameter values in function definitions):

```
In [5]: # Yes:
def get_account(name, id=0.0):
    return find_account(n=name, i=id)

# No:
def get_account(name, id = 0.0):
    return find_account(n = name, i = id)
```

More generally, avoid extra space on the same line when calling functions or when defining variables:

```
In [6]: # Yes:
long_function_name(list_argument[1], dictionary_argument['key'])

# No:
long_function_name( list_argument[ 1 ] , dictionary_argument[ 'key' ] )

In [7]: # Yes:
x = 1
y = 2
long_variable = 3

# No:
x           = 1
y           = 2
long_variable = 3
```

There's much more to styling Python code than I can cover here.³ As with naming variables, the best way to acquire code styling best practices is to read and write more code.

3: Companies like Google or Facebook typically document their coding style practices in entire books.

Code comments

No matter how descriptive your variable names are, adding comments to your code will make it easier for yourself and others who read it to understand the logic behind the code. Commenting data analysis code is particularly important because it allows you to explain how data manipulations were applied and why they are important for interpreting analysis outcomes.

In Python, anything that comes after the hash `#` character gets ignored when your code runs.⁴

```
In [8]: # a commented line          1
message = "hello" # inline comment      2
```

Most of the examples shown in this chapter are very simple and don't justify detailed comments. Indeed, sometimes comments make your code more difficult to understand rather than the opposite:

```
In [9]: # the following line of code      1
# prints the word "hello" on a         2
# separate line then terminates       3
print('hello')                      4
```

```
hello
```

4: In Jupyter notebooks, both code comments and Markdown cells can be used to document your code and add narrative text. Remember that the hash symbol does different things depending on cell type: in code cells, it starts a comment line; in Markdown cells, it adds a header style to text.

It's up to you to figure out the level of detail needed in your code, and as with naming variables, getting this right takes some practice. At first, you can add a cell at the top of your notebook with your name, date, and short description of what you want to achieve

in that notebook. Even simple comments like these will help you keep your work organized:

```
In [10]: # author:      horatio      1
# date:        08/17/2020      2
# description: python ABCs      3
```

Jupyter notebooks also allow you to write styled text using Markdown cells. You might not be sure whether to include code explanations in comments or Markdown cells. A good rule of thumb is to keep code explanations or meta-data (such as your name and date) in comments and longer narrative text that explains the data analysis process or the business context for the analysis in Markdown cells.

One last aspect related to comments that you should be aware of is that they tend to go stale, meaning that after code changes, the comments describing it do not get updated to reflect those changes. Try to keep comments up-to-date with code changes. Old comments that contradict what is happening in the code are often more confusing than no comments at all.

Summary

This chapter showed you some tips for how to style your Python code: how to name variables, how to break long lines of code so they're easier to read, how to use whitespace and comments, and a few others. These style guidelines are a part of Python, just like its keywords and operators; following them will pay off.

WORKING WITH TABLES

PART TWO

Working with tables is where the rubber meets the road in using Python for accounting. This part of the book introduces the main features of the `pandas`¹ Python library: a powerful and straightforward data manipulation tool that you will be using to work with tables — whether Excel spreadsheets, CSV files, or any other tabular data — in Python. If you used Anaconda to install Python, `pandas` is already on your computer.

Many of the things you do in Excel right now can be done in `pandas`, but often in an easier, faster, and reproducible way. Learning how to use `pandas` will give you the most return on the investment you're making by reading this book.

¹: The `pandas` library has been in active development since 2010 and now has almost two thousand developers and enthusiasts from various fields contributing to it. It is a mature piece of software that has become an essential data science tool. It was initially designed for financial time series analysis but has slowly taken over many other domains.

Setup

In your Python for Accounting workspace, there is a folder for the second part of the book. In it, you will find a Jupyter notebook for each of the chapters ahead, as well as the data files you need for the code examples we'll go through.

You'll mainly work with two datasets:

- ▶ “*Q1Sales.xlsx*” is an Excel file containing a wholesale supplier’s sales account entries during the first quarter of 2020. It contains three sheets, one for each month in Q1 2020. The first five rows in the “*January*” sheet are shown below:

	InvoiceNo	Channel	Product Name	ProductID	Account	\
0	1532	Shoppe.com	Cannon Water Bomb Balloon...	T&G/CAN-97509	Sales	
1	1533	Walcart	LEGO Ninja Turtles Stealt...	T&G/LEG-37777	Sales	
2	1534	Bullseye		NaN	T&G/PET-14209	Sales
3	1535	Bullseye	Transformers Age of Extin...	T&G/TRA-20170	Sales	
4	1535	Bullseye	Transformers Age of Extin...	T&G/TRA-20170	Sales	

	AccountNo	Date	Deadline	Currency	Unit Price	Quantity	Total
0	5004	2020-01-01	11/23/19	USD	20.11	14	281.54
1	5004	2020-01-01	06/15/20	USD	6.70	1	6.70
2	5004	2020-01-01	05/07/20	USD	11.67	5	58.35
3	5004	2020-01-01	12/22/19	USD	13.46	6	80.76
4	5004	2020-01-01	12/22/19	USD	13.46	6	80.76

The supplier sells thousands of products, across many different product categories, through both online (“*iBay.com*”, “*Shoppe.com*”, “*Understock.com*”) and retail (“*Bullseye*” and “*Walcart*”) stores — values in the `Channel` column identify the sale store.

There are few problems with this dataset, some of which you may have already noticed (e.g., missing values, duplicate rows, and a few others). As I introduce more of Python’s table handling tools in the following chapters, you’ll see how easy it is to fix these problems with Python and `pandas`.

- ▶ “*products.csv*” is a CSV file containing additional information about each product sold by the supplier. The first five rows in “*products.csv*” are listed below:

	ProductID	Product Name	Brand	Category
0	MI/SNA-81654	Snark SN-5 Tuner for...	Snark	Musical Instruments
1	MI/STU-67796	Studio Microphone Mi...	Generic	Musical Instruments
2	MI/MUS-73312	Musician's Gear Tubu...	Musician's Gear	Musical Instruments
3	MI/STR-01505	String Swing CC01K H...	String Swing	Musical Instruments
4	MI/DUN-82082	Dunlop 5005 Pick Hol...	Jim Dunlop	Musical Instruments

Later in this part of the book, you’ll see how you can join the two datasets on their common `ProductID` column using `pandas`.

I designed these datasets to suit the tasks you’ll be doing, but they are based on real-world public data.² Some of the project chapters ahead use other datasets instead of the ones mentioned above — I’ll introduce those datasets as they’re needed.

When you’re ready, launch JupyterLab, go to your workspace, open the first notebook in part two, and let’s see what `pandas` can do.

2: The products dataset I used to create these data was made available by Julian McAuley at jmcauley.ucsd.edu/data/amazon.

This chapter is a brief tour of pandas's main features. It aims to show you how Excel tables look like in Jupyter notebooks and get you acquainted with pandas syntax — which is slightly different from the Python code you saw in the previous chapters. Even though pandas has its particular code style, it is designed around the same principles of code readability and simplicity at the core of Python, so you'll quickly learn its ropes.

The pandas library is a data analysis and data manipulation tool. While there's more to accounting than analyzing data, there's also plenty of overlap between the two, which is why having pandas in your toolkit can be very handy.

Working with data, whether in accounting or astronomy, typically involves the same high-level tasks:

- ▶ **Reading and writing data:** from and to various data sources, including Excel spreadsheets, CSV or HTML files, or different databases.
- ▶ **Preparing and transforming data:** cleaning, combining and re-shaping data to get it in the format you need it; aggregating data and deriving new datasets by computing metrics on aggregated data (e.g., pivoting a table).
- ▶ **Visualizing results:** creating plots or table summaries of your data to communicate your work.

The pandas library has built-in features that support all these high-level tasks. We'll go over these features in plenty of detail later; for now, let's take a quick look at how pandas works.

Getting started

To use pandas, you first need to `import` it in your notebook:

```
In [1]: import pandas as pd
```

You already know that the `import` keyword gives you access to Python code stored somewhere on your computer (i.e., the Python libraries you installed with Anaconda). In this case, the `import` statement gives you access to pandas's code. It is common to use the `pd` alias when importing pandas, as above. This alias makes it easier to reference pandas functions without having to type pandas

in front of them — whenever you see `pd` in code examples, it's referring to `pandas`.

Reading data from spreadsheets

Accessing data often means just opening a file on your computer — in accounting, that file is likely an Excel workbook.

You can read the “`Q1Sales.xlsx`” Excel file from your workspace folder and assign it to a variable called `ledger_df` using `pandas`'s `read_excel` function:

```
In [2]: ledger_df = pd.read_excel('Q1Sales.xlsx')
```

The `ledger_df` variable created above is a `DataFrame`. `DataFrame` objects are just `pandas`'s way of representing entire tables in Python code — more details on how they work in the following chapter.

You can inspect `ledger_df` in a separate code cell to see that it is indeed an entire table of values:

```
In [3]: ledger_df
```

```
Out [3]:
```

	InvoiceNo	Channel	Product Name	Unit Price	Quantity	Total
0	1532	Shoppe.com	Cannon Water Bom...	20.11	14	281.54
1	1533	Walcart	LEGO Ninja Turtl...	6.70	1	6.70
2	1534	Bullseye	Nan	11.67	5	58.35
3	1535	Bullseye	Transformers Age...	13.46	6	80.76
4	1535	Bullseye	Transformers Age...	13.46	6	80.76
...
14049	15581	Bullseye	AC Adapter/Power...	28.72	8	229.76
14050	15582	Bullseye	Cisco Systems Gi...	33.39	1	33.39
14051	15583	Understock.com	Philips AJ3116M/...	4.18	1	4.18
14052	15584	iBay.com	Nan	4.78	25	119.50
14053	15585	Understock.com	Sirius Satellite...	33.16	2	66.32

[14054 rows x 12 columns]

The output above looks slightly different from the one in your Jupyter notebook, which should look more like figure 12.1 below. I use a compact display style for tables in the book, but they still refer to the same underlying data as those you see in your Jupyter notebook when you run the code examples.

Notice that, in your notebook, a truncated version of the table is shown. You only see the top five and bottom five rows as the code output, and in the middle of the table, you have ellipses instead of actual rows of values.

By default, large tables that have more than ten rows¹ are shown this way to keep them from filling up your entire notebook — the underlying data is still there, just not shown in your notebook.

1: If your tables have more than twenty columns, the middle columns will also be truncated when you try to see them in your notebook.

The screenshot shows a Jupyter notebook interface with the title "Chapter 01 - Pandas in a nutshell". In the code cell [1], the command `import pandas as pd` is run. In the code cell [2], the command `ledger_df = pd.read_excel('Q1Sales.xlsx')` is run. In the code cell [3], the variable `ledger_df` is displayed, showing a truncated view of a DataFrame with 14054 rows and 12 columns. The DataFrame contains columns for InvoiceNo, Channel, Product Name, ProductID, Account, AccountNo, Date, Deadline, Currency, Unit Price, Quantity, and Total. A blue box highlights the header row and the first five data rows. A red box highlights the text "Number of rows and columns" at the bottom of the DataFrame output.

	InvoiceNo	Channel	Product Name	ProductID	Account	AccountNo	Date	Deadline	Currency	Unit Price	Quantity	Total
0	1532	Shoppe.com	Cannon Water Bomb Balloons 100 Pack	T&G/CAN-97509	Sales	5004	2020-01-01	11/23/19	USD	20.11	14	281.54
1	1533	Walcart	LEGO Ninja Turtles Stealth Shell in Pursuit 79102	T&G/LEG-37777	Sales	5004	2020-01-01	06/15/20	USD	6.70	1	6.70
2	1534	Bullseye	NaN	T&G/PET-14209	Sales	5004	2020-01-01	05/07/20	USD	11.67	5	58.35
3	1535	Bullseye	Transformers Age of Extinction Generations Del...	T&G/TRA-20170	Sales	5004	2020-01-01	12/22/19	USD	13.46	6	80.76
4	1535	Bullseye	Transformers Age of Extinction Generations Del...	T&G/TRA-20170	Sales	5004	2020-01-01	12/22/19	USD	13.46	6	80.76
...
14049	15581	Bullseye	AC Adapter/Power Supply Unit (RV320K9NA)	E/CIS-74992	Sales	5004	2020-01-31	February 23, 2020	USD	28.72	8	229.76
14050	15582	Bullseye	Cisco Systems Cisco Small Business RV320K9NA	E/PHI-08100	Sales	5004	2020-01-31	January 21, 2020	USD	33.39	1	33.39
14051	15583	Understock.com	Philips AJ3116M/37 Digital Tuning Clock Radio ...	E/POL-61164	Sales	5004	2020-01-31	March 22, 2020	USD	4.18	1	4.18
14052	15584	iBay.com	NaN	E/SIR-83381	Sales	5004	2020-01-31	June 25, 2020	USD	4.78	25	119.50
14053	15585	Understock.com	Sirius Satellite Radio XADH2 Home Access Kit f...					February 01, 2020	USD	33.16	2	66.32

14054 rows × 12 columns Number of rows and columns

Figure 12.1: You can view a `DataFrame` variable in your Jupyter notebook by typing its name in a cell and running that cell. By default, the output is truncated, so you do not see the entire `DataFrame`, but rather only its top five and bottom five rows (if your `DataFrame` has more than twenty columns, the middle columns will also be truncated when you try to view it). This is by design because working with tables using `pandas` is different from working with tables in Excel: you do not edit table cells directly, you manipulate entire columns or the whole table by writing code. In such a setting, seeing the entire table in your notebook is less useful and just occupies a lot of visual space.

Tables are truncated by design because working with tables in `pandas` is different from working with tables in Excel: you can't edit values directly by clicking on a table cell; you need to write some code to do that.² In such a setting, displaying the entire table in your notebook is not that useful: there's not much you can do with the displayed table, and it just occupies a lot of visual space, which is why you see a truncated version.

You might find this truncated output challenging to work with at times, especially if you're used to Excel. However, you can display entire tables in your notebooks if you want to (and configure other options related to table display), as you'll see later.

The key thing to remember is that `ledger_df` still contains all the data read from "`Q1Sales.xlsx`"³ even though when you display it in your Jupyter notebook, you do not see all of its contents.

2: Most often, you will manipulate entire columns or the whole table, not individual cell values, using `pandas`'s functions or methods.

3: Right now it contains data from the first sheet in "`Q1Sales.xlsx`", but you can read data from multiple sheets just as easily.

Preparing and transforming data

Now that you know what a `DataFrame` looks like in your Jupyter notebook, let's go through some common ways to filter, prepare,

and transform data using pandas.

Unlike Excel, in pandas you reference columns by name⁴, not by label: instead of column B, you have column 'Channel'. To select all values from the 'Channel' column, run the following code:

```
In [4]: ledger_df['Channel']
```

```
Out [4]: 0      Shoppe.com
1      Walcart
2      Bullseye
3      Bullseye
4      Bullseye
...
14049    Bullseye
14050    Bullseye
14051  Understock.com
14052    iBay.com
14053  Understock.com
Name: Channel, Length: 14054, dtype: object
```

4: Column names are the ones listed in the table header.

This line of code returns a list-like sequence of values you can use for any operation you usually perform on columns in Excel. For example, to see how many times each value shows up in the 'Channel' column:

```
In [5]: ledger_df['Channel'].value_counts()
```

```
Out [5]: Understock.com    4821
Shoppe.com        3132
iBay.com         2877
Walcart          1851
Bullseye         1373
Name: Channel, dtype: int64
```

Or to compute the sum of the 'Total' column:

```
In [6]: ledger_df['Total'].sum()
```

```
Out [6]: 1823206.56
```

The power of using Python and pandas is even more apparent when you use the two together. For instance, you can define a custom Python function and apply it to all the values in a column:

```
In [7]: def make_upper(value):
    return value.strip('.com').upper()

ledger_df['Channel'].apply(make_upper)
```

```
Out [7]: 0      SHOPPE
1      WALCART
2      BULLSEYE
3      BULLSEYE
4      BULLSEYE
...
14049    BULLSEYE
```

```
14050      BULLSEYE
14051      UNDERSTOCK
14052          IBAY
14053      UNDERSTOCK
Name: Channel, Length: 14054, dtype: object
```

The `make_upper` function above removes `'.com'` from a string value passed as an argument, and returns an uppercase version of the modified string. You could use a `for` loop to call the function on each value in the `'Channel'` column, but pandas makes it easy to apply `make_upper` on all values at once.

Although this code works, it doesn't update the initial table. To update all values in the `'Channel'` column, you need to assign the output of applying `make_upper` back to the `'Channel'` column:

```
In [8]: ledger_df['Channel'] = ledger_df['Channel'].apply(make_upper)
```

Now take a look at `ledger_df` to see the changes:

```
In [9]: ledger_df
```

```
Out [9]:   InvoiceNo    Channel       Product Name ... Unit Price Quantity  Total
0          1532    SHOPPE    Cannon Water Bom... ...  20.11      14  281.54
1          1533    WALCART    LEGO Ninja Turtl... ...   6.70       1   6.70
2          1534    BULLSEYE           NaN     ...  11.67       5  58.35
3          1535    BULLSEYE  Transformers Age... ...  13.46       6  80.76
4          1535    BULLSEYE  Transformers Age... ...  13.46       6  80.76
...
14049      15581    BULLSEYE  AC Adapter/Power... ...  28.72       8 229.76
14050      15582    BULLSEYE  Cisco Systems Gi... ...  33.39       1  33.39
14051      15583  UNDERSTOCK  Philips AJ3116M/... ...   4.18       1   4.18
14052      15584          IBAY           NaN     ...   4.78      25 119.50
14053      15585  UNDERSTOCK  Sirius Satellite... ...  33.16       2  66.32

[14054 rows x 12 columns]
```

You can define and apply any custom function you need on any of the table columns — even on multiple columns at once.

Filtering data is also straightforward by chaining conditional statements in the following way:

```
In [10]: ledger_df[
    (ledger_df['Channel'] == 'WALCART') &
    (ledger_df['Product Name'].str.contains('Camera')) &
    (ledger_df['Quantity'] > 10)
]
```

```
Out [10]:   InvoiceNo    Channel       Product Name ... Unit Price Quantity  Total
1342        2874    WALCART  3 in 1 Camera Le... ...  15.01      34 510.34
5812        7344    WALCART  Olympus SP-820UZ... ...  19.48      16 311.68
6148        7680    WALCART  Casio EXILIM Dig... ...  14.91      18 268.38
8358        9890    WALCART  3 in 1 Camera Le... ...  15.01      23 345.23
10577      12109    WALCART  Foscam FI8910W W... ...  23.69      40 947.60
11713      13245    WALCART  Foscam New Versi... ...   4.12      38 156.56
```

[6 rows x 12 columns]

The code above filters the original table to keep rows where values in the 'Channel' column are 'WALCART' exactly, values in the 'Product Name' column contain the word 'Camera', and values in the 'Quantity' column are greater than 10.

This style of chaining conditional statements in pandas is different from Python `if-else` statements, but isn't tricky to master — we will go through more filtering examples in the following chapters.

There are several methods you can use to summarize tables in pandas, one of which is the familiar `pivot_table`. For example, to compute the daily totals for each channel in these sales data, you can use:

```
In [11]: ledger_df.pivot_table(
    index='Date',
    columns='Channel',
    values='Total',
    aggfunc='sum'
)
```

	Channel	BULLSEYE	IBAY	SHOPPE	UNDERSTOCK	WALCART
Date						
2020-01-01	9179.39	5637.54	6911.72	20707.62	13593.17	
2020-01-02	5652.32	5959.61	17351.46	18280.59	12040.16	
2020-01-03	6127.92	8346.60	10578.60	17191.15	9876.21	
2020-01-04	10370.95	10168.41	6052.03	17034.69	12811.26	
2020-01-05	4641.02	12462.30	11866.74	17074.18	8318.34	
...
2020-01-27	692.58	8244.02	11004.59	28902.72	1762.95	
2020-01-28	1016.11	18908.66	11705.26	29157.55	2932.35	
2020-01-29	1025.87	18472.83	11861.63	31370.99	1755.18	
2020-01-30	3814.04	12742.84	12872.70	20599.03	1216.91	
2020-01-31	3586.07	15104.62	12375.40	32507.55	12741.69	

[31 rows x 5 columns]

The output of any of these method or function calls can be assigned to other variables and used in different parts of your code. For example, you can assign the pivot table above to a new variable called `daily_totals_df`:

```
In [12]: daily_totals_df = ledger_df.pivot_table(
    index='Date',
    columns='Channel',
    values='Total',
    aggfunc='sum'
)
```

Now you can use `daily_totals_df` as a separate table and slice, filter, or reshape it as we did with `ledger_df` so far. This is the style

of code we'll be exploring in detail in the next chapters: moving from one table to another using Python and `pandas`.

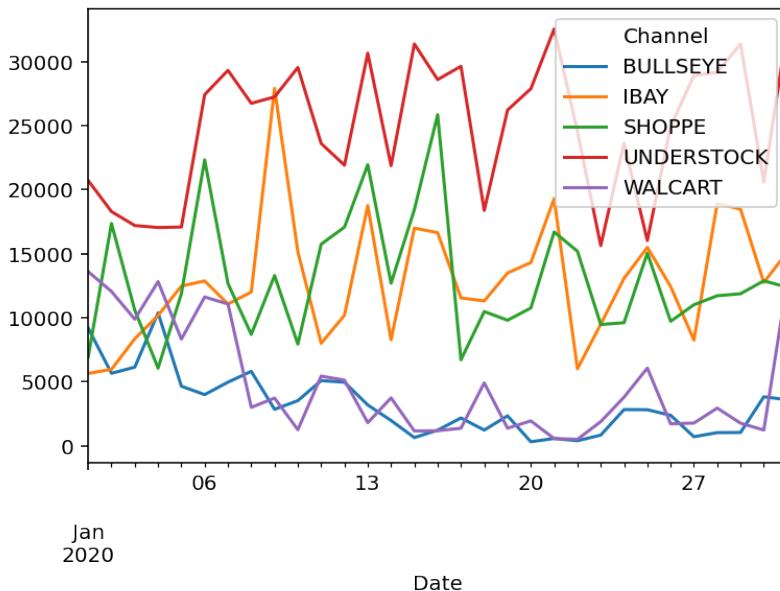
If any of these examples are unclear right now, don't worry, we'll revisit them in the following chapters. But first, let's quickly turn `daily_totals_df` into a plot.

Visualizing data

Data visualization is built-into `pandas` — you can turn a `DataFrame` into a plot by simply calling the `plot` method:

```
In [13]: daily_totals_df.plot()
```

```
Out [13]: <matplotlib.axes._subplots.AxesSubplot at 0x126cea690>
```



As you can see, plots are displayed underneath code cells, directly in your notebook. The plot here uses the `daily_totals_df` table you created earlier with the `pivot_table` method. It shows the total daily revenues for each of the channels in these sales data. While the details of how to customize this plot will have to wait, you can already see how simple it is to create the plot.

You can make various types of plots using `pandas`, not just line plots. In addition, there are other Python visualization libraries you can use for specific use-cases — we'll cover the most popular ones later in the book.

Writing data to a spreadsheet

Finally, to write `daily_totals_df` back to an Excel spreadsheet, you can use the code below:

```
In [14]: daily_totals_df.to_excel('DailyTotals.xlsx')
```

After you run the code, open “*DailyTotals.xlsx*” in Excel⁵ to see that it has same contents as `daily_totals_df`.

5: The file will be in the same folder as your Jupyter notebook.

This concludes our quick tour of `pandas`. The aim of this tour was to show you how tables look like in Jupyter notebooks and briefly illustrate how `pandas` can be used for everyday operations with tabular data (e.g., filtering, pivoting, plotting). As you’ll see over the next chapters, `pandas` provides out-of-the-box tools for almost any kind of data manipulation you can imagine. On top of that, `pandas` is engineered to be fast even with data many times larger than what Excel can handle.

Now that we covered the quick tour of `pandas`, let’s take a more scenic route and look at the details of how `pandas` works.

Tables, columns, and values

If you work with Excel, you already know what tables and columns are — pandas has equivalent data structures¹ that go by different names but are otherwise similar to the tables and columns you know well. These are the `DataFrame`, `Series`, and `Index` objects.

As you saw in the previous chapter, a `DataFrame` is a table represented in Python code. It is made up of multiple `Series` objects (i.e., a `DataFrame` is made up of one or more `Series` objects just like a table is made up of one or more columns). Each row or column in a `DataFrame` object has an associated label you can use to access its values (e.g., a column or row name). These column and row labels are stored in an `Index` object, which is also part of the `DataFrame`.

`DataFrame`, `Series` and `Index` objects are central to working with pandas. What exactly these objects do and how they work together is the topic of this chapter.

Series

A `Series` is a pandas data structure² designed to represent a column in a table. It is a *labeled* sequence of values, so it is more like a Python dictionary than a Python list. The best way to illustrate this is by creating a `Series` object from a Python dictionary:

```
In [1]: import pandas as pd  
In [2]: pd.Series({'first_row': 3, 'second_row': -20, 'third_row': 121})  
Out [2]: first_row      3  
         second_row    -20  
         third_row     121  
        dtype: int64  
  
In [3]: pd.Series({0: 'Assets', 5: 'Revenue', -3: 'Sales'})  
Out [3]: 0      Assets  
5      Revenue  
-3      Sales  
        dtype: object
```

`Series` objects map a label to a value. Both labels and values in a `Series` can be of any type: strings, numbers, booleans, dates. The last line in the output above tells you what type of values the `Series` stores — more on pandas types later.

1: A data structure is an abstraction that refers to a collection of values and the functions, methods, or operations that can be applied to those values. For example, a Python list is a data structure.

2: As opposed to a Python built-in data structure, such as a list or a dictionary.

By design, each row in a `Series` is labeled. If you want to access a specific value in the `Series`, you need to use its label, as you do with a Python dictionary:

```
In [4]: values = pd.Series({'first_row': 3, 'second_row': -20, 'third_row': 121})

values['second_row']
```

```
Out [4]: -20
```

Unlike Python dictionaries, `Series` objects have more methods you can work with, many of which we'll go over in the next chapter. For example, you can sum the numbers in `values` using:

```
In [5]: values.sum()
```

```
Out [5]: 104
```

`Series` objects are designed to be more efficient than Python's built-in dictionaries (both in terms of speed and memory use), which is why `pandas` uses them instead.

DataFrame

`DataFrame` objects are tables represented in Python code.³ Besides containers for their values, they are also a collection of methods you can apply to those values.

You can put multiple `Series` objects in a Python dictionary to get a `DataFrame`. You won't create `DataFrame` objects this way often because `pandas` puts everything together for you whenever you read data from a file.⁴ However, for this example:

```
In [6]: df = pd.DataFrame({
    "first_column": pd.Series({"first_row": 3, "second_row": -20, "third_row": 121}),
    "second_column": pd.Series({"first_row": 0.23, "second_row": 2, "third_row": -3.5}),
    "third_column": pd.Series({"second_row": 'Assets'})
})
```

```
df
```

```
Out [6]:      first_column  second_column  third_column
first_row          3           0.23         NaN
second_row        -20          2.00       Assets
third_row         121          -3.50         NaN
```

3: Because you'll be using `pandas` exclusively in the chapters ahead, I'll sometimes use the terms *table* and *DataFrame* interchangeably.

4: Using its `read_excel` or any of the other `read` functions.

The output above is a `DataFrame` created from scratch (i.e., not by reading data from a file). The Python dictionary you used to create this `DataFrame` maps column names to `Series` objects (which themselves map row labels to values). To access data from this `DataFrame` object, you need to use its column or row labels:

```
In [7]: df['first_column']
```

```
Out[7]: first      3.0
         second    -20.0
         third     121.0
         Name: first_column, dtype: float64
```

```
In [8]: df['first_column']['second_row']
```

```
Out[8]: -20.0
```

In Excel, you reference table values by their column letter (i.e., a letter from A to Z) and their row number. In pandas, columns and rows have actual names, instead of single-letter labels or row numbers: you reference the '`first_column`' column by name, not by the letter A, as you do in Excel; similarly, you reference '`second_row`' by its name. DataFrame column and row labels don't have to be strings, they can be numbers or other types of values.

You may have noticed that our hand-crafted DataFrame has a few `NaN`⁵ values (if not, take a look at the last column in the table above). When you put multiple Series together in a DataFrame, they get *aligned* around their row labels. Wherever the Series row labels don't match, pandas adds `NaN` values so that each column in the final DataFrame has the same number of entries. The `NaN` is similar to Python's `None` in that both are used to indicate the absence of a value — unlike Excel's #N/A, which tells you something went wrong when calling a function.

5: `NaN` stands for *not a number*, not the bread.

Index and axes

Every DataFrame has labels associated with its rows and columns. These labels are accessible through the `index`, `columns`, and `axes` DataFrame attributes.⁶ For example, to access `df`'s column names, you can run:

```
In [9]: df.columns
```

```
Out[9]: Index(['first_column', 'second_column', 'third_column'], dtype='object')
```

Similarly, to access its row labels, you can use:

```
In [10]: df.index
```

```
Out[10]: Index(['first_row', 'second_row', 'third_row'], dtype='object')
```

Both column and row labels are stored in list-like sequences of values. These sequences of values are `Index` objects. An `Index` object is *list-like* because you can use it as a Python list (e.g., go through its values using a `for` loop or access its elements by their position):

```
In [11]: df.columns[2]
```

```
Out[11]: 'third_column'
```

6: Why *attributes* and not *methods*? Labels are stored internally in the DataFrame object when you create it (e.g., either by reading data from an Excel file or manually, as we did here). They are not *computed* when you work with the DataFrame. On the other hand, methods *compute* something (e.g., a sum) on the data stored inside the DataFrame.

```
In [12]: df.index[1]
```

```
Out [12]: 'second_row'
```

Index objects are not simple Python lists because they are designed to make accessing values in a `DataFrame` efficient and are more complicated in their internal machinery than Python lists. However, there is no practical difference: you will often use them as you use Python lists.

I want to highlight that there is no distinction between column and row labels in the way they work. For `pandas`, both sets of labels serve the same purpose: they point to values in the table. You can easily “rotate” your table to make column labels become row labels and vice-versa, using the `transpose` method:

```
In [13]: df
```

```
Out [13]:      first_column  second_column  third_column
first_row           3            0.23        NaN
second_row          -20           2.00      Assets
third_row           121           -3.50        NaN
```

```
In [14]: df.transpose()
```

```
Out [14]:      first_row  second_row  third_row
first_column         3          -20         121
second_column        0.23          2         -3.5
third_column         NaN        Assets        NaN
```

In the table above, the first row in `df` is now the first column. You probably won’t need to rotate tables often, but this example helps illustrate that `DataFrame` objects have the same kind of labels on both columns and rows, and there is no difference between the way they are stored or how they work.

Axes

You can also access a `DataFrame`’s row and column labels through its `axes` attribute:

```
In [15]: df.axes
```

```
Out [15]: [Index(['first_row', 'second_row', 'third_row'], dtype='object'),
Index(['first_column', 'second_column', 'third_column'], dtype='object')]
```

The output above is a Python list containing `df`’s row and column `Index` objects. There’s no practical benefit in using the `axes` attribute instead of `columns` or `index`; you get the same result by running:

```
In [16]: [df.index, df.columns]
```

However, it is helpful to think of a `DataFrame` as having two dimensions or two *axes*: a horizontal dimension, which goes left to right and is called the *columns axis*; and a vertical dimension that goes top to bottom and is called the *rows axis*.

Some `DataFrame` methods allow you to specify an `axis='rows'` or `axis='columns'` keyword argument that tells `pandas` how to run the method. For example, you can use the `DataFrame count` method to find out how many non-empty (i.e., non-`NaN`) values there are in each of `df`'s columns:

```
In [17]: df.count(axis='rows')
```

```
Out [17]: first_column    3
second_column     3
third_column      1
dtype: int64
```

Similarly, you can use the same method to find out how many non-empty values there are in each of `df`'s rows with:

```
In [18]: df.count(axis='columns')
```

```
Out [18]: first_row      2
second_row     3
third_row      2
dtype: int64
```

At first, it may seem confusing that if you pass the `axis='rows'` keyword argument, the result counts non-empty values in each *column*. However, the `axis` argument specifies which `DataFrame` axis the method is *applied on* — I remember this by thinking that the `axis` argument tells `pandas` which `DataFrame` dimension to “*squash*”, leaving the other `DataFrame` dimension unchanged. There will be many examples of using `axis` in the following chapters, so don't worry if it's not clear how to use it yet.

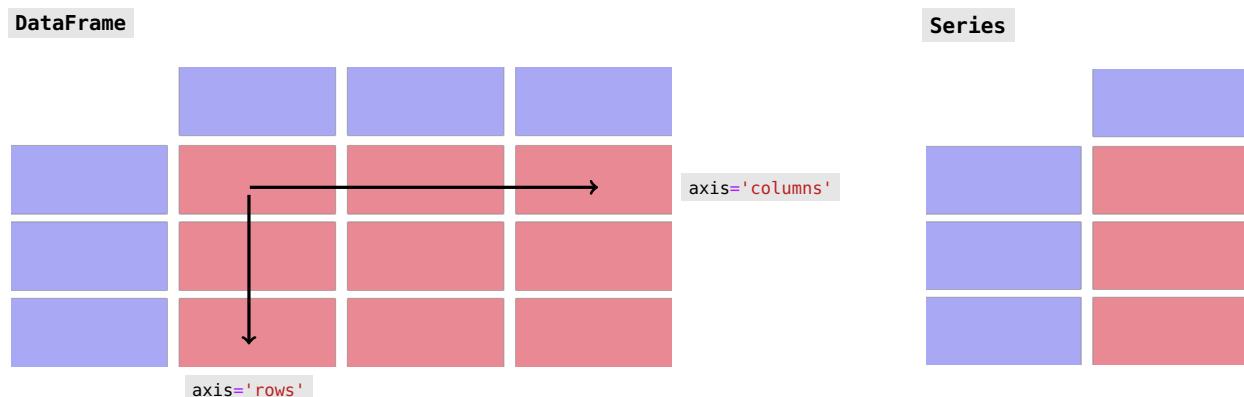


Figure 13.1: Visual representation of a pandas `DataFrame` (on the left) and of a pandas `Series` (on the right). Data values are shown in pink, whereas column and row labels are shown in blue. In both figures, blue boxes at the top represent the column `Index` and blue boxes on the left represent the row `Index`.

Values and types

If you work with Excel, you probably know that you can change column or cell type through the “Format” menu.⁷ Figure 13.2 below shows Excel’s cell formatting dialog, with the different data types you can choose from.

7: At least in Microsoft Excel 16.

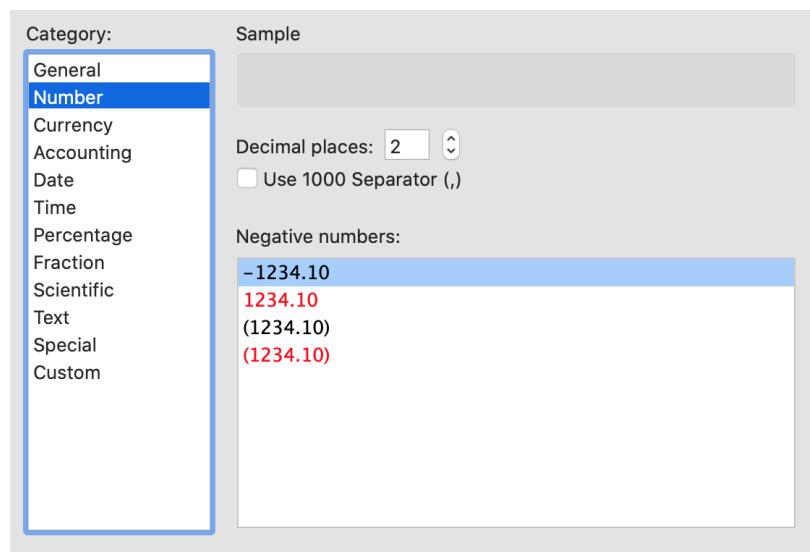


Figure 13.2: Cell formatting dialog in Excel.

Similarly, `DataFrame` objects can store different data types: numbers (whole or decimal), text, dates, and others. You can check what type of data is in each `DataFrame` column with the `info` method:

In [19]: `df`

Out [19]:

	first_column	second_column	third_column
first_row	3	0.23	NaN
second_row	-20	2.00	Assets
third_row	121	-3.50	NaN

In [20]: `df.info()`

```
Out [20]: <class 'pandas.core.frame.DataFrame'>
Index: 3 entries, first_row to third_row
Data columns (total 3 columns):
 #   Column      Non-Null Count  Dtype  
---  --  
 0   first_column    3 non-null      int64  
 1   second_column   3 non-null      float64 
 2   third_column    1 non-null      object  
dtypes: float64(1), int64(1), object(1)
memory usage: 96.0+ bytes
```

The output above has several pieces of information about your table: how many rows it has (i.e., `Index: 3 entries`), their labels (i.e., `first_row to third_row`), how many columns (i.e., `Data columns (total 3 columns)`) and for each column, its name, the number of non-empty values (under `Non-Null Count`) and the type of data it stores (under `Dtype`, for *data type*). In this example, the first column stores whole numbers (i.e., integers), the second column stores decimal numbers (i.e., floats), and the last column has a mix of missing values (i.e., `NaN`) and string values. The `object` type is a generic data type that pandas uses when there are different types of values in the same column.

The last two lines in the output above show how many columns there are with each data type, and how much of your computer's memory is used to store the table (in this case, not much).

In its internal code, pandas uses complex machinery to make data operations efficient⁸ — which is why in the output above under the `Dtype` header you see slightly different type keywords than Python's built-in `int` and `float`. Even so, for all practical purposes, these data types are identical to the standard Python types you learned about earlier in the book.

8: Faster and use less memory.

You won't use these pandas-specific type keywords anywhere in your code, however, they do show up in the output of various pandas methods, so it's useful to know what they mean (for instance, they show up in the output of `info`). You will sometimes even see variations of these keywords (e.g., `int32`, `int64`, or `Int64` are all pandas type indicators for integers). While there are subtle differences between these variations, all you need to remember is that there's nothing special about pandas's data types; they work just like Python's built-in types.

Table 13.1: pandas data types and their Excel equivalents. These (and more) types are described in the official pandas documentation at pandas.pydata.org/pandas-docs/stable/user_guide/basics.html#basics-dtypes.

Type	Alias	Excel equivalent	Description
Float	<code>float64</code>	Number	Columns storing decimal numbers (i.e., floating-point numbers).

Table 13.1: pandas data types and their Excel equivalents. These (and more) types are described in the official pandas documentation at pandas.pydata.org/pandas-docs/stable/user_guide/basics.html#basics-dtypes.

Type	Alias	Excel equivalent	Description
Integer	int32, int64, Int32, Int64, Int32, Int64	Number	Columns storing whole numbers (i.e., positive or negative integers).
Boolean	bool, boolean	-	Columns storing <code>True</code> or <code>False</code> values.
Datetime	datetime64	Date, Time	Column storing dates, times or both.
Category	category	-	Column storing a limited (and usually fixed) number of possible values (e.g., one of red, blue or green).
String	object, string	Text	Columns storing any kind of text.
Object	object	General	Generic type for any value not covered by the types above, or for columns that have mixed values (i.e., numbers and text in the same column).

Not all of Excel’s data types have equivalents in pandas. Many of Excel’s cell formatting options are related to the way values are *displayed*, rather than what they *are*. For instance, Excel’s “Currency”, “Fraction”, or “Percentage” cell formats change the way values get shown on your screen, but not the way your computer stores them. In pandas, when you change a column’s data type, your computer stores its values differently. However, even in pandas you can change the display of DataFrame values to get them to look exactly the way you want them to (e.g., with currency symbols in front or a percentage sign at the end). We’ll go over changing the display of DataFrame columns at the end of this part of the book.

Column data types are consequential because depending on what type is assigned to a column, different methods or operations are possible with its values. For example, you can compute the sum of values in `df`’s first column with:

```
In [21]: df['first_column'].sum()
```

```
Out [21]: 104
```

But if you change⁹ the column data type to `'string'`, you can’t compute the sum anymore:

```
In [22]: df['first_column'].astype('string').sum()
```

```
Out [22]:
```

```
-----  
TypeError                                     Traceback (most recent call last)  
<ipython-input-11-f8a05a91172c> in <module>  
----> 1 df['first_column'].astype('string').sum()  
...  
TypeError: Cannot perform reduction 'sum' with string dtype
```

⁹: More on changing column types in the following chapters.

When you create a DataFrame, either by reading an Excel spreadsheet or manually, as we did in the example above, pandas tries to figure out what type of data is stored in each column. At times, it

can't figure out the right types because there's something wrong with the data themselves (e.g., numbers are stored with currency symbols), and it assigns the most generic data type to columns (i.e., the '`object`' type). In that case, some methods might not work as you expect them to, and you will need to clean the data and assign types yourself — something we'll look at later.

Summary

This chapter introduced `pandas`'s workhorses: the `DataFrame`, `Series` and `Index` objects. These objects are how tables and columns get represented in Python code and are similar to the Excel tables and columns you're already familiar with.

The following chapter goes over the `pandas` methods you can use to read and write Excel files.

Reading and writing Excel files

Now that you know how to represent tables in Python code let's take a look at some of the ways you can read and write data from or to Excel files using `pandas`.

Remember that you need to `import pandas` in your Jupyter notebook to access its functions:

```
In [1]: import pandas as pd
```

Reading Excel files

You probably work with spreadsheets¹ often. `pandas` comes with a `read_excel` function to read data from a spreadsheet into a `DataFrame` variable — you've already seen it in action earlier, in our brief tour of `pandas`. To read data from the “`Q1Sales.xlsx`” file into a `DataFrame`, you can use:

```
In [2]: ledger_df = pd.read_excel('Q1Sales.xlsx')
```

This one-liner returns a `DataFrame` object containing all data from the first sheet in “`Q1Sales.xlsx`” and assigns it the `ledger_df` name. Inspect `ledger_df` in a separate cell to see for yourself:

```
In [3]: ledger_df
```

```
Out [3]:   InvoiceNo      Channel       Product Name ... Unit Price Quantity  Total
0        1532  Shoppe.com    Cannon Water Bomb ...    ...  20.11      14  281.54
1        1533      Walcart  LEGO Ninja Turtles...    ...   6.70       1   6.70
2        1534     Bullseye           NaN    ...    ...  11.67       5  58.35
3        1535     Bullseye  Transformers Age o...    ...  13.46       6  80.76
4        1535     Bullseye  Transformers Age o...    ...  13.46       6  80.76
...
14049     15581     Bullseye  AC Adapter/Power S...    ...  28.72       8 229.76
14050     15582     Bullseye  Cisco Systems Giga...    ...  33.39       1  33.39
14051     15583  Understock.com  Philips AJ3116M/37...    ...   4.18       1   4.18
14052     15584      iBay.com           NaN    ...    ...  4.78      25 119.50
14053     15585  Understock.com    Sirius Satellite R...    ...  33.16       2  66.32

[14054 rows x 12 columns]
```

By default, `read_excel` reads data only from the first sheet of an Excel file. If you want to read data from any other sheet in the file, you can pass the name of the sheet you want to read as the `sheet_name` argument to `read_excel`:

```
In [4]: ledger_df = pd.read_excel('Q1Sales.xlsx', sheet_name='March')
```

1: Whether from an Excel file or any other file that has one of the following extensions: `.xls`, `.xlsx`, `.xlsm`, `.xlsb`, or `.odf`.

	InvoiceNo	Channel	Product Name	Unit Price	Quantity	Total	
0	29486	Walcart	Vic Firth American...	22.39	6	134.34	
1	29487	Walcart	Archives Spiral Bo...	25.65	1	25.65	
2	29488	Bullseye	AKG WMS40 Mini Dua...	8.98	2	17.96	
3	29489	Shoppe.com	LE Blue Case for A...	8.33	9	74.97	
4	29490	Understock.com	STARFISH Cookie Cu...	17.96	80	1436.80	
...	
9749	39235	iBay.com	Nature's Bounty Ga...	5.55	2	11.10	
9750	39216	Shoppe.com	Funko Wonder Woman...	28.56	1	28.56	
9751	39219	Shoppe.com	MONO GS1 GS1-BTY-B...	3.33	1	3.33	
9752	39238	Shoppe.com		NaN	34.76	10	347.60
9753	39239	Understock.com	3 Collapsible Bowl...	6.39	15	95.85	

[9754 rows x 12 columns]

If you need to read data from different sheets and keep those data as separate variables, you can use `read_excel` multiple times:

```
In [5]: jan_ledger_df = pd.read_excel('Q1Sales.xlsx')  
feb_ledger_df = pd.read_excel('Q1Sales.xlsx', sheet_name='February')  
mar_ledger_df = pd.read_excel('Q1Sales.xlsx', sheet_name='March')
```

You won't always remember the actual sheet names in your Excel files, but you might remember their position in the file. In that case, you can specify their position instead of their names in the value you pass to `sheet_name`:

```
In [6]: ledger_df = pd.read_excel('Q1Sales.xlsx', sheet_name=0)
```

Notice that even sheet positions start at 0 — counting starts at 0 in everything you do with Python code.

Besides `sheet_name`, `read_excel` allows you to specify several other keyword arguments that control what data is read from an Excel file. Table 14.1 lists some of the most useful ones, with examples.

Table 14.1: Keyword arguments available with pandas's `read_excel` function. You can use any combination of keyword arguments when reading data, as needed.

Parameter name	Example	Description
<code>sheet_name</code>	<code>pd.read_excel('Q1Sales.xlsx', sheet_name='March')</code>	Reads all data from a sheet named "March" in "Q1Sales.xlsx".
<code>header</code>	<code>pd.read_excel('Q1Sales.xlsx', header=2)</code>	Uses values in row 2 of the spreadsheet as DataFrame column names, skipping all rows above. Useful when data in your spreadsheet doesn't start on the first row.
	<code>pd.read_excel('Q1Sales.xlsx', header=[3, 4])</code>	Uses rows 3 and 4 as DataFrame column names (the columns will have multiple levels), skips all rows above.
<code>names</code>	<code>pd.read_excel('Q1Sales.xlsx', names=['Date', 'Total Sales'])</code>	Ignores the header row and uses the column names passed here. If there is no header row in the spreadsheet, you can pass <code>header=None</code> and a list of column names to use instead.

Table 14.1: Keyword arguments available with pandas's `read_excel` function. You can use any combination of keyword arguments when reading data, as needed.

Parameter name	Example	Description
<code>usecols</code>	<code>pd.read_excel('Q1Sales.xlsx', usecols=['A', 'B', 'D:G'])</code>	Reads only columns <i>A</i> , <i>B</i> and <i>D</i> through <i>G</i> (inclusive) of the first spreadsheet in "Q1Sales.xlsx".
	<code>pd.read_excel('Q1Sales.xlsx', usecols=[0, 1, 3:6])</code>	Reads only columns 1, 2 and 4 through 7 of the spreadsheet (same as above).

You can use other pandas functions to read tabular data from different kinds of files, not just spreadsheets — `read_csv` might be useful if you work with CSV files often.² You can find the other pandas data-reading functions with the autocomplete feature in JupyterLab by typing `pd.read_` in a separate cell and pressing the TAB key:

In [7]: `pd.read_<TAB>`

I want to clarify that when you read data from a spreadsheet³ with `read_excel`, you make a temporary copy of all data in your spreadsheet. This copy is stored in your computer's working memory (i.e., its random access memory or RAM). The `ledger_df` variable you created above is a copy of your data. Anything you do with the `DataFrame` variable is not saved on your computer drive (and doesn't change any data in your original spreadsheet) until you explicitly write the `DataFrame` back to a file — we'll take a look at how to do that later in this chapter.⁴

When you use `read_excel` or any of the other `read_` functions, a couple of things happen behind the scenes. pandas sets up the row and column indexes for your `DataFrame`, so that data access and manipulation is fast; it also tries to figure out what type of data is stored in each `DataFrame` column to enable the appropriate methods for each.

Now that you have the sales data in a `DataFrame`, you can start slicing, selecting, and sorting it any way you want to; but first, let's take a closer look at this dataset.

Inspecting data

You already know how to view a `DataFrame` variable: type its name in a cell and run it. By default, this will display the top and bottom five rows of your `DataFrame`.⁵

In [8]: `ledger_df`

2: The others are `read_html`, `read_spss`, `read_sas`.

3: Or any other data file.

4: Excel works the same way, but doesn't make it explicit that you are working on a temporary copy of the data. When Excel crashes, it sometimes doesn't write your changes on the computer's disk and its temporary copy of your work gets lost forever.

5: Later, we will go over changing the default number of rows shown when inspecting a `DataFrame` variable, if you want to see more or fewer rows.

```
Out [8]:
```

	InvoiceNo	Channel	Product Name	Unit Price	Quantity	Total
0	1532	Shoppe.com	Cannon Water Bomb ...	20.11	14	281.54
1	1533	Walcart	LEGO Ninja Turtles...	6.70	1	6.70
2	1534	Bullseye	Nan	11.67	5	58.35
3	1535	Bullseye	Transformers Age o...	13.46	6	80.76
4	1535	Bullseye	Transformers Age o...	13.46	6	80.76
...
14049	15581	Bullseye	AC Adapter/Power S...	28.72	8	229.76
14050	15582	Bullseye	Cisco Systems Giga...	33.39	1	33.39
14051	15583	Understock.com	Philips AJ3116M/37...	4.18	1	4.18
14052	15584	iBay.com	Nan	4.78	25	119.50
14053	15585	Understock.com	Sirius Satellite R...	33.16	2	66.32

[14054 rows x 12 columns]

When the displayed table is truncated (i.e., when your table has more than 10 rows), the last line in the output tells you how many rows and columns there are in total.

If you want to take a quick look at your data, there are two `DataFrame` methods you can use to display the top or bottom rows: `head` and `tail`. For example, to display just the first 3 rows in your `DataFrame`, you can use the `head` method:

```
In [9]: ledger_df.head(3)
```

```
Out [9]:
```

	InvoiceNo	Channel	Product Name	Unit Price	Quantity	Total
0	1532	Shoppe.com	Cannon Water Bomb Ba...	20.11	14	281.54
1	1533	Walcart	LEGO Ninja Turtles S...	6.70	1	6.70
2	1534	Bullseye	Nan	11.67	5	58.35

If you want to see more of the table, you can pass a larger number as an argument to `head` (e.g., `head(20)` to see the top 20 rows). Similarly, to display the last 3 rows, you can use:

```
In [10]: ledger_df.tail(3)
```

```
Out [10]:
```

	InvoiceNo	Channel	Product Name	Unit Price	Quantity	Total
14051	15583	Understock.com	Philips AJ3116M/37...	4.18	1	4.18
14052	15584	iBay.com	Nan	4.78	25	119.50
14053	15585	Understock.com	Sirius Satellite R...	33.16	2	66.32

[3 rows x 12 columns]

The number 3 passed as an argument to both methods above tells `pandas` how many rows to return. By default, if you do not pass a number, both methods will return the first or last 5 rows in your `DataFrame`. These methods *return* a `DataFrame`, so you can even assign their output to another variable if, for example, you want to work with just the top 3 rows in your table.

Another useful `DataFrame` method that allows you to examine data quickly is `info`: it gives you a `DataFrame` summary, including how many rows and columns there are in your table, some information about the values in each of its columns (how many empty values

there are in each column, what type of values they hold), and how much memory the entire table uses up.

```
In [11]: ledger_df.info()
```

```
Out [11]: <class 'pandas.core.frame.DataFrame'>
RangeIndex: 14054 entries, 0 to 14053
Data columns (total 12 columns):
 #   Column      Non-Null Count  Dtype  
--- 
 0   InvoiceNo   14054 non-null   int64  
 1   Channel     14054 non-null   object  
 2   Product Name 12362 non-null   object  
 3   ProductID   14054 non-null   object  
 4   Account     14054 non-null   object  
 5   AccountNo   14054 non-null   int64  
 6   Date        14054 non-null   datetime64[ns]
 7   Deadline    14054 non-null   object  
 8   Currency    14054 non-null   object  
 9   Unit Price  14054 non-null   float64 
 10  Quantity    14054 non-null   int64  
 11  Total       14054 non-null   float64 
dtypes: datetime64[ns](1), float64(2), int64(3), object(6)
memory usage: 1.3+ MB
```

I briefly mentioned the `info` method in our quick tour of pandas. Using `info` is a great way to get a quick overview of your data, without manually checking every column.

One thing to notice in the output of `info` is that columns containing text values are listed as having the `object` type, instead of the `string` type (e.g., `Product Name`). Columns with the `object` type often contain `string` values — as you will see in the next chapters, even though they are listed as `object` columns, you can work with their values as you would with any Python string.⁶

Finally, when you need to check how many rows and columns there are in your `DataFrame`, you can inspect its `shape` attribute:

```
In [12]: ledger_df.shape
```

```
Out [12]: (14054, 12)
```

The `shape` attribute is a Python *tuple* containing the number of rows and columns in your table. It is a `DataFrame` attribute (instead of a `DataFrame` method) because it is part of the `DataFrame` object, rather than computed when you call a method.

6: The reason they are listed as `object` columns instead of `string` columns is that pandas has only recently introduced the `string` data type and not all of its functions have changed to use it yet.

Writing Excel files

Like the `read_excel` function reads data from Excel files, pandas `DataFrame` objects have a `to_excel` method that writes their contents to an Excel file. For example, to write `ledger_df` to an

A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	InvoiceNo	Channel	Product Name	ProductID	Account	AccountNo	Date	Deadline	Currency	Unit Price	Quantity	Total	
0	1532	Shoppe.com	Cannon Water	T&G/CAN-97509	Sales	5004	2020-01-01 00:00:00	2020-06-05 00:00:00	USD	20.11	14	281.54	
1	1533	Walcart	LEGO Ninja Tur	T&G/LEG-37777	Sales	5004	2020-01-01 00:00:00	2019-11-12 00:00:00	USD	6.7	1	6.7	
2	1534	Bullseye	T&G/PET-14209	Sales	5004	2020-01-01 00:00:00	2020-05-28 00:00:00	USD	11.67	5	58.35		
3	1535	Bullseye	Transformers A	T&G/TRA-20170	Sales	5004	2020-01-01 00:00:00	2020-04-08 00:00:00	USD	13.46	6	80.76	
4	1536	Understock.com	Ty Beanie Boos	T&G/TY-67653	Sales	5004	2020-01-01 00:00:00	2020-07-14 00:00:00	USD	27.89	28	780.92	
5	1537	Understock.com	Swimways Poo	T&G/SWI-41388	Sales	5004	2020-01-01 00:00:00	2019-12-01 00:00:00	USD	24.11	13	313.43	
6	1538	Shoppe.com	Fantasy Flight	T&G/FAN-71205	Sales	5004	2020-01-01 00:00:00	2019-12-31 00:00:00	USD	9.67	3	29.01	
7	1539	Bullseye	Fantasy Flight C	T&G/FAN-71205	Sales	5004	2020-01-01 00:00:00	2020-05-09 00:00:00	USD	9.59	1	9.59	
8	1540	Shoppe.com	Hape Little Piai	T&G/HAP-00999	Sales	5004	2020-01-01 00:00:00	2020-05-03 00:00:00	USD	15.14	5	75.7	

Figure 14.1: Screenshot of “JanQ1Sales.xlsx” opened in Excel. Notice that `ledger_df`’s row labels are now column A of the output spreadsheet. Using the `index=False` keyword argument with `to_excel` removes them from the final spreadsheet.

Excel file called “`JanQ1Sales.xlsx`” in the same folder as your Jupyter notebook, you can use:

```
In [13]: ledger_df.to_excel('JanQ1Sales.xlsx')
```

This method doesn’t return anything, so there’s no output underneath the cell, but once it finishes what it’s doing, you should see a new Excel file show up in your JupyterLab file navigator (on the left of your screen). That’s how you turn `DataFrame` variables into Excel files.

You can also use a full path⁷ instead of a file name if you want to store your Excel file in a specific folder:

```
In [14]: ledger_df.to_excel('C:/Users/Horatio/Documents/Python for Accounting/JanQ1Sales.xlsx')
```

The file path above uses forward slashes instead of the more familiar backslashes used in Windows file paths. If you find using forward slashes annoying, you can also write file paths using backslashes, but you’ll have to put an `r` in front of them:

```
In [15]: ledger_df.to_excel(r'C:\Users\Horatio\Documents\Python for Accounting\JanQ1Sales.xlsx')
```

Both `read_excel` and `to_excel` work with full paths. Keep in mind that `to_excel` will overwrite any existing file with the same name and path as the one you specify without asking twice, so make sure you don’t overwrite anything important.

You can also specify the name of the sheet to write to, if you pass a value as the `sheet_name` argument when using `to_excel`:

```
In [16]: ledger_df.to_excel('JanQ1Sales.xlsx', sheet_name='Sales')
```

This will write all data in `ledger_df` to a single⁸ sheet named “`Sales`” in “`JanQ1Sales.xlsx`”. If you open the new file with Excel, you should see something similar to figure 14.1.

7: The same goes for `read_excel`, if you want to read an Excel file at a specific path on your drive.

8: How to write data to multiple sheets in the same Excel file will have to wait for the first project chapter.

Notice in figure 14.1 that `ledger_df`'s row labels are part of the spreadsheet (column A). Row labels are not particularly useful in Excel, so you can exclude them from the spreadsheet with:

```
In [17]: ledger_df.to_excel('JanQ1Sales.xlsx', sheet_name='Sales', index=False)
```

The `index=False` keyword argument above tells pandas not to write row labels in the output spreadsheet. Some of the other keyword arguments you can use with `to_excel` are listed in table 14.2 below.

Table 14.2: Keyword arguments available with pandas's `to_excel` function. You can use any combination of keyword arguments when writing data.

Parameter name	Example	Description
<code>sheet_name</code>	<code>ledger_df.to_excel('JanQ1Sales.xlsx', sheet_name='Sales')</code>	Writes all data from <code>ledger_df</code> to a sheet named "Sales" in "JanQ1Sales.xlsx".
<code>index</code>	<code>ledger_df.to_excel('JanQ1Sales.xlsx', index=False)</code>	Writes data from <code>ledger_df</code> , without its row labels, to a sheet named "Sheet1" in "JanQ1Sales.xlsx".
<code>columns</code>	<code>ledger_df.to_excel('JanQ1Sales.xlsx', columns=['ProductID', 'Total'])</code>	Writes the 'ProductID' and 'Total' columns from <code>ledger_df</code> to a sheet named "Sheet1" in "JanQ1Sales.xlsx".
<code>startrow</code>	<code>ledger_df.to_excel('JanQ1Sales.xlsx', startrow=2)</code>	Writes all data from <code>ledger_df</code> to a sheet named "Sheet1" in "JanQ1Sales.xlsx", but leaves the first row empty (i.e., data will start at row 2 in the spreadsheet).
<code>startcol</code>	<code>ledger_df.to_excel('JanQ1Sales.xlsx', startcol=2)</code>	Writes all data from <code>ledger_df</code> to a sheet named "Sheet1" in "JanQ1Sales.xlsx", but leaves the first column empty (i.e., data will start at column B in the spreadsheet).

If you need to write data to a different type of file, other than an Excel spreadsheet, you can use one of pandas's many `DataFrame` `to_` methods — many of which accept similar keyword arguments as `to_excel`. For example, to write `ledger_df` to a CSV file instead of an Excel spreadsheet, you can use:⁹

```
In [18]: ledger_df.to_csv('JanQ1Sales.csv', index=False)
```

As I mentioned earlier for `read_excel`, you can find all the other `DataFrame` writing methods on your own, using the autocomplete feature in JupyterLab, by typing `ledger_df.to_` in a separate cell and pressing the TAB key:

```
In [19]: ledger_df.to_<TAB>
```

This will open an autocomplete menu, and show you all the `DataFrame` methods you can use for writing data to a file. You

9: Notice that in the file name, you have to type the file extension yourself (e.g., .csv or .xlsx).

might not recognize many of the methods listed — but if you see a method that seems useful, you can always bring up its documentation and read more about what it does by selecting it from the autocomplete menu, adding a question mark after its name, and running that in a separate cell. For example, to learn more about what `to_clipboard` does, you can run:

```
In [20]: ledger_df.to_clipboard?
```

The same goes for any of the methods or functions we've already covered. Whenever you get stuck, use JupyterLab's autocomplete feature or the documentation helper (i.e., the `?` at the end of a method name) to find help.

The `pandas` library has a lot of functions, each with several arguments that control how they work. Remembering all of them is impossible — fortunately, it isn't necessary either: use the autocomplete menu, look up documentation, or search the web to figure out the details whenever you need to.

Summary

This chapter showed you how to read and write Excel files using `pandas`'s `read_excel` and `to_excel` functions. Reading data from an Excel spreadsheet into a `pandas DataFrame` creates a copy of the spreadsheet data — when you finish working on your data, you can write the `DataFrame` variable back to an Excel file. However, everything you do between reading and writing data leaves a code trace in your notebook. If you don't save your `DataFrame` to a file, you can always return to your code and re-run it.

Reading and writing data from and to Excel files is great, but you probably want to do something with those data in between. Let's take a look at how you can slice, filter, and sort tables next.

Slicing, filtering, and sorting tables

15

What's the first thing you do when you start working with a new dataset in Excel? You probably copy the few columns you need for your task in a separate "working" sheet or delete the columns you don't need. You then put filters on columns (i.e., you put your data in a table) to enable quick filtering and perhaps sort the entire table differently.

This chapter walks you through a similar workflow using pandas: selecting or removing columns from a DataFrame, filtering DataFrame rows using conditional statements, and sorting a DataFrame by the values in one or more of its columns.

To run the code examples in this chapter, you first need to `import pandas` and read the sales data:

```
In [1]: import pandas as pd  
  
ledger_df = pd.read_excel('Q1Sales.xlsx')
```

Selecting columns

To select all values in a DataFrame column, you need to type the DataFrame variable name followed by a pair of square brackets and a column name (in quotes) inside.¹ For example, to select the 'Product Name' column from `ledger_df`:

```
In [2]: ledger_df['Product Name']  
  
Out [2]: 0 Cannon Water Bomb Balloons 100 Pack  
1 LEGO Ninja Turtles Stealth Shell in Pursuit 79102  
2 NaN  
3 Transformers Age of Extinction Generations Del...  
4 Transformers Age of Extinction Generations Del...  
...  
14049 AC Adapter/Power Supply&Cord for Lenovo 3000 G...  
14050 Cisco Systems Gigabit VPN Router (RV320K9NA)  
14051 Philips AJ3116M/37 Digital Tuning Clock Radio ...  
14052 NaN  
14053 Sirius Satellite Radio XADH2 Home Access Kit f...  
Name: Product Name, Length: 14054, dtype: object
```

¹: This notation is identical to accessing values in a Python dictionary using their key.

The same kind of code can be used to access values from any other column in `ledger_df` by name:

```
In [3]: ledger_df['Unit Price']
```

```
Out[3]: 0      20.11
       1      6.70
       2     11.67
       3     13.46
       4     13.46
       ...
      14049    28.72
      14050    33.39
      14051    4.18
      14052    4.78
      14053    33.16
Name: Unit Price, Length: 14054, dtype: float64
```

In both examples, the output is a `pandas Series` object. The last line in the output above tells you the column name, its length (i.e., the number of rows it has), and its data type — in this case, '`Unit Price`' has 14 054 float values (i.e., decimal numbers). Like `DataFrame` objects, `Series` have their own methods and attributes that you'll uncover more-and-more as we progress. For example, you can calculate the median unit price in these sales data using:

```
In [4]: ledger_df['Unit Price'].median()
```

```
Out[4]: 10.22
```

A common error when selecting columns by name is `KeyError`. When selecting columns, make sure you type their names exactly — if you leave a space at the end of the column name or make any typo, you'll get the following error message:

```
In [5]: ledger_df['Unit Price ']
```

```
Out[5]: KeyError                               Traceback (most recent call last)
<ipython-input-80-ad5494a8805a> in <module>
...-> 1 ledger_df['Unit Price ']
...
KeyError: 'Unit Price '
```

When you want to select more than one column from a `DataFrame` you can use a Python list with all the column names you want instead of a single name:

```
In [6]: ledger_df[['ProductID', 'Product Name', 'Unit Price', 'Total']]
```

```
Out[6]:   ProductID      Product Name  Unit Price  Total
0  T&G/CAN-97509  Cannon Water Bomb ...    20.11  281.54
1  T&G/LEG-37777    LEGO Ninja Turtles...     6.70    6.70
2  T&G/PET-14209                  NaN    11.67  58.35
3  T&G/TRA-20170  Transformers Age o...    13.46  80.76
4  T&G/TRA-20170  Transformers Age o...    13.46  80.76
...
          ...
14049    E/AC-63975  AC Adapter/Power S...    28.72 229.76
14050    E/CIS-74992  Cisco Systems Giga...    33.39  33.39
14051    E/PHI-08100  Philips AJ3116M/37...     4.18    4.18
14052    E/POL-61164                  NaN     4.78 119.50
14053    E/SIR-83381  Sirius Satellite R...    33.16   66.32
```

```
[14054 rows x 4 columns]
```

Notice the two sets of square brackets that surround the column names. The first set of brackets is just how you access columns in a `DataFrame`, whereas the second set of brackets defines a Python list of column names. If you need the list of column names elsewhere in your code, you can store it as a separate variable and use it to select column in the same way:

```
In [7]: column_names = ['ProductID', 'Product Name', 'Unit Price', 'Total']  
1  
2  
3  
ledger_df[column_names]
```

```
Out [7]:      ProductID      Product Name  Unit Price  Total  
0    T&G/CAN-97509  Cannon Water Bomb ...    20.11  281.54  
1    T&G/LEG-37777  LEGO Ninja Turtles...    6.70   6.70  
2    T&G/PET-14209                      NaN  11.67  58.35  
3    T&G/TRA-20170  Transformers Age o...  13.46  80.76  
4    T&G/TRA-20170  Transformers Age o...  13.46  80.76  
...       ...       ...       ...  
14049   E/AC-63975  AC Adapter/Power S...  28.72  229.76  
14050   E/CIS-74992  Cisco Systems Giga...  33.39  33.39  
14051   E/PHI-08100  Philips AJ3116M/37...  4.18   4.18  
14052   E/POL-61164                      NaN  4.78  119.50  
14053   E/SIR-83381  Sirius Satellite R...  33.16  66.32
```

```
[14054 rows x 4 columns]
```

The output above is another `DataFrame`: you can assign it to a variable if you want to, or even assign it back to `ledger_df` if you're going to discard all the other columns in your table.

```
In [8]: products_df = ledger_df[['ProductID', 'Product Name', 'Unit Price', 'Total']]  
  
products_df
```

```
Out [8]:      ProductID      Product Name  Unit Price  Total  
0    T&G/CAN-97509  Cannon Water Bomb ...    20.11  281.54  
1    T&G/LEG-37777  LEGO Ninja Turtles...    6.70   6.70  
2    T&G/PET-14209                      NaN  11.67  58.35  
3    T&G/TRA-20170  Transformers Age o...  13.46  80.76  
4    T&G/TRA-20170  Transformers Age o...  13.46  80.76  
...       ...       ...       ...  
14049   E/AC-63975  AC Adapter/Power S...  28.72  229.76  
14050   E/CIS-74992  Cisco Systems Giga...  33.39  33.39  
14051   E/PHI-08100  Philips AJ3116M/37...  4.18   4.18  
14052   E/POL-61164                      NaN  4.78  119.50  
14053   E/SIR-83381  Sirius Satellite R...  33.16  66.32
```

```
[14054 rows x 4 columns]
```

You will often forget column names (I do); remember you can list all column names by running:

```
In [9]: ledger_df.columns
```

```
Out [9]: Index(['InvoiceNo', 'Channel', 'Product Name', 'ProductID', 'Account',
   'AccountNo', 'Date', 'Deadline', 'Currency', 'Unit Price', 'Quantity',
   'Total'],
  dtype='object')
```

The `columns` attribute gives you access to the DataFrame `Index` object used to store column names. Because the `Index` object is a list-like structure, you can use the output above just like you would any regular Python list (e.g., slice or loop over it using a `for` loop). For example, to select the first three² columns in `ledger_df`:

```
In [10]: ledger_df.columns[:3]
```

```
Out [10]: Index(['InvoiceNo', 'Channel', 'Product Name'], dtype='object')
```

```
In [11]: ledger_df[ledger_df.columns[:3]]
```

```
Out [11]:      InvoiceNo        Channel       Product Name
0            1532  Shoppe.com  Cannon Water Bomb ...
1            1533      Walcart  LEGO Ninja Turtles...
2            1534     Bullseye                  NaN
3            1535     Bullseye  Transformers Age o...
4            1535     Bullseye  Transformers Age o...
...
14049        15581     Bullseye  AC Adapter/Power S...
14050        15582     Bullseye  Cisco Systems Giga...
14051        15583  Understock.com  Philips AJ3116M/37...
14052        15584      iBay.com                  NaN
14053        15585  Understock.com  Sirius Satellite R...
[14054 rows x 3 columns]
```

2: Remember that slicing a Python list is non-inclusive on the right side (i.e., after the colon).

You can even use list comprehensions to select just the columns that contain a specific keyword in their name or satisfy some other condition — for example, to select all columns that have the word '`Product`' in their name:

```
In [12]: [name for name in ledger_df.columns if 'Product' in name]
```

```
Out [12]: ['Product Name', 'ProductID']
```

```
In [13]: ledger_df[[name for name in ledger_df.columns if 'Product' in name]]
```

```
Out [13]:      Product Name      ProductID
0    Cannon Water Bomb ...  T&G/CAN-97509
1    LEGO Ninja Turtles...  T&G/LEG-37777
2                  NaN  T&G/PET-14209
3  Transformers Age o...  T&G/TRA-20170
4  Transformers Age o...  T&G/TRA-20170
...
14049  AC Adapter/Power S...  E/AC-63975
14050  Cisco Systems Giga...  E/CIS-74992
14051  Philips AJ3116M/37...  E/PHI-08100
14052                  NaN  E/POL-61164
14053  Sirius Satellite R...  E/SIR-83381
```

```
[14054 rows x 2 columns]
```

In this last example, notice again the double square brackets, one set for accessing columns in a `DataFrame`, the other for creating a Python list comprehension (if you can't remember what those are exactly, see chapter 6 for a refresher).

Removing columns

Often after reading a large spreadsheet, you will want to keep only a few of its columns. If you know which columns you want to keep, you can select them using square brackets, as you saw earlier, and then assign them to another variable (or the same variable if you don't need the entire dataset):

```
In [14]: products_df = ledger_df[['ProductID', 'Product Name', 'Unit Price', 'Total']]
```

On the other hand, if you want to remove a few columns, it's easier to list the columns you want to remove rather than the ones you want to keep. In that case, you can define a list of columns you don't need anymore and then use the `drop` method:

```
In [15]: columns_to_remove = ['InvoiceNo', 'Account', 'AccountNo', 'Currency']
```

```
ledger_df.drop(columns_to_remove, axis='columns')
```

```
Out [15]:
```

	Channel	Product Name	ProductID	...	Unit Price	Quantity	Total	
0	Shoppe.com	Cannon Water Bom...	T&G/CAN-97509	...	20.11	14	281.54	
1	Walcart	LEGO Ninja Turtl...	T&G/LEG-37777	...	6.70	1	6.70	
2	Bullseye		NaN	T&G/PET-14209	...	11.67	5	58.35
3	Bullseye	Transformers Age...	T&G/TRA-20170	...	13.46	6	80.76	
4	Bullseye	Transformers Age...	T&G/TRA-20170	...	13.46	6	80.76	
...	
14049	Bullseye	AC Adapter/Power...	E/AC-63975	...	28.72	8	229.76	
14050	Bullseye	Cisco Systems Gi...	E/CIS-74992	...	33.39	1	33.39	
14051	Understock.com	Philips AJ3116M/...	E/PHI-08100	...	4.18	1	4.18	
14052	iBay.com		NaN	E/POL-61164	...	4.78	25	119.50
14053	Understock.com	Sirius Satellite...	E/SIR-83381	...	33.16	2	66.32	

```
[14054 rows x 8 columns]
```

Although you can't see the entire table, notice on the last line in the output above that it now has 8 columns instead of 12.

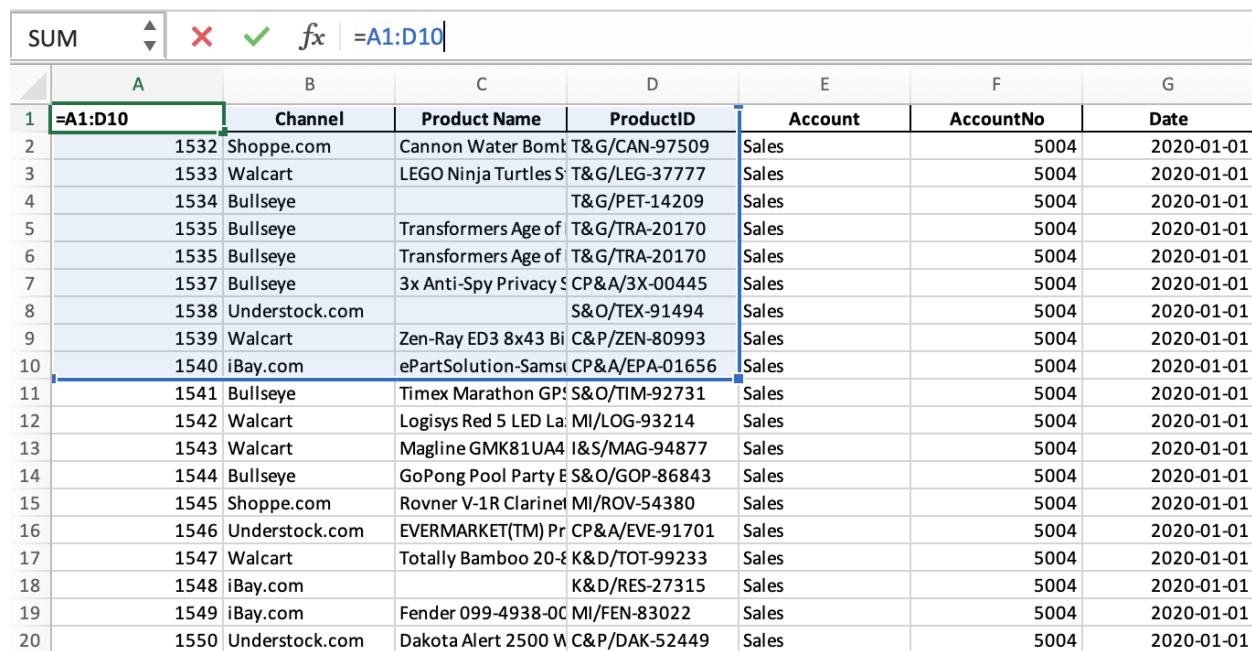
The additional argument passed to `drop, axis='columns'`, tells pandas that the labels to drop are column names. You can use the same method to remove rows from your `DataFrame`, by specifying a list of row labels rather than a list of column names and passing `axis='rows'` to the `drop` method.

Selecting rows and columns

In addition to selecting columns, pandas allows you to select both rows and columns from a `DataFrame` at the same time using the `loc` and `iloc` *indexing operators*. These operators enable you to select a subset of values from a `DataFrame` using either row and column *labels* (in the case of `loc`) or row and column *positions* in the `DataFrame` (in the case of `iloc`). Depending on your selection, the result of using these operators is either another `DataFrame`, a `Series` or a single value.

Using `loc`

Using the `loc` operator is similar to how you select cell ranges in Excel. Let's start with an Excel example and then translate that into pandas — figure 15.1 shows how you can select the top ten rows of the first four columns in “*Q1Sales.xlsx*” using Excel.



The screenshot shows a portion of an Excel spreadsheet titled "Q1Sales.xlsx". The formula bar at the top displays "=A1:D10". The table has columns labeled A through G. The first 10 rows are highlighted with a blue selection bar. Column A contains numerical IDs (1532 to 1550), Column B contains Channel names (Shoppe.com, Walcart, Bullseye, etc.), Column C contains Product Names (Cannon Water Bombs, LEGO Ninja Turtles, etc.), Column D contains Product IDs (T&G/CAN-97509, T&G/LEG-37777, etc.), Column E contains Account names (Sales), Column F contains Account numbers (5004), and Column G contains Dates (2020-01-01). The formula =A1:D10 is shown in the formula bar, indicating the range of cells selected.

	SUM	A	B	C	D	E	F	G
1	=A1:D10	Channel	Product Name	ProductID	Account	AccountNo	Date	
2	1532	Shoppe.com	Cannon Water Bombs	T&G/CAN-97509	Sales	5004	2020-01-01	
3	1533	Walcart	LEGO Ninja Turtles	T&G/LEG-37777	Sales	5004	2020-01-01	
4	1534	Bullseye		T&G/PET-14209	Sales	5004	2020-01-01	
5	1535	Bullseye	Transformers Age of	T&G/TRA-20170	Sales	5004	2020-01-01	
6	1535	Bullseye	Transformers Age of	T&G/TRA-20170	Sales	5004	2020-01-01	
7	1537	Bullseye	3x Anti-Spy Privacy	S CP&A/3X-00445	Sales	5004	2020-01-01	
8	1538	Understock.com		S&O/TEX-91494	Sales	5004	2020-01-01	
9	1539	Walcart	Zen-Ray ED3 8x43 Binoculars	C&P/ZEN-80993	Sales	5004	2020-01-01	
10	1540	iBay.com	ePartSolution-Samsung	CP&A/EPA-01656	Sales	5004	2020-01-01	
11	1541	Bullseye	Timex Marathon GPS Watch	S&O/TIM-92731	Sales	5004	2020-01-01	
12	1542	Walcart	Logisys Red 5 LED Lamp	MI/LOG-93214	Sales	5004	2020-01-01	
13	1543	Walcart	Magline GMK81UA4	I&S/MAG-94877	Sales	5004	2020-01-01	
14	1544	Bullseye	GoPong Pool Party Game	S&O/GOP-86843	Sales	5004	2020-01-01	
15	1545	Shoppe.com	Rovner V-1R Clarinet	MI/ROV-54380	Sales	5004	2020-01-01	
16	1546	Understock.com	EVERMARKET(TM) Pro	CP&A/EVE-91701	Sales	5004	2020-01-01	
17	1547	Walcart	Totally Bamboo 20-L	K&D/TOT-99233	Sales	5004	2020-01-01	
18	1548	iBay.com		K&D/RES-27315	Sales	5004	2020-01-01	
19	1549	iBay.com	Fender 099-4938-00	MI/FEN-83022	Sales	5004	2020-01-01	
20	1550	Understock.com	Dakota Alert 2500 Wireless	C&P/DAK-52449	Sales	5004	2020-01-01	

Figure 15.1: Selecting the top ten rows of the first four column in “*Q1Sales.xlsx*” using Excel.

The Excel formula used to select a cell range above is `=A1:D10`. In this formula, the letters reference table columns, whereas the numbers reference table rows; put together, they reference a range of values across both dimensions of the table.

In pandas, you can make the same selection using the `loc` operator. You use `loc` like a regular `DataFrame` method by typing the `DataFrame` variable name, followed by a period, the `loc` keyword, and a pair of square brackets. Inside the brackets you need to pass

the row and column labels you want to access. To make the same range selection as above using `loc`, you need to run:

```
In [16]: ledger_df.loc[0:9, 'InvoiceNo':'ProductID']
```

```
Out [16]:
```

	InvoiceNo	Channel	Product Name	ProductID
0	1532	Shoppe.com	Cannon Water Bom...	T&G/CAN-97509
1	1533	Walcart	LEGO Ninja Turtl...	T&G/LEG-37777
2	1534	Bullseye		NaN
3	1535	Bullseye	Transformers Age...	T&G/TRA-20170
4	1535	Bullseye	Transformers Age...	T&G/TRA-20170
5	1537	Bullseye	3x Anti-Spy Priv...	CP&A/3X-00445
6	1538	Understock.com		NaN
7	1539	Walcart	Zen-Ray ED3 8x43...	C&P/ZEN-80993
8	1540	iBay.com	ePartSolution-Sa...	CP&A/EPA-01656
9	1541	Bullseye	Timex Marathon G...	S&O/TIM-92731

The output is a `DataFrame` that you can assign to a separate variable if you want to.

Unlike Excel's formula, with `loc` you first have to specify the row labels you want to select, followed by the column labels: so A1:D10 in Excel becomes `.loc[0:9, 'InvoiceNo': 'ProductID']` in pandas. Because the row labels in `ledger_df` start at 0, the row range in the example above goes from 0 to 9 instead of 1 to 10 as it does in Excel.

Besides label ranges, you can use Python lists to enumerate the rows or columns you want to select:

```
In [17]: ledger_df.loc[[0, 1, 2, 3, 4], ['InvoiceNo', 'Channel', 'Unit Price', 'Total']]
```

```
Out [17]:
```

	InvoiceNo	Channel	Unit Price	Total
0	1532	Shoppe.com	20.11	281.54
1	1533	Walcart	6.70	6.70
2	1534	Bullseye	11.67	58.35
3	1535	Bullseye	13.46	80.76
4	1535	Bullseye	13.46	80.76

If you want to select *all* rows in a `DataFrame`, you can use a single colon instead of a range of row labels:

```
In [18]: ledger_df.loc[:, 'Channel':'Date']
```

```
Out [18]:
```

	Channel	Product Name	ProductID	Account	AccountNo	Date	
0	Shoppe.com	Cannon Water Bom...	T&G/CAN-97509	Sales	5004	2020-01-01	
1	Walcart	LEGO Ninja Turtl...	T&G/LEG-37777	Sales	5004	2020-01-01	
2	Bullseye		NaN	T&G/PET-14209	Sales	5004	2020-01-01
3	Bullseye	Transformers Age...	T&G/TRA-20170	Sales	5004	2020-01-01	
4	Bullseye	Transformers Age...	T&G/TRA-20170	Sales	5004	2020-01-01	
...	
14049	Bullseye	AC Adapter/Power...	E/AC-63975	Sales	5004	2020-01-31	
14050	Bullseye	Cisco Systems Gi...	E/CIS-74992	Sales	5004	2020-01-31	
14051	Understock.com	Philips AJ3116M/...	E/PHI-08100	Sales	5004	2020-01-31	
14052	iBay.com		NaN	E/POL-61164	Sales	5004	2020-01-31
14053	Understock.com	Sirius Satellite...	E/SIR-83381	Sales	5004	2020-01-31	

```
[14054 rows x 6 columns]
```

The same works for selecting columns; instead of a range or a list of column labels, you can use a single colon to select all columns in a DataFrame:

```
In [19]: ledger_df.loc[0:4, :]
```

```
Out [19]:   InvoiceNo    Channel      Product Name ... Unit Price Quantity  Total
0         1532  Shoppe.com  Cannon Water Bomb ...    ...     20.11      14  281.54
1         1533    Walcart  LEGO Ninja Turtles...    ...      6.70       1    6.70
2         1534    Bullseye           NaN    ...     11.67       5   58.35
3         1535    Bullseye  Transformers Age o...    ...     13.46       6   80.76
4         1535    Bullseye  Transformers Age o...    ...     13.46       6   80.76
```

```
[5 rows x 12 columns]
```

Similarly, if you want to select a single row or column (or both), you can use row and column labels by themselves (i.e., not in a list or a range):

```
In [20]: ledger_df.loc[0, 'Channel']
```

```
Out [20]: 'Shoppe.com'
```

Using iloc

The `iloc` indexing operator does the same thing as `loc`, but instead of using row and column labels, it uses row and column *positions* in the DataFrame specified as integers (the *i* stands for integer). With `iloc`, you can make the same selection as in the previous Excel example (i.e., A1:D10) using:

```
In [21]: ledger_df.iloc[0:10, 0:4]
```

```
Out [21]:   InvoiceNo    Channel      Product Name  ProductID
0         1532  Shoppe.com  Cannon Water Bomb ...  T&G/CAN-97509
1         1533    Walcart  LEGO Ninja Turtles...  T&G/LEG-37777
2         1534    Bullseye           NaN  T&G/PET-14209
3         1535    Bullseye  Transformers Age o...  T&G/TRA-20170
4         1535    Bullseye  Transformers Age o...  T&G/TRA-20170
5         1537    Bullseye  3x Anti-Spy Privac...  CP&A/3X-00445
6         1538  Understock.com           NaN  S&O/TEX-91494
7         1539    Walcart  Zen-Ray ED3 8x43 B...  C&P/ZEN-80993
8         1540    iBay.com  ePartSolution-Sams...  CP&A/EPA-01656
9         1541    Bullseye  Timex Marathon GPS...  S&O/TIM-92731
```

Row and column positions start with 0, so you need to use 0 as the start of your ranges if you want to include the first row or column in your selections when using `iloc`. The eagle-eyed among you may have noticed that the row range above goes up to 10, not 9 as in the previous example using `loc`, yet you still get ten rows in the

output. This is because `iloc` ranges are inclusive only on the left side, not on both, as are `loc` ranges.³

To select multiple rows and columns from `ledger_df`, you can also enumerate the positions you want to select using Python lists:

```
In [22]: ledger_df.iloc[[0, 1, 2, 3, 4], [1, 2, 9, 11]]
```

```
Out [22]:
```

	Channel	Product Name	Unit Price	Total
0	Shoppe.com	Cannon Water Bom...	20.11	281.54
1	Walcart	LEGO Ninja Turtl...	6.70	6.70
2	Bullseye		NaN	11.67
3	Bullseye	Transformers Age...	13.46	80.76
4	Bullseye	Transformers Age...	13.46	80.76

You don't have to write a number before or after the colon if you want to specify a range that starts with the first or ends with the last possible item (row or column):

```
In [23]: ledger_df.iloc[:, :3]
```

```
Out [23]:
```

	InvoiceNo	Channel	Product Name
0	1532	Shoppe.com	Cannon Water Bom...
1	1533	Walcart	LEGO Ninja Turtl...
2	1534	Bullseye	NaN
3	1535	Bullseye	Transformers Age...
4	1535	Bullseye	Transformers Age...
...
14049	15581	Bullseye	AC Adapter/Power...
14050	15582	Bullseye	Cisco Systems Gi...
14051	15583	Understock.com	Philips AJ3116M/...
14052	15584	iBay.com	NaN
14053	15585	Understock.com	Sirius Satellite...

[14054 rows x 3 columns]

You can even use negative integers to select rows or columns starting from the end of a `DataFrame` — this selects the last four rows and last four columns in `ledger_df`:

```
In [24]: ledger_df.iloc[-4:, -4:]
```

```
Out [24]:
```

	Deadline	Unit Price	Quantity	Total
24310	2020-06-13	33.39	1	33.39
24311	2020-05-09	4.18	1	4.18
24312	2020-07-25	4.78	25	119.50
24313	2020-06-18	33.16	2	66.32

The two indexing operators can be used in intricate ways and can even be chained one after the other for more complicated table slices. Depending on your selection, their output is a `DataFrame`, a `Series` or a single value — all of which you can assign to other variables and use elsewhere in your code. We'll revisit `loc` and `iloc` often throughout the rest of the book.

3: There's a good reason for this difference between `iloc` and `loc`. Label ranges used with `loc` are inclusive at both ends because you don't always know the order of labels in your tables. Excluding the right end of your label range when using `loc` would require you to know what label comes right after it, if you wanted to include the right end of your range in your table slice. This can be inconvenient, if you don't know the order of labels in your table. On the other hand, excluding the right end of an integer range — as `iloc` does — is consistent with slicing Python lists.

The `loc` and `iloc` trap

Slicing using `iloc` and `loc` can get tricky when `DataFrame` rows have integer labels, as is the case with `ledger_df`. Consider the following example:

```
In [25]: ledger_df.loc[0:4, 'Channel']
```

```
Out [25]: 0    Understock.com
           1        Walcart
           2        Walcart
           3    Understock.com
           4      Bullseye
Name: Channel, dtype: object
```

Here, you use a label range with `loc` to select the first five rows in `ledger_df`. The reason this works is that row labels in `ledger_df` are, indeed, integers. However, if you use the same range with `iloc`, you get a different result:

```
In [26]: ledger_df.iloc[0:4, 1]
```

```
Out [26]: 0    Shoppe.com
           1        Walcart
           2      Bullseye
           3      Bullseye
Name: Channel, dtype: object
```

The first example uses `loc` and selects the first five rows of the '`Channel`' column (i.e., the second column in `ledger_df`). In contrast, the second example uses `iloc` and selects the first *four* rows of the same column, even though the specified row range seems to be the same.

The reason for this difference is that `loc` selects all rows between and including the row labeled 0 and row labeled 4, whereas `iloc` selects rows based on their position in `ledger_df` and excludes the right limit of the row range (i.e., the row position specified after the colon). This might seem like a minor detail, but most of the `DataFrame` variables you will work with will have integer row labels; being mindful of this difference between `loc` and `iloc` might save some headache later on.

Filtering data

In our quick tour of `pandas`, you saw a concise example of filtering the sales data we've been working with based on values in its '`Channel`' column, similar to the code below:

```
In [27]: ledger_df[ledger_df['Channel'] == 'Walcart']
```

Out [27]:

	InvoiceNo	Channel	Product Name	...	Unit Price	Quantity	Total
1	1533	Walcart	LEGO Ninja Turtl...	...	6.70	1	6.70
7	1539	Walcart	Zen-Ray ED3 8x43...	...	28.29	1	28.29
10	1542	Walcart	Logisys Red 5 LE...	...	19.49	1	19.49
11	1543	Walcart	Magline GMK81UA4...	...	18.42	4	73.68
15	1547	Walcart	Totally Bamboo 2...	...	4.25	1	4.25
...
14012	15408	Walcart	OXO Good Grips L...	...	10.10	1	10.10
14022	15427	Walcart	Applied Nutritio...	...	7.72	1	7.72
14028	15459	Walcart	Update Internati...	...	3.91	1	3.91
14034	15471	Walcart	Kikkerland Biode...	...	17.18	16	274.88
14039	15499	Walcart	Anchor Hocking 4...	...	7.56	44	332.64

[1851 rows x 12 columns]

This code filters `ledger_df` and returns a `DataFrame` that contains all rows from `ledger_df` where values in the '`Channel`' column are equal to '`Walcart`' (remember that double equals checks for equality, single equals assigns a value to a variable).

If you run just the equality check between the square brackets above, by itself, in a separate cell, you will see that it returns a `Series` with the same number of rows as `ledger_df`, containing only boolean values (i.e., `True` or `False` values):

In [28]: `ledger_df['Channel'] == 'Walcart'`

Out [28]:

0	False
1	True
2	False
3	False
4	False
...	
14049	False
14050	False
14051	False
14052	False
14053	False

Name: Channel, Length: 14054, dtype: bool

The values in this boolean `Series` show which rows in `ledger_df` meet the condition and which don't. It is just a regular `Series`, so you can assign it to a separate variable or sum its values⁴ to find out how many rows in `ledger_df` meet the condition:

In [29]: `rows_to_keep = ledger_df['Channel'] == 'Walcart'`

`rows_to_keep.sum()`

Out [29]: 1851

However, as you saw above, this boolean `Series` is most useful when you want to filter rows:

In [30]: `ledger_df[rows_to_keep]`

4: Because boolean values are just numbers in disguise: `True` is equal to 1, `False` is equal to 0.

Out [30]:

	InvoiceNo	Channel	Product Name	...	Unit Price	Quantity	Total
1	1533	Walcart	LEGO Ninja Turtl...	...	6.70	1	6.70
7	1539	Walcart	Zen-Ray ED3 8x43...	...	28.29	1	28.29
10	1542	Walcart	Logisys Red 5 LE...	...	19.49	1	19.49
11	1543	Walcart	Magline GMK81UA4...	...	18.42	4	73.68
15	1547	Walcart	Totally Bamboo 2...	...	4.25	1	4.25
...
14012	15408	Walcart	OXO Good Grips L...	...	10.10	1	10.10
14022	15427	Walcart	Applied Nutritio...	...	7.72	1	7.72
14028	15459	Walcart	Update Internati...	...	3.91	1	3.91
14034	15471	Walcart	Kikkerland Biode...	...	17.18	16	274.88
14039	15499	Walcart	Anchor Hocking 4...	...	7.56	44	332.64

[1851 rows x 12 columns]

The easiest way to create boolean `Series` is with conditional statements, as in the example above.⁵ You can write conditional statements that work with `DataFrame` variables just like Python conditional statements. The example above checks for equality using the `==` operator, but you can use any other comparison operator instead (i.e., `<`, `>`, `!=` which is not equals, `<=` or `>=`).

You will often want to combine multiple conditional statements to filter rows in more complicated ways. When specifying multiple statements, each condition must be surrounded by parentheses:

In [31]:

```
1 ledger_df[
2     (ledger_df['Channel'] == 'Walcart') &
3     (ledger_df['Quantity'] > 25)
4 ]
```

5: But any sequence of boolean values can be used to filter a `DataFrame` (even if you type it by hand, as a long Python list), if it has the same number of `True` or `False` values as there are rows in the `DataFrame` you want to filter.

Out [31]:

	InvoiceNo	Channel	Product Name	...	Unit Price	Quantity	Total
28	1560	Walcart	Conntek 14422 RV...	...	4.47	27	120.69
144	1676	Walcart	[Strip of 6] E...	...	11.83	61	721.63
213	1604	Walcart	Conntek 14422 RV...	...	4.47	27	120.69
237	1769	Walcart	Child Constructi...	...	19.68	43	846.24
292	1824	Walcart	Ultima Replenish...	...	13.34	62	827.08
...
13751	15144	Walcart	Anchor Hocking 4...	...	7.56	44	332.64
13759	15291	Walcart	AC Adapter/batte...	...	14.49	68	985.32
13827	15359	Walcart	AKG Pro Audio K9...	...	8.10	31	251.10
13897	15429	Walcart		NaN	13.34	83	1107.22
14039	15499	Walcart	Anchor Hocking 4...	...	7.56	44	332.64

[86 rows x 12 columns]

One difference from regular Python when using `pandas` is that you cannot use `and` or `or` to combine `DataFrame` conditional statements. With `pandas`, you need to use the `&` operator for `and`, and the `|` operator for `or`. The same goes for the negation operator: you cannot use `not` with a boolean `Series` to negate it, you need to use the `~` (tilde) operator:

```
In [32]: ledger_df[~(ledger_df['Channel'] == 'Walcart')] 1
# same as 2
# ledger_df[ledger_df['Channel'] != 'Walcart'] 3
4
```

	InvoiceNo	Channel	Product Name	Unit Price	Quantity	Total
0	1532	Shoppe.com	Cannon Water Bom...	20.11	14	281.54
2	1534	Bullseye		11.67	5	58.35
3	1535	Bullseye	Transformers Age...	13.46	6	80.76
4	1535	Bullseye	Transformers Age...	13.46	6	80.76
5	1537	Bullseye	3x Anti-Spy Priv...	7.39	8	59.12
...
14049	15581	Bullseye	AC Adapter/Power...	28.72	8	229.76
14050	15582	Bullseye	Cisco Systems Gi...	33.39	1	33.39
14051	15583	Understock.com	Philips AJ3116M/...	4.18	1	4.18
14052	15584	iBay.com		4.78	25	119.50
14053	15585	Understock.com	Sirius Satellite...	33.16	2	66.32

[12203 rows x 12 columns]

Combining conditional statements using logical operators still returns a Series of `True` or `False` values. If you run any of the statements above in a separate cell, you will get the same kind of boolean Series you saw earlier:

```
In [33]: (ledger_df['Channel'] == 'Walcart') & (ledger_df['Quantity'] > 25)
```

0	False
1	False
2	False
3	False
4	False
...	
14049	False
14050	False
14051	False
14052	False
14053	False
Length:	14054, dtype: bool

```
In [34]: ~(ledger_df['Channel'] == 'Walcart')
```

0	True
1	False
2	True
3	True
4	True
...	
14049	True
14050	True
14051	True
14052	True
14053	True
Name:	Channel, Length: 14054, dtype: bool

There are a few `Series` methods that return a boolean `Series` you can use to simplify writing long `DataFrame` filters. One such method is `isin`: instead of writing multiple conditional statements and joining them with the pandas “*or*” `|` operator, you can check whether each value in a column matches an element in a sequence of values with:

```
In [35]: # Instead of:
(
    (ledger_df['Channel'] == 'Understock.com') |
    (ledger_df['Channel'] == 'iBay.com') |
    (ledger_df['Channel'] == 'Shoppe.com')
)

# You can use:
ledger_df['Channel'].isin(['Understock.com', 'iBay.com', 'Shoppe.com'])
```

```
Out [35]: 0      True
1      False
2      False
3      False
4      False
...
14049  False
14050  False
14051  True
14052  True
14053  True
Name: Channel, Length: 14054, dtype: bool
```

Notice that the argument passed to `isin` is a Python list containing the values we want to check for (and not just the values, one after the other). Similarly, you can use `between` to check whether values in a column are in a given range (inclusive at both ends):

```
In [36]: # Instead of:
(ledger_df['Quantity'] >= 10) & (ledger_df['Quantity'] <= 100)

# You can use:
ledger_df['Quantity'].between(10, 100)
```

```
Out [36]: 0      True
1      False
2      False
3      False
4      False
...
14049  False
14050  False
14051  False
14052  True
14053  False
Name: Quantity, Length: 14054, dtype: bool
```

You can use the output of these methods to filter `DataFrame` rows, just like boolean `Series` that result from conditional statements.

Using `loc` for filtering

Occasionally you will need to filter rows and select columns at the same time. There are several ways you can do that — perhaps the most straightforward way is to chain multiple square bracket operations on a `DataFrame`:

```
In [37]: ledger_df[ledger_df['Channel'] == 'Walcart'][['Channel', 'Quantity', 'Total']]
```

```
Out [37]:   Channel  Quantity  Total
1      Walcart      1    6.70
7      Walcart      1   28.29
10     Walcart      1   19.49
11     Walcart      4   73.68
15     Walcart      1    4.25
...
14012  Walcart      1   10.10
14022  Walcart      1    7.72
14028  Walcart      1    3.91
14034  Walcart     16  274.88
14039  Walcart     44  332.64
```

[1851 rows x 3 columns]

Filtering rows returns a `DataFrame`, so you can use the output of a filtering operation as any other `DataFrame`, without assigning it to a variable — here, I used it to select several columns. You can keep chaining square bracket operations this way as long as their output is a `DataFrame`, but code like this becomes hard to understand fast.

Another way to filter rows and select columns at the same time is with the `loc` indexing operator:

```
In [38]: ledger_df.loc[
    ledger_df['Channel'] == 'Walcart',
    ['Channel', 'Quantity', 'Total']
]
```

```
Out [38]:   Channel  Quantity  Total
1      Walcart      1    6.70
7      Walcart      1   28.29
10     Walcart      1   19.49
11     Walcart      4   73.68
15     Walcart      1    4.25
...
14012  Walcart      1   10.10
14022  Walcart      1    7.72
14028  Walcart      1    3.91
14034  Walcart     16  274.88
```

```
14039 Walcart      44 332.64
```

```
[1851 rows x 3 columns]
```

Notice that instead of a list of row labels, the first argument passed to `loc` is a boolean `Series` (created using a conditional statement), whereas the second argument is a list of column labels. So `loc` actually works with row labels *and* boolean `Series` for selecting rows (but `iloc` doesn't).

This might not seem like a significant improvement over the previous example. Still, using conditional statements with `loc` will become handy later because it enables filter and edit operations: you can use `loc` to modify multiple values in your table based on a condition — more on this in the following chapters.

Sorting data

To sort the rows in a `DataFrame`, you can use the `sort_values` method and specify the column you want to sort by:

```
In [39]: ledger_df.sort_values(by='Total')
```

```
Out [39]:
```

	InvoiceNo	Channel	Product Name	...	Unit Price	Quantity	Total
7613	9145	iBay.com	Magic: the Gathe...	...	0.06	15	0.90
12877	14409	Understock.com	Magic: the Gathe...	...	0.06	17	1.02
7810	9342	iBay.com	Magic: the Gathe...	...	0.06	26	1.56
7812	9270	iBay.com	Magic: the Gathe...	...	0.06	26	1.56
9822	11354	Understock.com	Urban Rebounding...	...	1.69	1	1.69
...
4757	6162	Understock.com	AC Adapter/batte...	...	14.88	226	3362.88
6163	7526	iBay.com	Large Display Di...	...	64.15	56	3592.40
6141	7673	iBay.com	Large Display Di...	...	64.15	56	3592.40
8006	9538	iBay.com	Large Display Di...	...	64.15	61	3913.15
5212	6744	iBay.com	Large Display Di...	...	64.15	68	4362.20

```
[14054 rows x 12 columns]
```

By default, sorting is ascending. You can pass the `ascending=False` keyword argument to `sort_values` if you want to sort in descending order:

```
In [40]: ledger_df.sort_values(by='Total', ascending=False)
```

```
Out [40]:
```

	InvoiceNo	Channel	Product Name	...	Unit Price	Quantity	Total
5212	6744	iBay.com	Large Display Di...	...	64.15	68	4362.20
8006	9538	iBay.com	Large Display Di...	...	64.15	61	3913.15
6141	7673	iBay.com	Large Display Di...	...	64.15	56	3592.40
6163	7526	iBay.com	Large Display Di...	...	64.15	56	3592.40
8797	10329	Understock.com	AC Adapter/batte...	...	14.88	226	3362.88
...
9822	11354	Understock.com	Urban Rebounding...	...	1.69	1	1.69
7810	9342	iBay.com	Magic: the Gathe...	...	0.06	26	1.56

7812	9270	iBay.com	Magic: the Gathe...	...	0.06	26	1.56
12877	14409	Understock.com	Magic: the Gathe...	...	0.06	17	1.02
7613	9145	iBay.com	Magic: the Gathe...	...	0.06	15	0.90

[14054 rows x 12 columns]

Notice in the output above that, after sorting, row labels are not consecutive anymore. This is because row labels (even if they are numbers) don't indicate a row's position, but are just row identifiers in a `DataFrame`. After sorting, these labels get shuffled around with their associated rows, as above.

If you want to sort by multiple columns, you can pass a list of column names to `sort_values`. When you use multiple columns with `sort_values`, pandas uses values from the first column in your list to sort rows. In case of equality among values in the first column, values from the second columns are used — and so on for all the columns you pass to `sort_values`. You can even pass a list of `True` or `False` values to the `ascending` keyword argument to control the sorting direction for each separate column:

In [41]: `ledger_df.sort_values(by=['Quantity', 'Total'], ascending=[False, True])`

	InvoiceNo	Channel	Product Name	...	Unit Price	Quantity	Total	
4747	6279	Understock.com	AC Adapter/batte...	...	14.88	226	3362.88	
4757	6162	Understock.com	AC Adapter/batte...	...	14.88	226	3362.88	
4969	6397	Understock.com	AC Adapter/batte...	...	14.88	226	3362.88	
8797	10329	Understock.com	AC Adapter/batte...	...	14.88	226	3362.88	
5499	7031	iBay.com		NaN	...	13.28	184	2443.52
...	
616	2148	Understock.com	Kodak EasyShare	...	166.30	1	166.30	
4248	5780	Understock.com	Kodak EasyShare	...	166.30	1	166.30	
4485	5839	Understock.com	Kodak EasyShare	...	166.30	1	166.30	
6152	7684	Understock.com	Kodak EasyShare	...	166.30	1	166.30	
6164	7531	Understock.com	Kodak EasyShare	...	166.30	1	166.30	

[14054 rows x 12 columns]

This sorts the entire `DataFrame`, first in descending order, using values from the `'Quantity'` column, and in the case of equal quantities, using values from the `'Total'` column, but in ascending order (i.e., the values passed to `ascending` controls this behavior).

Keep in mind that if you have any missing values (i.e., NaNs) in the column or columns you sort by, these values will be placed, by default, at the bottom of the sorted `DataFrame` (regardless of whether you pass `ascending=False` or not).

Summary

This chapter showed you how to slice, filter and sort `pandas DataFrame` objects. With what you learned in the previous chapters, you can now read, write, sort, and slice Excel spreadsheets using `pandas`.⁶ Next, let's look at how you can put the `pandas` features we've covered so far together, in a quick project that re-organizes “`Q1Sales.xlsx`” by channel and revenue.

6: Well done for making it this far!

Project: Organizing sales data by channel

16

If you've ever needed to split a large Excel file into multiple spreadsheets — to share them with different people or upload them as separate files on one of the platforms you use — you've likely mastered the art of copying and pasting rows. Copy-and-paste can often be the right tool for the job, but when you need to repeat the same file-splitting steps every other week, or when your data is too large and unwieldy for Excel, it can quickly become a headache.

This short project chapter shows you how to split an Excel file into multiple sheets with `pandas`. You'll use the “`Q1Sales.xlsx`” file you're already familiar with and split it into five spreadsheets, each one containing sales from one of the five sales channels in the data. You'll also sort the data in each spreadsheet by the `'Total'` and `'Quantity'` columns so that each sheet displays the highest-grossing sales at the top.

To get started, launch this chapter's notebook in JupyterLab — you'll find the notebook in your *Python for Accounting* workspace, in the “`P2 - Working with tables`” folder. Once you have the notebook open, the first line of code you need to run is importing `pandas`:

```
In [1]: import pandas as pd
```

As I mentioned in the previous chapters, this `import` statement gives you access to `pandas`'s functions in your current notebook. To load data from “`Q1Sales.xlsx`” into your notebook, you need the `read_excel` `pandas` function — let's use it to read the first sheet of “`Q1Sales.xlsx`” and assign its data to a `DataFrame` variable:

```
In [2]: jan_sales_df = pd.read_excel('Q1Sales.xlsx', sheet_name='January')
```

```
jan_sales_df
```

```
Out [2]:
```

	InvoiceNo	Channel	Product Name	Unit Price	Quantity	Total
0	1532	Shoppe.com	Cannon Water Bomb...	20.11	14	281.54
1	1533	Walcart	LEGO Ninja Turtle...	6.70	1	6.70
2	1534	Bullseye	Transformers Age ...	11.67	5	58.35
3	1535	Bullseye	Transformers Age ...	13.46	6	80.76
4	1535	Bullseye	Transformers Age ...	13.46	6	80.76
...
14049	15581	Bullseye	AC Adapter/Power ...	28.72	8	229.76
14050	15582	Bullseye	Cisco Systems Gig...	33.39	1	33.39
14051	15583	Understock.com	Philips AJ3116M/3...	4.18	1	4.18
14052	15584	iBay.com	NaN	4.78	25	119.50
14053	15585	Understock.com	Sirius Satellite ...	33.16	2	66.32

[14054 rows × 12 columns]

The code above reads data from the first sheet in “Q1Sales.xlsx”, but there are two more spreadsheets in that workbook. To read all the data in “Q1Sales.xlsx”, you can repeat the code above for the remaining sheets and assign the output of each `read_excel` call to separate `DataFrame` variables:

```
In [3]: jan_sales_df = pd.read_excel('Q1Sales.xlsx', sheet_name='January')  
feb_sales_df = pd.read_excel('Q1Sales.xlsx', sheet_name='February')  
mar_sales_df = pd.read_excel('Q1Sales.xlsx', sheet_name='March')
```

Now that you have all the data in “Q1Sales.xlsx” loaded in your notebook, let’s simplify the code a bit. Instead of having three table variables to work with, let’s put all their data in a single `DataFrame`. You can do that by using `pandas`’s `concat` function, which is the `pandas` equivalent of copy-pasting rows:

```
In [4]: sales_df = pd.concat([jan_sales_df, feb_sales_df, mar_sales_df], ignore_index=True)
```

In the code above, the first argument passed to `concat` is a Python list containing the three `DataFrame` variables you want to combine; the `ignore_index=True` keyword argument tells `pandas` to discard row labels when combining the input `DataFrame` objects. The output of `concat` is another `DataFrame` that you assign to the `sales_df` variable. We’ll come back to `concat` in the following chapters and uncover more of its features, but for now, check `sales_df` to convince yourself that it has the same data as the three `DataFrame` variables you created before:

```
In [5]: sales_df
```

	InvoiceNo	Channel	Product Name	...	Unit Price	Quantity	Total
0	1532	Shoppe.com	Cannon Water Bomb...	...	20.11	14	281.54
1	1533	Walcart	LEGO Ninja Turtle...	...	6.70	1	6.70
2	1534	Bullseye		NaN	11.67	5	58.35
3	1535	Bullseye	Transformers Age	13.46	6	80.76
4	1535	Bullseye	Transformers Age	13.46	6	80.76
...
37703	39235	iBay.com	Nature's Bounty G...	...	5.55	2	11.10
37704	39216	Shoppe.com	Funko Wonder Woma...	...	28.56	1	28.56
37705	39219	Shoppe.com	MONO GS1 GS1-BTY-	...	3.33	1	3.33
37706	39238	Shoppe.com		NaN	34.76	10	347.60
37707	39239	Understock.com	3 Collapsible Bow...	...	6.39	15	95.85

[37708 rows × 12 columns]

Having lots of variables you don’t need can make code unwieldy. In the code above, you don’t need one `DataFrame` variable for each sheet in “Q1Sales.xlsx”. You can change the data-reading code above (and skip all intermediate variables) by passing multiple `read_excel` calls to `concat` directly:

```
In [6]: sales_df = pd.concat([
    pd.read_excel('Q1Sales.xlsx', sheet_name='January'),
    pd.read_excel('Q1Sales.xlsx', sheet_name='February'),
    pd.read_excel('Q1Sales.xlsx', sheet_name='March')
],
ignore_index=True)
```

You get the same `sales_df` but without the extra variables.

The goal of this short project is to create an Excel file with five different sheets, each one containing sales from one of the five channels in `sales_df`. To get where we're going, let's first select all '`Understock.com`' sales from `sales_df` and sort the filtered `DataFrame` by its '`Total`' and '`Quantity`' columns:

```
In [7]: channel = 'Understock.com'
channel_df = sales_df[sales_df['Channel'] == channel]
channel_df = channel_df.sort_values(['Total', 'Quantity'], ascending=False)
```

```
In [8]: channel_df
```

```
Out [8]:   ProductID      Product Name      Channel  Unit Price  Quantity  Total
23620  H&PC/LAR-98606  Large Display Dig...  Understock.com    64.32      54  3473.28
36621  H&PC/LAR-98606  Large Display Dig...  Understock.com    64.32      54  3473.28
4747    E/AC-44106    AC Adapter/batter...  Understock.com    14.88     226  3362.88
4757    E/AC-44106    AC Adapter/batter...  Understock.com    14.88     226  3362.88
4969    E/AC-44106    AC Adapter/batter...  Understock.com    14.88     226  3362.88
...
...
...
15679   M&T/URB-83617  Urban Rebounding ...  Understock.com    1.69       1   1.69
12877   T&G/MAG-22549  Magic: the Gather...  Understock.com    0.06      17   1.02
15855   T&G/MAG-22549  Magic: the Gather...  Understock.com    0.06      11   0.66
24012   T&G/MAG-22549  Magic: the Gather...  Understock.com    0.06      11   0.66
24022   T&G/MAG-22549  Magic: the Gather...  Understock.com    0.06      11   0.66

[13188 rows x 6 columns]
```

The code above should look familiar: it filters `sales_df` and assigns the filtered `DataFrame` to a variable called `channel_df`. You can write `channel_df` to a new Excel file using:

```
In [9]: channel_df.to_excel('Q1ChannelSales.xlsx', sheet_name='Understock.com', index=False)
```

Running the code above will create an Excel file called "`Q1ChannelSales.xlsx`" in the same folder as your Jupyter notebook, with one sheet called "`Understock.com`" containing the same data as `channel_df`. However, for this project, you need to repeat the filter and sort operation above for all the sales channels in `sales_df`; whenever you want to repeat something in code, you need a loop.

Let's set up a `for` loop that goes through each channel in our sales data and prints its name:

```
In [10]: channels = ['Bullseye', 'iBay.com', 'Shoppe.com', 'Understock.com', 'Walcart']  
1  
2  
for channel in channels:  
3  
    print(channel)  
4
```

```
Out [10]: Bullseye  
iBay.com  
Shoppe.com  
Understock.com  
Walcart
```

The loop above doesn't do anything useful yet: it goes through each value in the `channels` list defined on line 1 and prints it. However, instead of printing channel names, you can use the loop to do some real work, like filtering `sales_df` and creating a different `channel_df` for every channel:

```
In [11]: channels = ['Bullseye', 'iBay.com', 'Shoppe.com', 'Understock.com', 'Walcart']  
1  
2  
for channel in channels:  
3  
    channel_df = sales_df[sales_df['Channel'] == channel]  
4  
    channel_df = channel_df.sort_values(['Total', 'Quantity'], ascending=False)
```

The code above doesn't output anything, but if you inspect `channel_df` now, you'll see it contains all `sales_df` rows where values in the `'Channel'` column are `'Walcart'`:

```
In [12]: channel_df  
1  
2  
Out [12]:   InvoiceNo  Channel      Product Name ... Unit Price Quantity  Total  
1 2461        3993  Walcart  Large Display Dig... ... 64.47     23 1482.81  
2 8332        9864  Walcart  AC Adapter/batter... ... 14.49    100 1449.00  
3 15293       16825  Walcart  AC Adapter/batter... ... 14.49    100 1449.00  
4 15509       16895  Walcart  AC Adapter/batter... ... 14.49    100 1449.00  
5 23888       25420  Walcart  AC Adapter/batter... ... 14.49    100 1449.00  
6 ...         ...     ...      ... ... ... ...  
7 936         2332  Walcart  Vibrating Slim Je... ... 2.10     1    2.10  
8 11222       12754  Walcart  Vibrating Slim Je... ... 2.10     1    2.10  
9 14186       15718  Walcart  Blackberry Q10 Wh... ... 1.87     1    1.87  
10 14311       15712  Walcart  Blackberry Q10 Wh... ... 1.87     1    1.87  
11 33627       35159  Walcart  Red Dragon VT 3-3... ... 0.19     6    1.14  
12  
[4574 rows x 12 columns]
```

Why `'Walcart'` sales? Because the `for` loop goes through all values in the `channels` list and keeps assigning a different DataFrame to `channel_df`. The last value in the `channels` list is `'Walcart'`, so at the last iteration of the loop, `channel_df` gets assigned only those rows in `sales_df` where values in the `'Channel'` column are equal to `'Walcart'`.

The last step you need to add to the `for` loop above is writing each `channel_df` to a different sheet in `"Q1ChannelSales.xlsx"`. You can do that by extending the previous example with the code below:

```
In [13]: channels = ['Bullseye', 'iBay.com', 'Shoppe.com', 'Understock.com', 'Walcart']

output_file = pd.ExcelWriter('Q1ChannelSales.xlsx')

for channel in channels:
    channel_df = sales_df[sales_df['Channel'] == channel]
    channel_df = channel_df.sort_values(['Total', 'Quantity'], ascending=False)
    channel_df.to_excel(output_file, sheet_name=channel, index=False)

output_file.save()
```

There are two new lines of code in the code above. On line 3, you use pandas's `ExcelWriter` object to open an Excel file and assign it to a variable called `output_file`. The `output_file` variable is an object that represents the opened file in Python code. When you need to write data to multiple sheets in the same Excel file, you have to first open the Excel file as I did above.

As before, the loop then goes through each channel, filters and sorts rows from `sales_df`, and assigns them to `channel_df`. The loop also calls `to_excel`, passing the channel name as the `sheet_name` keyword argument — this is how each channel's sales get written to a different sheet in the open “`Q1ChannelSales.xlsx`” Excel file. When the loop completes, you call `output_file.save()` (on line 10), which makes sure the data in the open file gets written to your computer's disk (you have to call `save` this way if you want to make sure your Excel file and its data get saved).

You might come across another way of writing the same code that uses Python's `with` operator.¹ In the code below, the `with` operator opens and saves files for you, so you don't have to remember to call `save` on your open file variable once you're done writing data to it. Using `with` you get the same result as above by running:

```
In [14]: with pd.ExcelWriter('Q1ChannelSales.xlsx') as output_file:
    for channel in channels:
        channel_df = sales_df[sales_df['Channel'] == channel]
        channel_df = channel_df.sort_values(['Total', 'Quantity'], ascending=False)
        channel_df.to_excel(output_file, sheet_name=channel, index=False)
```

Notice in both examples that instead of a file name, you pass the `output_file` variable as the first argument to `to_excel`. This tells pandas to use the already open file when writing data and is the only way to write data to multiple spreadsheets in the same Excel file. If you open “`Q1ChannelSales.xlsx`” in Excel after running the code above, you will see something similar to figure 16.1 below.

The entire code you need to read data from “`Q1Sales.xlsx`”, combine it in a single `DataFrame`, and write it to another Excel file, with one sheet for each sales channel is shown below:

1: Python's `with` operator is nothing like VBA's With statement. In Python, `with` creates a context that handles errors for you.

```
In [15]: sales_df = pd.concat([
    pd.read_excel('Q1Sales.xlsx', sheet_name='January'),
    pd.read_excel('Q1Sales.xlsx', sheet_name='February'),
    pd.read_excel('Q1Sales.xlsx', sheet_name='March')
], ignore_index=True)
channels = ['Bullseye', 'iBay.com', 'Shoppe.com', 'Understock.com', 'Walcart']

with pd.ExcelWriter('Q1ChannelSales.xlsx') as output_file:
    for channel in channels:
        channel_df = sales_df[sales_df['Channel'] == channel]
        channel_df = channel_df.sort_values(['Total', 'Quantity'], ascending=False)
        channel_df.to_excel(output_file, sheet_name=channel, index=False)
```

You can extend the block of code in the `for` loop to add other slicing or sorting steps, depending on what you need from your data. If you need to split large datasets into multiple sheets often, code like the one above can save you considerable time.

Whenever you need to repeat data-handling steps, you should consider using a `for` loop. Figuring out the loop can take some trial-and-error: start by running a few steps manually to see what pandas functions you need, and then incrementally extend your code by adding other features. Don't be put-off by errors: Jupyter notebooks are designed for interactive coding, which means trying different things in your code, getting errors, and figuring out how to fix them until you make it all work.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	InvoiceNo	Channel	Product Name	ProductID	Account	AccountNo	Date	Deadline	Currency	Unit Price	Quantity	Total	
2	17329	Bullseye	AC Adapter/batte	E/AC-44106	Sales	5004	2020-02-04 00:00:00	2020/06/22	USD	14.3	77	1101.1	
3	20249	Bullseye	AC Adapter/batte	E/AC-44106	Sales	5004	2020-02-11 00:00:00	2020/06/01	USD	14.3	77	1101.1	
4	20179	Bullseye	AC Adapter/batte	E/AC-44106	Sales	5004	2020-02-11 00:00:00	2020/06/01	USD	14.3	77	1101.1	
5	20463	Bullseye	AC Adapter/batte	E/AC-44106	Sales	5004	2020-02-11 00:00:00	2020/06/01	USD	14.3	77	1101.1	
6	12383	Bullseye	AC Adapter/batte	E/AC-44106	Sales	5004	2020-01-24 00:00:00	2020/01/14	USD	14.3	64	915.2	
7	18320	Bullseye	AC Adapter/batte	E/AC-44106	Sales	5004	2020-02-07 00:00:00	2019/12/17	USD	14.3	64	915.2	
8	18261	Bullseye	AC Adapter/batte	E/AC-44106	Sales	5004	2020-02-07 00:00:00	2019/12/17	USD	14.3	64	915.2	
9	22341	Bullseye	Zippo Butane Fue	H&PC/ZIP-27394	Sales	5004	2020-02-15 00:00:00	Fri Mar 13 00:00:00	USD	17.84	51	909.84	
10	5184	Bullseye	Cat People / The C	M&T/CAT-34864	Sales	5004	2020-01-08 00:00:00	April 09 2020	USD	14.61	58	847.38	
11	5210	Bullseye	Cat People / The C	M&T/CAT-34864	Sales	5004	2020-01-08 00:00:00	April 09 2020	USD	14.61	58	847.38	
12	3505	Bullseye		H&PC/LAR-98606	Sales	5004	2020-01-04 00:00:00	Mon Apr 13 00:00:00	USD	64.39	13	837.07	
13	7709	Bullseye	Ultima Replenish	H&PC/ULT-64807	Sales	5004	2020-01-13 00:00:00	Tue Feb 18 00:00:00	USD	13.33	60	799.8	
14	1736	Bullseye	Spectrum AIL DT	MI/SPE-65286	Sales	5004	2020-01-01 00:00:00	10/27/19	USD	44.87	17	762.79	
15	1767	Bullseye	Spectrum AIL DT	MI/SPE-65286	Sales	5004	2020-01-01 00:00:00	10/27/19	USD	44.87	17	762.79	
16	35204	Bullseye	Spectrum AIL DT	MI/SPE-65286	Sales	5004	2020-03-15 00:00:00	01/10/20	USD	44.87	17	762.79	
17	35175	Bullseye	Spectrum AIL DT	MI/SPE-65286	Sales	5004	2020-03-15 00:00:00	01/10/20	USD	44.87	17	762.79	
18	6747	Bullseye	Fender Premium	MI/FEN-54916	Sales	5004	2020-01-11 00:00:00	04/18/20	USD	13.12	56	734.72	
19	24359	Bullseye	Foscam FI8910W	C&P/FOS-95687	Sales	5004	2020-02-19 00:00:00	11-29-19	USD	23.9	29	693.1	
20	32133	Bullseye	Foscam FI8910W	C&P/FOS-95687	Sales	5004	2020-03-07 00:00:00	2-11-20	USD	23.9	29	693.1	

Figure 16.1: Screenshot of “Q1ChannelSales.xlsx” opened in Excel.

Summary

This quick project chapter showed you how to split the “*Q1Sales.xlsx*” Excel file into multiple spreadsheets. Once you start getting comfortable with `pandas` and its functions, you’ll use it to replace all your manual data-handling, including copying-and-pasting rows from one sheet to another.

Next, let’s head back to the scenic tour of `pandas` and see how you can add and modify `DataFrame` columns.

Adding and modifying columns

When you open a spreadsheet in Excel, you get endless empty columns just waiting to be filled up with data. In pandas you don't get any empty columns, but you can still easily add new data to your `DataFrame` variables — this chapter shows you how.

As before, you need to `import pandas` and read "`Q1Sales.xlsx`" into a `DataFrame` to run the code examples ahead.

Adding columns

Adding columns to a `DataFrame` uses the already familiar `=` operator. Assuming you have already imported `pandas` and have the sales data loaded into a `DataFrame` called `ledger_df`, you can add a new column named '`Quarter`' to the `DataFrame` using:

```
In [1]: ledger_df['Quarter'] = 'Q1'

ledger_df
```

	InvoiceNo	Channel	Product Name	Quantity	Total	Quarter
0	1532	Shoppe.com	Cannon Water Bom...	14	281.54	Q1
1	1533	Walcart	LEGO Ninja Turtl...	1	6.70	Q1
2	1534	Bullseye	Nan	5	58.35	Q1
3	1535	Bullseye	Transformers Age...	6	80.76	Q1
4	1535	Bullseye	Transformers Age...	6	80.76	Q1
...
14049	15581	Bullseye	AC Adapter/Power...	8	229.76	Q1
14050	15582	Bullseye	Cisco Systems Gi...	1	33.39	Q1
14051	15583	Understock.com	Philips AJ3116M/...	1	4.18	Q1
14052	15584	iBay.com	Nan	25	119.50	Q1
14053	15585	Understock.com	Sirius Satellite...	2	66.32	Q1

[14054 rows x 13 columns]

The example above adds a new column at the end of `ledger_df` and fills it with the '`Q1`' string value.

Using the same code, you can assign new values to an existing column, which overwrites all its current values:

```
In [2]: ledger_df['Quarter'] = 1

ledger_df
```

```
Out [2]:
```

	InvoiceNo	Channel	Product Name	...	Quantity	Total	Quarter
0	1532	Shoppe.com	Cannon Water Bom...	...	14	281.54	1
1	1533	Walcart	LEGO Ninja Turtl...	...	1	6.70	1
2	1534	Bullseye		NaN	5	58.35	1
3	1535	Bullseye	Transformers Age...	...	6	80.76	1
4	1535	Bullseye	Transformers Age...	...	6	80.76	1
...
14049	15581	Bullseye	AC Adapter/Power...	...	8	229.76	1
14050	15582	Bullseye	Cisco Systems Gi...	...	1	33.39	1
14051	15583	Understock.com	Philips AJ3116M/...	...	1	4.18	1
14052	15584	iBay.com		NaN	25	119.50	1
14053	15585	Understock.com	Sirius Satellite...	...	2	66.32	1

[14054 rows x 13 columns]

Most often, you won't want to fill a column with a single value but rather use some of your table's existing values to create a new column. For example, to calculate the tax amount (assuming a 19% sales tax) for each row in `ledger_df` you can use:

```
In [3]: ledger_df['Total'] * (19 / 100)
```

```
Out [3]:
```

0	53.4926
1	1.2730
2	11.0865
3	15.3444
4	15.3444
...	...
14049	43.6544
14050	6.3441
14051	0.7942
14052	22.7050
14053	12.6008

Name: Total, Length: 14054, dtype: float64

There's no need for a `for` loop to go through each value in the `'Total'` column: pandas makes column operations as simple as multiplying two numbers. In the code above, each value in `ledger_df`'s `'Total'` column is multiplied by 19%, and the output is a new `Series` object. You can add this `Series` object as a new column to `ledger_df` using:

```
In [4]: ledger_df['Sales Tax'] = ledger_df['Total'] * (19 / 100)
```

This code doesn't return anything, so there's no output beneath the cell, but if you take a look at `ledger_df`, you'll see the new column at the end of the table:

```
In [5]: ledger_df
```

	InvoiceNo	Channel	Product Name	...	Total	Quarter	Sales Tax
0	1532	Shoppe.com	Cannon Water Bom...	...	281.54	1	53.4926
1	1533	Walcart	LEGO Ninja Turtl...	...	6.70	1	1.2730
2	1534	Bullseye		NaN	58.35	1	11.0865
3	1535	Bullseye	Transformers Age...	...	80.76	1	15.3444

4	1535	Bullseye	Transformers	Age...	...	80.76	1	15.3444
...
14049	15581	Bullseye	AC Adapter/Power...	...	229.76	1	43.6544	
14050	15582	Bullseye	Cisco Systems Gi...	...	33.39	1	6.3441	
14051	15583	Understock.com	Philips AJ3116M/...	...	4.18	1	0.7942	
14052	15584	iBay.com		NaN	119.50	1	22.7050	
14053	15585	Understock.com	Sirius Satellite...	...	66.32	1	12.6008	

[14054 rows x 14 columns]

You can use several columns in the same expression (e.g., multiply two columns together) and assign the output to another column:

```
In [6]: ledger_df['Quantity'] * ledger_df['Unit Price']
```

```
Out [6]: 0      281.54
1      6.70
2     58.35
3    80.76
4    80.76
...
14049   229.76
14050   33.39
14051   4.18
14052   119.50
14053   66.32
Length: 14054, dtype: float64
```

```
In [7]: ledger_df['Total'] = ledger_df['Quantity'] * ledger_df['Unit Price']
```

In general, whatever you can do with single numbers in Python, you can also do with DataFrame columns. We'll look at more Series operations in the following chapters — many are similar to the example above, so you already know the gist of it.

New column names can be anything you want, but short names will save you from having to type a lot, and it's a good idea to keep them consistent with your table's existing columns (e.g., all lowercase letters, title case, etc.).

In pandas, you work with entire columns at once, not just individual values (as you do in regular Python). Thinking "*in columns*" takes some getting used to, so don't worry if it all feels like a mental workout right now. As you read and write more code, it will soon become second nature.

Renaming columns

Whenever you regret new column name choices or your table headers are confusing, you can take comfort in knowing that column names are not permanent. You can rename columns in

your DataFrame variables using the `rename` method by passing a dictionary mapping old names to new ones:

```
In [8]: ledger_df.rename(columns={"Sales Tax": "Tax"})
```

	InvoiceNo	Channel	Product Name	...	Total	Quarter	Tax
0	1532	Shoppe.com	Cannon Water Bom...	...	281.54	1	53.4926
1	1533	Walcart	LEGO Ninja Turtl...	...	6.70	1	1.2730
2	1534	Bullseye		NaN	58.35	1	11.0865
3	1535	Bullseye	Transformers Age...	...	80.76	1	15.3444
4	1535	Bullseye	Transformers Age...	...	80.76	1	15.3444
...
14049	15581	Bullseye	AC Adapter/Power...	...	229.76	1	43.6544
14050	15582	Bullseye	Cisco Systems Gi...	...	33.39	1	6.3441
14051	15583	Understock.com	Philips AJ3116M/...	...	4.18	1	0.7942
14052	15584	iBay.com		NaN	119.50	1	22.7050
14053	15585	Understock.com	Sirius Satellite...	...	66.32	1	12.6008

[14054 rows x 14 columns]

In the output above, the last column is now called '`Tax`' instead of '`Sales Tax`'. Notice that you pass the old-to-new name mapping to `rename` as the `columns` keyword argument. This method doesn't change the original `DataFrame`, but instead returns a new `DataFrame` — to update `ledger_df` with this new column name, you need to assign the output of `rename` back to `ledger_df`:

```
In [9]: ledger_df = ledger_df.rename(columns={'Sales Tax': 'Tax'})
```

```
1
```

```
2
```

```
3
```

```
Out [9]: Index(['InvoiceNo', 'Channel', 'Product Name', 'ProductID', 'Account',
   'AccountNo', 'Date', 'Deadline', 'Currency', 'Unit Price', 'Quantity',
   'Total', 'Quarter', 'Tax'],
   dtype='object')
```

Now `ledger_df`'s last column is called '`Tax`'. You can also rename multiple columns at once:

```
In [10]: ledger_df = ledger_df.rename(columns={
    'Product Name': 'Name',
    'Unit Price': 'Price',
    'Hello': 'World'
})
```

```
1
```

```
2
```

```
3
```

```
4
```

```
5
```

```
6
```

```
7
```

```
ledger_df
```

	InvoiceNo	Channel	Name	...	Total	Quarter	Tax
0	1532	Shoppe.com	Cannon Water Bom...	...	281.54	1	53.4926
1	1533	Walcart	LEGO Ninja Turtl...	...	6.70	1	1.2730
2	1534	Bullseye		NaN	58.35	1	11.0865
3	1535	Bullseye	Transformers Age...	...	80.76	1	15.3444
4	1535	Bullseye	Transformers Age...	...	80.76	1	15.3444
...
14049	15581	Bullseye	AC Adapter/Power...	...	229.76	1	43.6544
14050	15582	Bullseye	Cisco Systems Gi...	...	33.39	1	6.3441

```
14051      15583 Understock.com Philips AJ3116M/... ... 4.18 1 0.7942
14052      15584           iBay.com             NaN ... 119.50 1 22.7050
14053      15585 Understock.com Sirius Satellite... ... 66.32 1 12.6008
```

[14054 rows x 14 columns]

If you try to rename columns that don't exist in your `DataFrame` (or misspell column names when using `rename`), you won't see any errors (in the example above, I tried renaming the '`Hello`' column, which doesn't exist in `ledger_df`). When you rename a column and don't see the change you expected, check for typos in the old-to-new column name dictionary.

Replacing values

You may have noticed that some of the values in the '`Channel`' column end with '`.com`' and some don't. It might make sense to remove this ending and keep all channel identifiers using the same naming style. To do that, you can use the `replace` method:

```
In [11]: ledger_df['Channel'].replace({
    'Shoppe.com': 'Shoppe',
    'Understock.com': 'Understock',
    'iBay.com': 'iBay'
})
```

```
Out [11]: 0          Shoppe
1          Walcart
2          Bullseye
3          Bullseye
4          Understock
...
24309      Bullseye
24310      Bullseye
24311      Understock
24312      iBay
24313      Understock
Name: Channel, Length: 24314, dtype: object
```

The code above replaces values in the '`Channel`' column based on the old-to-new mapping passed as an argument to `replace`. Its output is a new `Series`, so if you want to update `ledger_df` with these new values, you need to assign the output above back to the '`Channel`' column:

```
In [12]: ledger_df['Channel'] = ledger_df['Channel'].replace({
    'Shoppe.com': 'Shoppe',
    'Understock.com': 'Understock',
    'iBay.com': 'iBay'
})
```

Now, if you take a look at `ledger_df`, you will see the new values:

```
In [13]: ledger_df
```

```
Out [13]:
```

	InvoiceNo	Channel	Name	...	Total	Quarter	Tax
0	1532	Shoppe	Cannon Water Bom...	...	281.54	1	53.4926
1	1533	Walcart	LEGO Ninja Turtl...	...	6.70	1	1.2730
2	1534	Bullseye		NaN	58.35	1	11.0865
3	1535	Bullseye	Transformers Age...	...	80.76	1	15.3444
4	1535	Bullseye	Transformers Age...	...	80.76	1	15.3444
...
14049	15581	Bullseye	AC Adapter/Power...	...	229.76	1	43.6544
14050	15582	Bullseye	Cisco Systems Gi...	...	33.39	1	6.3441
14051	15583	Understock	Philips AJ3116M/...	...	4.18	1	0.7942
14052	15584	iBay		NaN	119.50	1	22.7050
14053	15585	Understock	Sirius Satellite...	...	66.32	1	12.6008

[14054 rows x 14 columns]

This example uses the `replace` method on a single column (i.e., a `Series` object). The `replace` method is also available on `DataFrame` objects — when you call it on a `DataFrame`, it replace values in your entire table (i.e., in all its columns, wherever they appear). Keep in mind when using `replace` — with either `Series` or `DataFrame` objects — that only exact matches get replaced.

Assigning new values to a table slice

I mentioned earlier that working with tables in `pandas` is different from working with tables in Excel: you don't modify individual table entries directly but manipulate columns or the whole table using various methods or functions. However, you sometimes need to modify a few values (and not an entire column) even when using `pandas`.

To do that, you have to use the `loc` or `iloc` operators. First, you need to specify the *location* you want to modify and then assign a new value to that selection. Let's say you want to change the '`ProductID`' value in the second row of `ledger_df` because you know it's wrong — right now, it is T&G/LEG-37777:

```
In [14]: ledger_df.head(2)
```

```
Out [14]:
```

	InvoiceNo	Channel	Name	ProductID	...	Quantity	Total	Quarter	Tax
0	1532	Shoppe	Cannon Wate...	T&G/CAN-97509	...	14	281.54	1	53.4926
1	1533	Walcart	LEGO Ninja ...	T&G/LEG-37777	...	1	6.70	1	1.2730

[2 rows x 14 columns]

To modify this table entry, you first have to select it using `loc` or `iloc`, then assign it a new value:

```
In [15]: ledger_df.loc[1, 'ProductID']
```

Out [15]: T&G/LEG-37777

```
In [16]: # with .loc
ledger_df.loc[1, 'ProductID'] = 'T&G/LEG0-0190'

# or with .iloc
ledger_df.iloc[1, 3] = 'T&G/LEG0-0190'
```

Now, if you inspect `ledger_df`, you'll see the new value in the second row of the `'ProductID'` column:

```
In [17]: ledger_df.head(2)
```

```
Out [17]:   InvoiceNo  Channel          Name  ProductID  ...  Total Quarter Total W/Out Tax
0         1532    Shoppe Cannon Wat...  T&G/CAN-97509  ...  281.54      1     228.0474
1         1533   Walcart LEGO Ninja...  T&G/LEGO-0190  ...    6.70      1      5.4270
[2 rows x 14 columns]
```

This is the only way to assign new values to table slices using pandas: first, you make a selection using `loc` or `iloc`, then you assign it a new value. Although this may seem like a complicated way of achieving something easily done in Excel, changing values in a `DataFrame` using this kind of selection leaves a clear trace of the operations applied to your data, whereas manually modifying a sheet in Excel does not. However, if you need to modify many table entries manually, it will be easier to do that in Excel than it is with pandas.

You can also select ranges of values to modify, not just single entries. For example, you can set the first three entries in the `'Quarter'` column to `'One'` using:

```
In [18]: ledger_df.loc[0:2, 'Quarter'] = 'One'
```

```
ledger_df.head()
```

```
Out [18]:   InvoiceNo  Channel          Name  ...  Total Quarter Total W/Out Tax
0         1532    Shoppe Cannon Water Bom...  ...  281.54     One     228.0474
1         1533   Walcart LEGO Ninja Turtl...  ...    6.70     One      5.4270
2         1534   Bullseye             NaN  ...  58.35     One     47.2635
3         1535   Bullseye Transformers Age...  ...  80.76      1     65.4156
4         1536  Understock Ty Beanie Boos S...  ...  780.92      1    632.5452
[5 rows x 14 columns]
```

Or if you assign your table selection a sequence of equal length, you can change multiple table entries at once:

```
In [19]: ledger_df.loc[0:2, 'Quarter'] = ['One', 'Two', 'Three']
```

```
ledger_df.head()
```

```
Out [19]:
```

	InvoiceNo	Channel	Name	...	Total	Quarter	Total	W/Out	Tax
0	1532	Shoppe	Cannon Water Bom...	...	281.54	One		228.0474	
1	1533	Walcart	LEGO Ninja Turtl...	...	6.70	Two		5.4270	
2	1534	Bullseye		NaN	58.35	Three		47.2635	
3	1535	Bullseye	Transformers Age...	...	80.76	1		65.4156	
4	1536	Understock	Ty Beanie Boos S...	...	780.92	1		632.5452	

[5 rows x 14 columns]

Both `loc` or `iloc` can be used to modify table entries. However, only `loc` can be used to filter and edit your tables at the same time — let's see how.

Filter and edit

One problem with using `loc` to change table values is that you have to know their row and column labels. What if you want to modify several values in a column and don't know the row labels for these values? In that case, you can use conditional filtering with `loc` to first find the rows you need, then assign new values in the column you want to change.

Let's assume sales in the Walcart and Bullseye channels have different sales tax rates: instead of the 19% we applied to all rows in `ledger_df` previously, Walcart sales are taxed 9%, and Bullseye sales are taxed 16%. We want to update values in the '`Tax`' column to reflect these different tax rates for the two channels.

The first thing you need to do is filter the sales data on the '`Channel`' column, using `loc`, and select values in the '`Total`' column:

```
In [20]: ledger_df.loc[ledger_df['Channel'] == 'Walcart', 'Total']
```

```
Out [20]:
```

1	6.70
7	28.29
10	19.49
11	73.68
15	4.25
	...
14012	10.10
14022	7.72
14028	3.91
14034	274.88
14039	332.64

Name: Total, Length: 1851, dtype: float64

This selects the total sales amount for those rows in `ledger_df` where the sales channel is '`Walcart`' — nothing new, yet. To calculate the new tax amount, you can use:

```
In [21]: ledger_df.loc[ledger_df['Channel'] == 'Walcart', 'Total'] * (9 / 100)
```

```
Out[21]: 1      0.6030
         7      2.5461
        10     1.7541
       11      6.6312
      15      0.3825
      ...
    14012    0.9090
   14022    0.6948
  14028    0.3519
 14034    24.7392
14039    29.9376
Name: Total, Length: 1851, dtype: float64
```

This is what we wanted to calculate, but the code above doesn't update `ledger_df` with the new tax amounts. To update the '`Tax`' column, you need to assign these values back to a slice of `ledger_df`:

```
In [22]: walcart_rows = ledger_df['Channel'] == 'Walcart'

ledger_df.loc[walcart_rows, 'Tax'] = ledger_df.loc[walcart_rows, 'Total'] * (9 / 100)
```

Or if you want to do this in one go, you can run (the extra parentheses are there to split the long line of code in two):

```
In [23]: ledger_df.loc[ledger_df['Channel'] == 'Walcart', 'Tax'] = (
    ledger_df.loc[ledger_df['Channel'] == 'Walcart', 'Total'] * (9 / 100))  
1  
2  
3
```

Here, you use `loc` twice: first, to select the values you want to update; second, to calculate new values from another column in the table. Because both selections have the same size (i.e., the same number of columns and rows), you can assign one to the other using the `=` operator. You can use similar code to update Bullseye tax amounts (but with a 16% sales tax):

```
In [24]: ledger_df.loc[ledger_df['Channel'] == 'Bullseye', 'Tax'] = (
    ledger_df.loc[ledger_df['Channel'] == 'Bullseye', 'Total'] * (16 / 100))  
1  
2  
3
```

You can use conditional filtering with `loc` to define even more complicated table slices and update table values in a similar way. While it may seem like a lot of typing to modify some values, remember that every change you make to your data this way remains visible in your code, and you can re-run these changes whenever you need to.¹

Summary

This chapter showed you how to add, rename, and modify table columns with `pandas`. There are a few `pandas` quirks you need to get used to, but the benefits of using code to modify table entries versus manually changing them are worth the learning effort.

¹: Unlike Excel, where after several filter and edit operations, there's no record of what you did and if Excel crashes, you have to do them all over again.

So far, we've been looking at our data in close detail — but what if you want to get a high-level view of your tables? The next chapter takes a closer look at the `DataFrame` and `Series` methods you can use to summarize data in `pandas`.

Summarizing data

In addition to slicing a table or adding new columns to it, you will often want to extract specific pieces of information from its data: how many unique values there are in a column, how frequently they appear in the data, what the average value of a column is, etc. These operations can be labeled as data summaries — let's take a look at how you compute them in `pandas`.

Counting unique values

To find out how many unique values there are in each column of your table, you can use the `nunique` method:

```
In [1]: ledger_df.nunique()
```

```
Out[1]: InvoiceNo      10344
Channel          5
Product Name    2064
ProductID       2132
Account          1
AccountNo        1
Date             31
Deadline        1422
Currency         1
Unit Price     2474
Quantity        135
Total           5178
dtype: int64
```

The same method works with `Series` objects, so you can compute the number of unique values in a specific column using:

```
In [2]: ledger_df['Channel'].nunique()
```

```
Out[2]: 5
```

For single columns, you can also list unique values with:

```
In [3]: ledger_df['Channel'].unique()
```

```
Out[3]: array(['Shoppe.com', 'Walcart', 'Bullseye', 'Understock.com', 'iBay.com'],
              dtype=object)
```

The output above is a list-like array: it's not a regular Python list, but for our purposes, you can use it like one: you can loop over it or access items from it using their index. Another way you can use it is to check whether a particular value is among its items:

```
In [4]: 'Walcart' in ledger_df['Channel'].unique()
```

```
Out[4]: True  
In[5]: 'Amazon.com' in ledger_df['Channel'].unique()  
Out[5]: False
```

Even more useful than listing unique items is counting how many times each value appears in a column using the `value_counts` method:

```
In[6]: ledger_df['Channel'].value_counts()
```

```
Out[6]: Understock.com    4821  
Shoppe.com        3132  
iBay.com          2877  
Walcart           1851  
Bullseye          1373  
Name: Channel, dtype: int64
```

The output above is a `Series` object, so if you want the number of times '`Walcart`' appears in the '`Channel`' column, you can access it with:

```
In[7]: ledger_df['Channel'].value_counts().loc['Walcart']
```

```
Out[7]: 1851
```

Instead of raw counts, you might want to calculate the proportion of times each value appears in a column. In that case, you can use the `normalize` keyword argument with `value_counts` and tell `pandas` to divide all counts by their sum:

```
In[8]: ledger_df['Channel'].value_counts(normalize=True)
```

```
Out[8]: Understock.com    0.343034  
Shoppe.com        0.222855  
iBay.com          0.204710  
Walcart           0.131706  
Bullseye          0.097695  
Name: Channel, dtype: float64
```

Notice that the values returned now sum to 1. If you want percentages (i.e., make the values sum to 100 instead of 1), you can multiply the output above by 100:

```
In[9]: ledger_df['Channel'].value_counts(normalize=True) * 100
```

```
Out[9]: Understock.com    34.303401  
Shoppe.com        22.285470  
iBay.com          20.471040  
Walcart           13.170628  
Bullseye          9.769461  
Name: Channel, dtype: float64
```

This tells you that roughly 34.3% of rows in `ledger_df` have '`Understock.com`' as their '`Channel`' value.

Counting values works for multiple columns as well (i.e., it is available on both `Series` and `DataFrame` objects). For instance, to count how often different combinations of '`Channel`' and '`ProductID`' values show up in `ledger_df`, you can use:

```
In [10]: ledger_df[['Channel', 'ProductID']].value_counts()
```

```
Out [10]: Channel      ProductID
Understock.com  I&S/SIN-63159    19
                  MI/LAN-23501    18
                  H&PC/DUR-24730    16
Shoppe.com      K&D/AS-73604    15
Understock.com  M&T/BAB-42333    15
                  ...
iBay.com        E/NEW-91184     1
Understock.com  C&P/NEE-31972    1
                  C&P/NEW-62681     1
iBay.com        E/NEW-08016     1
                  T&G/^DE-22075     1
Length: 5010, dtype: int64
```

Averages and numerical summaries

Besides counts of unique values, you can compute a range of mathematical summaries of numerical data stored in `DataFrame` or `Series` objects. For example, to compute the average value in the '`Quantity`' column, you can use:

```
In [11]: ledger_df['Quantity'].mean()
```

```
Out [11]: 11.249893268820264
```

Just like `value_counts` works with multiple columns, you can use `mean` on several numerical columns at the same time as well:

```
In [12]: ledger_df[['Unit Price', 'Quantity', 'Total']].mean()
```

```
Out [12]: Unit Price    12.752201
          Quantity     11.249893
          Total         129.728658
          dtype: float64
```

This output is a `Series` object, with each item representing the average value in one of the listed columns.

In addition to `mean`, there are several other methods available on both `DataFrame` and `Series` objects you can use to extract summary statistics from numerical columns — table 18.1 lists some of them.

Table 18.1: Mathematical methods you can use to summarize numerical values in both `Series` and `DataFrame` objects.

Method	Example	Output	Description
<code>sum</code>	<code>ledger_df['Total'].sum()</code>	1823206.56	Returns the sum of values in the ' <code>Total</code> ' column.
<code>mean</code>	<code>ledger_df['Quantity'].mean()</code>	11.2498	Returns the average of values in the ' <code>Quantity</code> ' column.
<code>median</code>	<code>ledger_df['Unit Price'].median()</code>	10.22	Returns the median of values in the ' <code>Unit Price</code> ' column.
<code>max</code>	<code>ledger_df['Unit Price'].max()</code>	166.3	Returns the maximum value in the ' <code>Unit Price</code> ' column.
<code>min</code>	<code>ledger_df['Total'].min()</code>	0.8999	Returns the minimum value in the ' <code>Total</code> ' column.
<code>idxmax</code>	<code>ledger_df['Unit Price'].idxmax()</code>	616	Returns the row label for the maximum value in the ' <code>Unit Price</code> ' column.
<code>idxmin</code>	<code>ledger_df['Total'].idxmin()</code>	7613	Returns the row label for the minimum value in the ' <code>Total</code> ' column.
<code>count</code>	<code>ledger_df['Channel'].count()</code>	14054	Returns the number of non-empty (i.e., non- <code>NaN</code>) values in the ' <code>Channel</code> ' column. This method works with non-numerical values as well.
<code>quantile</code>	<code>ledger_df['Total'].quantile(.25)</code>	19.26	Computes the value at a quantile passed as an argument (values between 0 and 1 are valid arguments). This example tells you that 25% of values in the ' <code>Total</code> ' column are less than 19.26.

In addition to the methods listed above, both `pandas` objects (i.e., `Series` and `DataFrame`) have a method called `describe` that computes multiple summary statistics. If you use it on `ledger_df`, you will get a new `DataFrame` with the minimum, maximum, mean, and other summary values for all the numerical columns in `ledger_df`:

```
In [13]: ledger_df.describe()
```

```
Out [13]:
```

	InvoiceNo	AccountNo	Unit Price	Quantity	Total
count	14054.000000	14054.0	14054.000000	14054.000000	14054.000000
mean	8510.825103	5004.0	12.752201	11.249893	129.728658
std	4059.091864	0.0	10.278616	17.931963	251.303321
min	1532.000000	5004.0	0.060000	1.000000	0.900000
25%	4976.250000	5004.0	5.490000	2.000000	19.260000
50%	8495.000000	5004.0	10.220000	6.000000	46.340000
75%	12061.750000	5004.0	16.980000	13.000000	127.750000
max	15585.000000	5004.0	166.300000	226.000000	4362.200000

Even though the '`InvoiceNo`' and '`AccountNo`' columns store numerical values, they don't have numerical interpretation: in the output above, the average '`InvoiceNo`' value doesn't mean anything. In general, you will want to first select a subset of columns and then run `describe`, or run `describe` on just a single column:

```
In [14]: ledger_df[['Unit Price', 'Quantity', 'Total']].describe()
```

```
Out [14]:    Unit Price      Quantity      Total
count  14054.000000  14054.000000  14054.000000
mean    12.752201     11.249893   129.728658
std     10.278616     17.931963   251.303321
min     0.060000      1.000000    0.900000
25%    5.490000      2.000000    19.260000
50%    10.220000      6.000000    46.340000
75%    16.980000      13.000000   127.750000
max    166.300000     226.000000   4362.200000
```

```
In [15]: ledger_df['Total'].describe()
```

```
Out [15]: count    14054.000000
mean     129.728658
std      251.303321
min      0.900000
25%     19.260000
50%     46.340000
75%     127.750000
max     4362.200000
Name: Total, dtype: float64
```

High-level table information

We've already used the `DataFrame.info` method in earlier chapters to get a high-level view of a `DataFrame`. If you use it with `ledger_df`, you get the following output:

```
In [16]: ledger_df.info()
```

```
Out [16]: <class 'pandas.core.frame.DataFrame'>
RangeIndex: 14054 entries, 0 to 14053
Data columns (total 12 columns):
 #   Column        Non-Null Count  Dtype  
--- 
 0   InvoiceNo     14054 non-null   int64  
 1   Channel       14054 non-null   object  
 2   Product Name  12362 non-null   object  
 3   ProductID     14054 non-null   object  
 4   Account       14054 non-null   object  
 5   AccountNo     14054 non-null   int64  
 6   Date          14054 non-null   datetime64[ns]
 7   Deadline      14054 non-null   object  
 8   Currency      14054 non-null   object  
 9   Unit Price    14054 non-null   float64 
 10  Quantity      14054 non-null   int64  
 11  Total         14054 non-null   float64 
dtypes: datetime64[ns](1), float64(2), int64(3), object(6)
memory usage: 1.3+ MB
```

The output above is formatted text, not a `Series` or a `DataFrame` object like the output of other methods covered in this chapter.

There are several pieces of information about `ledger_df` above: how many rows it has, what their row labels are (`RangeIndex: 14054 entries, 0 to 14053` tells you that rows have integer labels in consecutive order, from 0 to 14053), how many columns (`Data columns (total 12 columns)`) and for each column, its name, the number of non-empty values (i.e., non-NaN values, under `Non-Null Count`) and the type of data it stores (under `Dtype`). Numerical columns with whole numbers have `int64` as their data type (e.g., `'Quantity'`), those with decimal numbers have `float64` (e.g., `'Unit Price'`), columns that store dates have `datetime64[ns]` (e.g., `'Date'`), and text columns have the `object Dtype` (e.g., `'Product Name'`). The last two lines in the output of `info` tell you how many columns there are in the table with each data type, and approximately how much of your computer's memory is used to store this table.¹

While most columns in `ledger_df` have the same number of non-empty values as the total number of rows (i.e., 14054), notice in the output above that the `'Product Name'` column has only 12362 non-null values — you may have noticed some empty values (i.e., NaN values) in the `'Product Name'` column. Missing data can be a problem and is one of the first things you need to address when analyzing new data — which is what the next chapter looks at.

¹: This estimate is actually not very good. You can get a more accurate estimate of how much memory is used to store your `DataFrame` with `ledger_df.info(memory_usage='deep')`. In this case, it is 9.8 MB.

Summary

This chapter went over several `pandas` methods you can use to summarize your data. Most of these methods are available on both `Series` and `DataFrame` objects, so you can use them to summarize single columns or entire tables at once.

Now that you know how to find missing values in a `DataFrame` with the `info` method, let's see how you can fill them in or filter them out next.

Cleaning data

19

Missing values, duplicate rows, numbers or dates stored as text are just some of the common problems you face when working with any data, including balance sheets or ledgers. Fixing these problems is generally labeled as data cleaning — and infamously takes up most of a data worker’s time. Fortunately, the designers of pandas know how real-world data looks like and equipped it with a set of data cleaning tools.

Dealing with missing values

In the real-world, table entries tend to go missing for a variety of reasons: corrupt data storage, errors in data processing, cannot-compute values (e.g., a number that can be computed for certain rows but not for others). When you start working with a new dataset, you will often need to find and deal with empty values before doing anything else.

You can either keep missing values as they are, remove the rows or columns they appear in, or fill them in with some other appropriate value. Fortunately, pandas makes all of these tasks straightforward.

What isn’t as straightforward is how pandas represents missing values — there are three different *sentinel values* used to indicate missing values in a DataFrame or Series: `NaN`, `NaT`, and `<NA>`. If you take a look at top rows in `ledger_df`, there’s one in the ‘`Product Name`’ column:

```
In [1]: ledger_df.head()
```

```
Out[1]:   InvoiceNo      Channel          Product Name ... Unit Price Quantity  Total
0        1532  Shoppe.com    Cannon Water Bomb Ba... ...  20.11      14  281.54
1        1533     Walcart    LEGO Ninja Turtles S... ...   6.70       1   6.70
2        1534     Bullseye           NaN ...       11.67      5  58.35
3        1535     Bullseye  Transformers Age of ... ...   13.46      6  80.76
4        1535     Bullseye  Transformers Age of ... ...   13.46      6  80.76
```

[5 rows x 12 columns]

To make things even more confusing, when you inspect this value in a separate cell, it gets printed as `nan` instead of `NaN`:

```
In [2]: ledger_df.iloc[2, 2]
```

```
Out [2]: nan
```

Together with Python's built-in `None` value, there sure are a lot of ways to represent nothing in Python and pandas.

However, all these sentinel values mean the same thing: value not available. In numerical or text column, pandas uses `NaN` to indicate empty values (which stands for *not a number*), whereas in date columns, it uses `NaT` (which stands for *not a timestamp*). Recently, pandas introduced the `<NA>` value, which is supposed to replace both `NaN` and `NaT` as the only "*not available*" value in future versions.¹

The closest Excel equivalent to pandas's sentinel values is the `#N/A` error value. However, Excel's `#N/A` is a cannot-compute value, which gets shown whenever a function cannot produce a value, rather than a missing value indicator (missing values in Excel are just empty cells). There is no cannot-compute value in pandas; whenever pandas can't compute something, you'll get an error.

After much ado about nothing, let's see how you can find these missing values in `ledger_df`.

1: The reason for so many variations of the same thing is fairly technical, but you can read more about it at pandas.pydata.org/pandas-docs/stable/user_guide/missing_data.html.

Finding missing values

You can use the `isna` method, available on both `Series` and `DataFrame` objects, to detect missing values:

```
In [3]: ledger_df['Product Name'].isna()
```

```
Out [3]: 0      False
1      False
2      True
3     False
4     False
...
14049    False
14050    False
14051    False
14052    True
14053    False
Name: Product Name, Length: 14054, dtype: bool
```

The output of `isna` is a familiar boolean `Series`, with the same number of rows as the `'Product Name'` column (i.e., containing `True` for any value in `'Product Name'` that is `None`, `NaN`, `NaT` or `<NA>`). You can use this boolean `Series` to filter `ledger_df`:

```
In [4]: ledger_df[ledger_df['Product Name'].isna()]
```

Out [4]:	InvoiceNo	Channel	Product Name	...	Unit Price	Quantity	Total
2	1534	Bullseye	NaN	...	11.67	5	58.35
6	1538	Understock.com	NaN	...	31.36	9	282.24
16	1548	iBay.com	NaN	...	8.07	5	40.35
35	1567	Understock.com	NaN	...	12.30	5	61.50
36	1568	Walcart	NaN	...	3.52	1	3.52
...
14021	15553	iBay.com	NaN	...	17.28	23	397.44
14025	15443	Understock.com	NaN	...	16.49	103	1698.47
14036	15495	Bullseye	NaN	...	14.85	9	133.65
14040	15572	iBay.com	NaN	...	16.76	6	100.56
14052	15584	iBay.com	NaN	...	4.78	25	119.50

[1692 rows x 12 columns]

Or if you just want to know how many missing values there are, because booleans are just numbers in disguise, you can sum the output of `isna`:

In [5]: `ledger_df['Product Name'].isna().sum()`

Out [5]: 1692

The `isna` method is available on `DataFrame` objects as well. Instead of checking each column separately, you can run `isna` on `ledger_df` directly, to get the number of missing values in each column at the same time:

In [6]: `ledger_df.isna().sum()`

Out [6]:

InvoiceNo	0
Channel	0
Product Name	1692
ProductID	0
Account	0
AccountNo	0
Date	0
Deadline	0
Currency	0
Unit Price	0
Quantity	0
Total	0
<code>dtype: int64</code>	

Even though the `info` method produces a similar output (i.e., it shows the number of non-empty values in each column), summing the output of `isna` can make missing values easier to spot.

The sales ledger has a lot of missing values in the '`Product Name`' column. Let's see what you can do with them.

Discarding missing values

The first option you might consider when coming across missing values is to remove them from your `DataFrame` — especially if you have entire rows or columns made up of empty values.

You can filter all rows in `ledger_df` where '`Product Name`' values are empty using:

```
In [7]: ledger_df[ledger_df['Product Name'].notna()]
```

```
Out [7]:
```

	InvoiceNo	Channel	Product Name	Unit Price	Quantity	Total
0	1532	Shoppe.com	Cannon Water Bomb Ba...	20.11	14	281.54
1	1533	Walcart	LEGO Ninja Turtles S...	6.70	1	6.70
3	1535	Bullseye	Transformers Age of ...	13.46	6	80.76
4	1535	Bullseye	Transformers Age of ...	13.46	6	80.76
5	1537	Bullseye	3x Anti-Spy Privacy ...	7.39	8	59.12
...
14048	15580	iBay.com	Lauri Toddler Tote	14.46	1	14.46
14049	15581	Bullseye	AC Adapter/Power Sup...	28.72	8	229.76
14050	15582	Bullseye	Cisco Systems Gigabi...	33.39	1	33.39
14051	15583	Understock.com	Philips AJ3116M/37 D...	4.18	1	4.18
14053	15585	Understock.com	Sirius Satellite Rad...	33.16	2	66.32

[12362 rows x 12 columns]

As with most examples so far, you need to assign this output back to `ledger_df` if you want to make this filter permanent.

The `notna` above is the negation of `isna`'s output — these two lines of code produce the same boolean `Series` (notice the tilde character on line 4):

```
In [8]: # these two lines of code
# produce the same output
1
2
3
4
5
~ledger_df['Product Name'].isna()
ledger_df['Product Name'].notna()
```

```
Out [8]:
```

0	True
1	True
2	False
3	True
4	True
...	...
14049	True
14050	True
14051	True
14052	False
14053	True

Name: Product Name, Length: 14054, dtype: bool

More generally, you can discard all empty values using the `dropna` method. By default, the `dropna` method discards all rows that have empty values in any of the columns:

```
In [9]: ledger_df.dropna()
```

```
Out [9]:
```

	InvoiceNo	Channel	Product Name	...	Unit Price	Quantity	Total
0	1532	Shoppe.com	Cannon Water Bomb Ba...	...	20.11	14	281.54
1	1533	Walcart	LEGO Ninja Turtles S...	...	6.70	1	6.70
3	1535	Bullseye	Transformers Age of	13.46	6	80.76
4	1535	Bullseye	Transformers Age of	13.46	6	80.76
5	1537	Bullseye	3x Anti-Spy Privacy	7.39	8	59.12
...
14048	15580	iBay.com	Lauri Toddler Tote	...	14.46	1	14.46
14049	15581	Bullseye	AC Adapter/Power Sup...	...	28.72	8	229.76
14050	15582	Bullseye	Cisco Systems Gigabi...	...	33.39	1	33.39
14051	15583	Understock.com	Philips AJ3116M/37 D...	...	4.18	1	4.18
14053	15585	Understock.com	Sirius Satellite Rad...	...	33.16	2	66.32

[12362 rows x 12 columns]

In this case, the code above does the same thing as the previous example: discards all rows in `ledger_df` where '`Product Name`' values are empty.

In general, `dropna` is useful when you have rows with missing values in multiple columns, and you need to discard all of those rows at once, without going through each column one by one.

You can change `dropna`'s behavior with several keyword arguments. For instance, you can tell it to look for missing values in a subset of a DataFrame's columns using the `subset` keyword argument:

```
In [10]: ledger_df.dropna(subset=['Product Name', 'ProductID'])
```

```
Out [10]:
```

	InvoiceNo	Channel	Product Name	...	Unit Price	Quantity	Total
0	1532	Shoppe.com	Cannon Water Bomb Ba...	...	20.11	14	281.54
1	1533	Walcart	LEGO Ninja Turtles S...	...	6.70	1	6.70
3	1535	Bullseye	Transformers Age of	13.46	6	80.76
4	1535	Bullseye	Transformers Age of	13.46	6	80.76
5	1537	Bullseye	3x Anti-Spy Privacy	7.39	8	59.12
...
14048	15580	iBay.com	Lauri Toddler Tote	...	14.46	1	14.46
14049	15581	Bullseye	AC Adapter/Power Sup...	...	28.72	8	229.76
14050	15582	Bullseye	Cisco Systems Gigabi...	...	33.39	1	33.39
14051	15583	Understock.com	Philips AJ3116M/37 D...	...	4.18	1	4.18
14053	15585	Understock.com	Sirius Satellite Rad...	...	33.16	2	66.32

[12362 rows x 12 columns]

This looks for empty values in only the two columns specified as the `subset` argument and discards all rows with empty values in those two columns.

In addition to `subset`, you also have the `how` keyword argument, which controls how many missing values a row needs to have to be discarded. We don't have any entirely empty rows in `ledger_df` right now, but as an example of using the `how` keyword argument:

```
In [11]: ledger_df.dropna(how='all')
```

This example would discard only those rows from `ledger_df` that are entirely empty (i.e., values in all columns are `NaN`).

Instead of rows, you can also tell `dropna` to discard *columns* with empty values instead, using the `axis` keyword argument. To discard all columns in `ledger_df` that have *any* empty values in them, you can use:

```
In [12]: ledger_df.dropna(how='any', axis='columns')
```

```
Out [12]:   InvoiceNo      Channel    ProductID ... Unit Price  Quantity  Total
0        1532  Shoppe.com  T&G/CAN-97509 ...  20.11      14  281.54
1        1533     Walcart  T&G/LEG-37777 ...   6.70       1   6.70
2        1534    Bullseye  T&G/PET-14209 ...  11.67       5  58.35
3        1535    Bullseye  T&G/TRA-20170 ...  13.46       6  80.76
4        1535    Bullseye  T&G/TRA-20170 ...  13.46       6  80.76
...
14049     ...        ...       ...  ...  ...  ...
14050     15581    Bullseye  E/AC-63975 ...  28.72       8 229.76
14050     15582    Bullseye  E/CIS-74992 ...  33.39       1 33.39
14051     15583  Understock.com  E/PHI-08100 ...   4.18       1   4.18
14052     15584     iBay.com  E/POL-61164 ...   4.78      25 119.50
14053     15585  Understock.com  E/SIR-83381 ...  33.16       2  66.32
```

[14054 rows x 11 columns]

Notice in the output above that instead of rows with empty values being dropped, the '`Product Name`' column was discarded entirely because it contained several `NaN` values.

As with all the other `DataFrame` methods, you can combine `dropna`'s keyword arguments in any way to get the result you need. Right now, they might not seem particularly useful, but in the next project chapter we'll take a look at how you can work with a general ledger in `pandas`, and you will see how practical `dropna` and its keyword arguments are.

Filling missing values

Instead of discarding rows or columns with missing values, you can also fill in those values. If you already know what to fill in missing values with, you can use the `fillna` method and pass the fill value as an argument:

```
In [13]: ledger_df['Product Name'].fillna('MISSING')
```

```
Out [13]: 0    Cannon Water Bomb Ba...
1    LEGO Ninja Turtles S...
2           MISSING
3    Transformers Age of ...
4    Transformers Age of ...
...
```

```

14049    AC Adapter/Power Sup...
14050    Cisco Systems Gigabi...
14051    Philips AJ3116M/37 D...
14052                  MISSING
14053    Sirius Satellite Rad...
Name: Product Name, Length: 14054, dtype: object

```

While not much of an improvement over the `Nan` values we had before, empty values in the '`Product Name`' column have been now filled with '`MISSING`'.

Instead of a predefined value, you can tell `fillna` to fill in empty values with the last valid entry in a column. The following code fills in missing values in the '`Product Name`' column with the first valid value (i.e., non-`Nan` value) found in the rows above:

```
In [14]: ledger_df['Product Name'].fillna(method='ffill')
```

```

Out [14]: 0      Cannon Water Bomb Ba...
1      LEGO Ninja Turtles S...
2      LEGO Ninja Turtles S...
3      Transformers Age of ...
4      Transformers Age of ...
...
14049    AC Adapter/Power Sup...
14050    Cisco Systems Gigabi...
14051    Philips AJ3116M/37 D...
14052    Philips AJ3116M/37 D...
14053    Sirius Satellite Rad...
Name: Product Name, Length: 14054, dtype: object

```

Notice that the third value in the output above, which previously was a `Nan`, is now a copy of the value right above it. Similarly, in the row labeled 14 052 (i.e., second row from the bottom), the value is a copy of the value directly above it.

You can tell `fillna` to use the first valid value found below, rather than above, to fill in missing values.²

```
In [15]: ledger_df['Product Name'].fillna(method='bfill')
```

```

Out [15]: 0      Cannon Water Bomb Ba...
1      LEGO Ninja Turtles S...
2      Transformers Age of ...
3      Transformers Age of ...
4      Transformers Age of ...
...
14049    AC Adapter/Power Sup...
14050    Cisco Systems Gigabi...
14051    Philips AJ3116M/37 D...
14052    Sirius Satellite Rad...
14053    Sirius Satellite Rad...
Name: Product Name, Length: 14054, dtype: object

```

2: The two keyword values, '`ffill`' and '`bfill`' stand for *forward fill* and *back fill*.

As in the previous example, the two missing values in the 'Product Name' column (i.e., in rows labeled 2 and 14 052) are now filled in with copies, but copies of the values below them, instead of the ones above.

The `axis` keyword argument is also available with `fillna`: when you want to fill values across columns, rather than down the rows of a `DataFrame`, you can pass the `axis='columns'` keyword argument to `fillna`.

Dealing with duplicate rows

Messy data often includes duplicate rows and columns (particularly if your dataset is put together by manually copying rows from different spreadsheets).

To detect if entire rows are duplicated in a `DataFrame`, you can use the `duplicated` method:

```
In [16]: ledger_df.duplicated()
```

```
Out [16]: 0      False
1      False
2      False
3      False
4      True
...
14049  False
14050  False
14051  False
14052  False
14053  False
Length: 14054, dtype: bool
```

Notice in the output above that the value in the fifth row is `True`. This method goes through each row in the `DataFrame`, and checks it against all the rows above it. If `duplicated` finds another row with the same values (in all columns) above any given row, it puts a `True` value in the boolean `Series` it returns — in this case, rows labeled 3 and 4 in `ledger_df` are duplicates:

```
In [17]: ledger_df.loc[3] == ledger_df.loc[4]
```

```
Out [17]: InvoiceNo      True
Channel        True
Product Name   True
ProductID      True
Account        True
AccountNo     True
Date           True
Deadline       True
Currency       True
```

```
Unit Price      True
Quantity       True
Total          True
dtype: bool
```

Notice that only row 4 is marked as a duplicate in the output of `duplicated`, whereas row 3 is not. If you want to mark all duplicate rows, including the first row that appears in the `DataFrame` with those values, you can use the `keep` keyword argument:

```
In [18]: ledger_df.duplicated(keep=False)
```

```
Out [18]: 0      False
1      False
2      False
3      True
4      True
...
14049    False
14050    False
14051    False
14052    False
14053    False
Length: 14054, dtype: bool
```

Now both row 3 and row 4 are marked as duplicates.

The output of `duplicated` is most useful when you want to discard duplicate rows from a `DataFrame`. For instance, you can use it to discard duplicate rows from `ledger_df`:

```
In [19]: ledger_df[~ledger_df.duplicated()]
```

```
Out [19]:   InvoiceNo     Channel      Product Name ... Unit Price Quantity  Total
0        1532  Shoppe.com  Cannon Water Bomb Ba... ...  20.11      14  281.54
1        1533      Walcart  LEGO Ninja Turtles S... ...   6.70       1   6.70
2        1534      Bullseye           NaN ...  11.67       5  58.35
3        1535      Bullseye  Transformers Age of ... ...  13.46       6  80.76
5        1537      Bullseye  3x Anti-Spy Privacy ... ...   7.39       8  59.12
...
14049     ...      ...           ... ...  ... ...
14050     15581      Bullseye  AC Adapter/Power Sup... ...  28.72       8 229.76
14050     15582      Bullseye  Cisco Systems Gigabi... ...  33.39       1  33.39
14051     15583  Understock.com  Philips AJ3116M/37 D... ...   4.18       1   4.18
14052     15584      iBay.com           NaN ...  4.78      25 119.50
14053     15585  Understock.com  Sirius Satellite Rad... ...  33.16       2  66.32
[13987 rows x 12 columns]
```

Take a look at the row labels above and notice that label 4 (and its associated row) is no longer in the table.

Just like the `dropna` method simplifies discarding empty values, the `drop_duplicates` method makes removing duplicate rows easier than using `duplicated`. To get the same result as above, and discard duplicate rows from `ledger_df`, you can use:

```
In [20]: ledger_df.drop_duplicates()
```

```
Out [20]:
```

	InvoiceNo	Channel	Product Name	...	Unit Price	Quantity	Total
0	1532	Shoppe.com	Cannon Water Bomb Ba...	...	20.11	14	281.54
1	1533	Walcart	LEGO Ninja Turtles S...	...	6.70	1	6.70
2	1534	Bullseye		NaN	11.67	5	58.35
3	1535	Bullseye	Transformers Age of	13.46	6	80.76
5	1537	Bullseye	3x Anti-Spy Privacy	7.39	8	59.12
...
14049	15581	Bullseye	AC Adapter/Power Sup...	...	28.72	8	229.76
14050	15582	Bullseye	Cisco Systems Gigabi...	...	33.39	1	33.39
14051	15583	Understock.com	Philips AJ3116M/37 D...	...	4.18	1	4.18
14052	15584	iBay.com		NaN	4.78	25	119.50
14053	15585	Understock.com	Sirius Satellite Rad...	...	33.16	2	66.32

[13987 rows x 12 columns]

You get the same result as above, but with less typing. You can also tell pandas to check for duplicates in a subset of columns only, by passing a list of column names to the `subset` keyword argument:

```
In [21]: ledger_df.drop_duplicates(subset=['InvoiceNo', 'ProductID'])
```

```
Out [21]:
```

	InvoiceNo	Channel	Product Name	...	Unit Price	Quantity	Total
0	1532	Shoppe.com	Cannon Water Bomb Ba...	...	20.11	14	281.54
1	1533	Walcart	LEGO Ninja Turtles S...	...	6.70	1	6.70
2	1534	Bullseye		NaN	11.67	5	58.35
3	1535	Bullseye	Transformers Age of	13.46	6	80.76
5	1537	Bullseye	3x Anti-Spy Privacy	7.39	8	59.12
...
14049	15581	Bullseye	AC Adapter/Power Sup...	...	28.72	8	229.76
14050	15582	Bullseye	Cisco Systems Gigabi...	...	33.39	1	33.39
14051	15583	Understock.com	Philips AJ3116M/37 D...	...	4.18	1	4.18
14052	15584	iBay.com		NaN	4.78	25	119.50
14053	15585	Understock.com	Sirius Satellite Rad...	...	33.16	2	66.32

[13986 rows x 12 columns]

In the output above, rows that have duplicate values in the `'InvoiceNo'` and `'ProductID'` columns have been discarded.

You can use the `subset` and `keep` keyword arguments with both `duplicated` and `drop_duplicated` (i.e., to specify which columns to use when checking for duplicates and whether to keep one of the duplicate rows or not).

Converting column data types

When you create a new `DataFrame` by reading an Excel spreadsheet with `read_excel`, pandas tries to figure out what type of data is stored in each column and assign the right data type to each `DataFrame` column (i.e., `int64`, `float64`, `datetime` or `object`).

Sometimes pandas can't figure out the right data type for a column — because there's something wrong with the column values (e.g., numbers and text values mixed in the same column) — and assigns it the most generic type it can: the `object` data type.

If you know that a column should have a certain data type (e.g., it has only numerical value) you can change its assigned type using the `astype` method. For instance, to change the '`Quantity`' column type to `float64` instead of `int64`, you can use:

```
In [22]: ledger_df['Quantity'].astype('float')
```

```
Out [22]: 0      14.0
1      1.0
2      5.0
3      6.0
4      6.0
...
14049    8.0
14050    1.0
14051    1.0
14052   25.0
14053    2.0
Name: Quantity, Length: 14054, dtype: float64
```

This turns all values in the '`Quantity`' column from whole numbers to decimal numbers. The `astype` method is particularly useful after cleaning columns that store numbers as text values.

Another useful `DataFrame` method that can fix issues with column data types is `convert_dtypes`. This method asks pandas to try and figure out better data types for each column in a `DataFrame`, and can often reduce the amount of computer memory needed to store your data:

```
In [23]: ledger_df.convert_dtypes().info()
```

```
Out [23]: <class 'pandas.core.frame.DataFrame'>
RangeIndex: 24024 entries, 0 to 24023
Data columns (total 12 columns):
 #   Column        Non-Null Count  Dtype  
 ---  -- 
 0   InvoiceNo     24024 non-null   Int64  
 1   Channel       24024 non-null   string 
 2   Product Name  21113 non-null   string 
 3   ProductID     24024 non-null   string 
 4   Account       24024 non-null   string 
 5   AccountNo     24024 non-null   Int64  
 6   Date          24024 non-null   datetime64[ns]
 7   Deadline      24024 non-null   datetime64[ns]
 8   Currency      24024 non-null   string 
 9   Unit Price    24024 non-null   float64 
 10  Quantity      24024 non-null   Int64  
 11  Total         24024 non-null   float64 
dtypes: Int64(3), datetime64[ns](2), float64(2), string(5)
```

memory usage: 2.3 MB

I chained the `info` method right after calling `convert_dtypes` to show you how `convert_dtypes` changes `ledger_df`. Notice in the output above that several columns in `ledger_df` were converted to the `string` data type.³ The `string` data type is a recent addition to pandas, which is why it doesn't get assigned to columns when you read data from Excel. It is a more explicit data type than the `object` type because it tells you that a column contains only string values (whereas `object` columns can contain anything, even mixed values, such as numbers and text).

The `convert_dtypes` method also replaces all missing values in the DataFrame (i.e., any `NaN`, `Nat` or `None` values) with the unified `<NA>` sentinel value:

In [24]: `ledger_df.convert_dtypes()`

Out [24]:

	InvoiceNo	Channel	Product Name	Unit Price	Quantity	Total
0	1532	Shoppe.com	Cannon Water Bomb Ba...	20.11	14	281.54
1	1533	Walcart	LEGO Ninja Turtles S...	6.70	1	6.70
2	1534	Bullseye	<NA>	11.67	5	58.35
3	1535	Bullseye	Transformers Age of ...	13.46	6	80.76
4	1535	Bullseye	Transformers Age of ...	13.46	6	80.76
...
14049	15581	Bullseye	AC Adapter/Power Sup...	28.72	8	229.76
14050	15582	Bullseye	Cisco Systems Gigabi...	33.39	1	33.39
14051	15583	Understock.com	Philips AJ3116M/37 D...	4.18	1	4.18
14052	15584	iBay.com	<NA>	4.78	25	119.50
14053	15585	Understock.com	Sirius Satellite Rad...	33.16	2	66.32

[14054 rows x 12 columns]

The changes produced by calling `convert_dtypes` are not critical, and you can still work with `ledger_df` without using `convert_dtypes`. However, calling `convert_dtypes` on a DataFrame can make it use less computer memory, and for large tables, that can speed up other table operations later on.

Summary

This chapter showed you how to deal with missing values and duplicates in a pandas DataFrame. We also took a quick look at the `convert_dtypes` method, which you can use to nudge pandas into finding better column data types for your tables.

We've covered many different pandas features in the last few chapters, some of which may seem somewhat abstract right now. Let's see how useful these features can be in the accounting world by reading and cleaning a general ledger Excel file with pandas.

3: The '`InvoiceNo`', '`AccountNo`' and '`Quantity`' columns also get converted to the `Int64` data type. `Int64`, unlike the `int64` data type you have seen before, is a recent addition to pandas and allows for the presence of missing values in numerical columns that store whole numbers. However, for most practical uses, `int64` and `Int64` are identical.

Project: Reading and cleaning a QuickBooks general ledger

20

A company's general ledger is rich with information: revenues, expenses, balances, adjustments are all recorded in the GL. However, as rich as it is with information, the general ledger is often tricky to work with: its format and organization make turning records into insights far more complicated than it needs to be.

This project chapter shows you how to use the `pandas` tools we've covered so far to clean and reformat a general ledger exported from QuickBooks. At the end of this chapter, you'll have a clean general ledger `DataFrame` that is easy to slice, filter, or handle in any way.

The QuickBooks general ledger export you'll be working with is in a file called "*QuickBooks GL.xlsx*" — it contains all postings for a fictitious company named *Carl's Design and Landscaping*. You'll find "*QuickBooks GL.xlsx*" in your *Python for Accounting* workspace under the `project_data` folder in "*P2 - Working with tables*". When you open "*QuickBooks GL.xlsx*" in Excel, you should see something similar to figure 20.1 below:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1		Carl's Design and Landscaping Services General Ledger All Dates																		
2		Acct	SubAcct	Date	Transaction Type	Num		Name		Memo/Description		Debit	Credit	Balance						
3		Checking																		
4																				
5																				
6																				
7																				
8				08/21/2020	Deposit															
9				10/02/2020	Bill Payment (Check)	10	Robertson & Associates			Opening Balance		5,000.00								
10				10/10/2020	Payment	1053	Bill's Windsurf Shop						300.00	4,700.00						
11				10/24/2020	Expense	12	Robertson & Associates						175.00							
12				11/15/2020	Check	4	Chin's Gas and Oil							54.55	4,570.45					
13				11/21/2020	Sales Tax Payment					Q1 Payment			38.50	4,531.95						
14				11/21/2020	Sales Tax Payment					Q1 Payment			38.40	4,493.55						
15				11/24/2020	Check	12	Books by Bessie						55.00	4,438.55						
16				11/24/2020	Expense	9	Tania's Nursery						88.09	4,349.46						
17				11/30/2020	Check	5	Chin's Gas and Oil							62.01	4,287.45					
18				12/01/2020	Expense	15	Tania's Nursery							108.09	4,179.36					
19				12/10/2020	Sales Receipt	1008	Kate Whelan						225.00							
20				12/10/2020	Payment	5664	Freeman Sporting Goods:55 Twin Lane						86.40							

Figure 20.1: "*QuickBooks GL.xlsx*" opened in Excel.

To get started, launch JupyterLab and open this chapter's notebook (you'll find the notebook in the same the "*P2 - Working with tables*" workspace folder). As always, the first line of code you need to run in your new notebook is:

```
In [1]: import pandas as pd
```

You may have noticed above that "*QuickBooks GL.xlsx*" isn't analysis-friendly: there are empty columns between data columns, the table headers are on the fifth row, there are lots of missing values throughout the data, etc. If you load "*QuickBooks GL.xlsx*" with `read_excel`, you'll get an unwieldy `DataFrame`:

```
In [2]: ledger_df = pd.read_excel('project_data/QuickBooks GL.xlsx')
```

```
ledger_df
```

```
Out[2]: Unnamed: 0 Carl's Design and Landscaping Services ... Unnamed: 18 Unnamed: 19
0      NaN           General Ledger     ...      NaN      NaN
1      NaN           All Dates       ...      NaN      NaN
2      NaN             NaN          ...      NaN      NaN
3      NaN             Acct         ...      NaN      Balance
4      NaN             NaN          ...      NaN      NaN
...    ...
476     NaN   Total for Miscellaneous     ...      NaN      NaN
477     NaN             NaN          ...      NaN      NaN
478     NaN   Not Specified       ...      NaN      NaN
479     NaN             NaN          ...      NaN      0
480     NaN   Total for Not Specified     ...      NaN      NaN
```

```
[481 rows x 20 columns]
```

Working with the DataFrame above doesn't seem straightforward — to make it easier to handle, let's fix its column headers first. You can see in the output above (and in figure 20.1) that the first four rows in the general ledger file describe the table and don't have any actual data. As such, you can tell pandas to skip the first four rows when reading "QuickBooks GL.xlsx" by passing the `skiprows` keyword argument to `read_excel`:

```
In [3]: ledger_df = pd.read_excel('project_data/QuickBooks GL.xlsx', skiprows=4)
```

```
ledger_df
```

```
Out[3]: Unnamed: 0          Acct  Unnamed: 2  ... Credit  Unnamed: 18 Balance
0      NaN           NaN      NaN  ...      NaN      NaN      NaN
1      NaN           Checking  NaN      ...      NaN      NaN      NaN
2      NaN           NaN      NaN  ...      NaN      NaN  5000.0
3      NaN           NaN      NaN  ...      300      NaN  4700.0
4      NaN           NaN      NaN  ...      NaN      NaN  4875.0
...    ...
472     NaN   Total for Miscel...  NaN  ...      NaN      NaN      NaN
473     NaN             NaN      NaN  ...      NaN      NaN      NaN
474     NaN   Not Specified  NaN  ...      NaN      NaN      NaN
475     NaN             NaN      NaN  ...      NaN      NaN      0.0
476     NaN   Total for Not Sp...  NaN  ...      NaN      NaN      NaN
```

```
[477 rows x 20 columns]
```

You now have some useful column headers; however, `ledger_df` still has several unnamed and empty columns ("QuickBooks GL.xlsx" contains empty columns between its data columns). You can drop all columns and rows that contain only `NaN` values by calling the `dropna` method after you read the file:

```
In [4]: ledger_df = pd.read_excel('project_data/QuickBooks GL.xlsx', skiprows=4) 1
      2
      3
      4
      5
ledger_df = ledger_df.dropna(how='all', axis='columns')
ledger_df = ledger_df.dropna(how='all', axis='rows')
ledger_df = ledger_df.rename(columns={'Acct': 'Account', 'SubAcct': 'SubAccount'})
```

Out [4]:

	Account	SubAccount	Date	Debit	Credit	Balance
1	Checking	NaN	NaN	...	NaN	NaN
2		NaN	NaN	08/21/2020	...	5000
3		NaN	NaN	10/02/2020	...	300
4		NaN	NaN	10/10/2020	...	175
5		NaN	NaN	10/24/2020	...	250
..	
471		NaN	NaN	12/28/2020	...	2000
472	Total for Miscel...		NaN		2916	NaN
474	Not Specified		NaN		NaN	NaN
475		NaN	NaN	12/18/2020	...	0
476	Total for Not Sp...		NaN		0	NaN

[444 rows x 10 columns]

The `how='all'` keyword argument used with `dropna` above tells pandas to remove rows or columns only if all of their values are `NaN`.¹ On line 5 above, you also rename two of `ledger_df`'s columns to more explicit names.

The table is much cleaner now, but we're still not done. You need to fill in missing values in the `'Account'` and `'SubAccount'` columns so you can filter the table on those columns later. In the `'Account'` column, you can fill in missing values by running:

```
In [5]: ledger_df['Account'] = ledger_df['Account'].fillna(method='ffill')

ledger_df
```

Out [5]:

	Account	SubAccount	Date	Debit	Credit	Balance
1	Checking	NaN	NaN	...	NaN	NaN
2	Checking	NaN	NaN	08/21/2020	...	5000
3	Checking	NaN	NaN	10/02/2020	...	300
4	Checking	NaN	NaN	10/10/2020	...	175
5	Checking	NaN	NaN	10/24/2020	...	250
..	
471	Miscellaneous	NaN	NaN	12/28/2020	...	2000
472	Total for Miscel...		NaN		2916	NaN
474	Not Specified		NaN		NaN	NaN
475	Not Specified		NaN	12/18/2020	...	0
476	Total for Not Sp...		NaN		0	NaN

[444 rows x 10 columns]

Passing the `method='ffill'` keyword argument² to `fillna` tells pandas to go through the `'Account'` column and, wherever it finds a `NaN`, replace it with the first non-empty value above it.

1: The default value for the `how` keyword argument is `'any'`, which makes `dropna` discard all rows or columns that contain even one `NaN` value.

2: The `'ffill'` value passed to `fillna` stands for *forward-fill*.

Now that you filled in all missing values in the 'Account' column, you can count the number of postings in each account by running:

```
In [6]: ledger_df['Account'].value_counts()
```

```
Out [6]: Landscaping Services           61
          Accounts Receivable (A/R)    49
          Checking                   45
          Accounts Payable (A/P)      26
          Inventory Asset            21
          ...
          Total for Landscaping Services with sub-accounts  1
          Total for Undeposited Funds       1
          Total for Notes Payable        1
          Total for Insurance           1
          Total for Services            1
```

That doesn't seem right — there are many subtotal rows in `ledger_df` skewing the count above. Even more, there are several 'SubAccount' subtotals:

```
In [7]: ledger_df['SubAccount'].value_counts()
```

```
Out [7]: Plants and Soil              2
          Total for Job Materials     2
          Total for Plants and Soil   2
          Total for Maintenance and Repair  2
          ...
          Total for Job Expenses      1
          Original Cost              1
          Total for Fountains and Garden Lighting  1
          Total for Bookkeeper        1
          Telephone                  1
```

Once the data is clean, you can easily compute account subtotals yourself; for now, let's remove subtotal rows from the table. There are several ways to do that with pandas — as you read the following chapters you'll discover most of them. For now, let's define a custom Python function that tells us if an account name contains the word '`Total`':

```
In [8]: def is_subtotal(name):
         return pd.notna(name) and 'Total' in name
```

The function above first checks if `name` is not `NaN` (using pandas's `notna`³ helper function), then checks if `name` contains the word '`Total`'. It returns the combined value of these checks as a boolean (i.e., either `True` if both checks are `True`, or `False` if either of them is `False`). The first check isn't strictly necessary, but it prevents the function from triggering an error if it gets a `NaN` value as its input.

You can call this function with any string value:

```
In [9]: is_subtotal('Accounts Receivable (A/R)')
```

3: Whenever you need to check whether a value is `NaN`, use pandas's `isna` or `notna` helper functions. The `isna` function returns `True` if your value is `NaN`, whereas `notna` returns `True` if your value isn't.

```
Out[9]: False
```

```
In [10]: is_subtotal('Total for Notes Payable')
```

```
Out[10]: True
```

Now let's use a Python list comprehension together with the `is_subtotal` function above to go through the values in the '`Account`' column and keep all account names that don't contain the word '`Total`' in a separate Python list:

```
In [11]: valid_accounts = [name for name in ledger_df['Account'].unique() if not is_subtotal(name)]
```

List comprehensions need some getting used to, but once they click you'll never stop using them — if the list comprehension above is confusing, it is equivalent to the following `for` loop:

```
valid_accounts = []
for name in ledger_df['Account'].unique():
    if not is_subtotal(name):
        valid_accounts.append(name)
```

If you inspect the `valid_accounts` list in a separate code cell, you'll see it contains all unique values in the '`Account`' column that don't have the word '`Total`' in them:

```
In [13]: valid_accounts
```

```
Out[13]: ['Checking',
 'Savings',
 'Accounts Receivable (A/R)',
 ...
 'Utilities',
 'Miscellaneous',
 'Not Specified']
```

The point of creating `valid_accounts` this way is to remove subtotals from `ledger_df` by running the following code:

```
In [14]: ledger_df = ledger_df[ledger_df['Account'].isin(valid_accounts)]
```

```
ledger_df
```

	Account	SubAccount	Date	Debit	Credit	Balance
1	Checking	NaN	NaN	NaN	NaN	NaN
2	Checking	NaN	08/21/2020	5000	NaN	5000.0
3	Checking	NaN	10/02/2020	300	300	4700.0
4	Checking	NaN	10/10/2020	175	NaN	4875.0
5	Checking	NaN	10/24/2020	250	250	4625.0
...
469	Miscellaneous	NaN	11/28/2020	250	NaN	250.0
470	Miscellaneous	NaN	12/20/2020	666	NaN	916.0
471	Miscellaneous	NaN	12/28/2020	2000	NaN	2916.0
474	Not Specified	NaN	NaN	NaN	NaN	NaN
475	Not Specified	NaN	12/18/2020	0	NaN	0.0

```
[411 rows x 10 columns]
```

You'll see a much easier way to get the same result in one line of code in the following chapter, but for now, let's apply the same approach to filtering subtotals from the '`SubAccount`' column. Altogether, the code to find and filter subtotals from both the '`Account`' and '`SubAccount`' columns is:

```
In [15]: def is_subtotal(name):
    return pd.notna(name) and 'Total' in name

valid_accounts = [name for name in ledger_df['Account'].unique()]
if not is_subtotal(name)]
valid_subaccounts = [name for name in ledger_df['SubAccount'].unique()]
if not is_subtotal(name)]

ledger_df = ledger_df[ledger_df['Account'].isin(valid_accounts)]
ledger_df = ledger_df[ledger_df['SubAccount'].isin(valid_subaccounts)]

ledger_df
```

```
Out [15]:
```

	Account	SubAccount	Date	Debit	Credit	Balance
1	Checking	NaN	NaN	...	NaN	NaN
2	Checking	NaN	08/21/2020	...	5000	NaN 5000.0
3	Checking	NaN	10/02/2020	...	NaN	300 4700.0
4	Checking	NaN	10/10/2020	...	175	NaN 4875.0
5	Checking	NaN	10/24/2020	...	NaN	250 4625.0
..
469	Miscellaneous	NaN	11/28/2020	...	250	NaN 250.0
470	Miscellaneous	NaN	12/20/2020	...	666	NaN 916.0
471	Miscellaneous	NaN	12/28/2020	...	2000	NaN 2916.0
474	Not Specified	NaN	NaN	...	NaN	NaN
475	Not Specified	NaN	12/18/2020	...	0	NaN 0.0

[387 rows x 10 columns]

There are still a lot of NaNs in the '`SubAccount`' column. Some accounts in the GL don't have sub-accounts, so this isn't a problem necessarily. However, if you take a look at the '`Utilities`' account, you'll see that some '`SubAccount`' values are indeed missing and need to be filled in:

```
In [16]: ledger_df[ledger_df['Account'] == 'Utilities']
```

```
Out [16]:
```

	Account	SubAccount	Date	Debit	Credit	Balance
457	Utilities	NaN	NaN	...	NaN	NaN
458	Utilities	Gas and Electric	NaN	...	NaN	NaN
459	Utilities	NaN	11/20/2020	...	86.44	NaN 86.44
460	Utilities	NaN	12/19/2020	...	114.09	NaN 200.53
462	Utilities	Telephone	NaN	...	NaN	NaN
463	Utilities	NaN	11/19/2020	...	56.50	NaN 56.50
464	Utilities	NaN	12/19/2020	...	74.36	NaN 130.86

The problem here is that you can't use the `fillna` method like you did for the '`Account`' column earlier. Consider the following

example showing postings in the '`Utilities`' and '`Miscellaneous`' accounts:

```
In [17]: example_df = ledger_df[ledger_df['Account'].isin(['Utilities', 'Miscellaneous'])]

example_df
```

```
Out [17]:   Account      SubAccount      Date    ...   Debit   Credit   Balance
457   Utilities           NaN        NaN  ...     NaN     NaN     NaN
458   Utilities  Gas and Electric       NaN  ...     NaN     NaN     NaN
459   Utilities           NaN  11/20/2020  ...   86.44     NaN   86.44
460   Utilities           NaN  12/19/2020  ...  114.09     NaN  200.53
462   Utilities      Telephone       NaN  ...     NaN     NaN     NaN
463   Utilities           NaN  11/19/2020  ...   56.50     NaN   56.50
464   Utilities           NaN  12/19/2020  ...   74.36     NaN  130.86
468  Miscellaneous          NaN        NaN  ...     NaN     NaN     NaN
469  Miscellaneous          NaN  11/28/2020  ...   250.00     NaN  250.00
470  Miscellaneous          NaN  12/20/2020  ...   666.00     NaN  916.00
471  Miscellaneous          NaN  12/28/2020  ...  2000.00     NaN 2916.00
```

[11 rows x 10 columns]

If you forward-fill missing values in the '`SubAccount`' column with the `fillna` method as before, sub-account values from the '`Utilities`' account will spill into the '`Miscellaneous`' account:

```
In [18]: example_df['SubAccount'] = example_df['SubAccount'].fillna(method='ffill')

example_df
```

```
Out [18]:   Account      SubAccount      Date    ...   Debit   Credit   Balance
457   Utilities           NaN        NaN  ...     NaN     NaN     NaN
458   Utilities  Gas and Electric       NaN  ...     NaN     NaN     NaN
459   Utilities  Gas and Electric  11/20/2020  ...   86.44     NaN   86.44
460   Utilities  Gas and Electric  12/19/2020  ...  114.09     NaN  200.53
462   Utilities      Telephone       NaN  ...     NaN     NaN     NaN
463   Utilities      Telephone  11/19/2020  ...   56.50     NaN   56.50
464   Utilities      Telephone  12/19/2020  ...   74.36     NaN  130.86
468  Miscellaneous      Telephone       NaN  ...     NaN     NaN     NaN
469  Miscellaneous      Telephone  11/28/2020  ...   250.00     NaN  250.00
470  Miscellaneous      Telephone  12/20/2020  ...   666.00     NaN  916.00
471  Miscellaneous      Telephone  12/28/2020  ...  2000.00     NaN 2916.00
```

[11 rows x 10 columns]

Notice the '`SubAccount`' value in the last four rows is '`Telephone`', which isn't a valid miscellaneous sub-account. To fill in missing values in the '`SubAccount`' column, you need to call `fillna` on each account separately. You can do that with a `for` loop and the `loc` slicing operator:

```
In [19]: for account in ledger_df['Account'].unique():
    ledger_df.loc[ledger_df['Account'] == account, 'SubAccount'] = ledger_df.loc[
        ledger_df['Account'] == account, 'SubAccount'].fillna(method='ffill')
```

The code above is a lot to chew on: the `for` loop goes through all unique values in the `'Account'` column and uses each account name) to filter and edit `ledger_df`.⁴ However, it does the trick: if you check the `'Utilities'` and `'Miscellaneous'` accounts, you'll see their sub-accounts filled in without spillover:

In [20]: `ledger_df`

Out [20]:

	Account	SubAccount	Date	...	Debit	Credit	Balance
457	Utilities	NaN	NaN	...	NaN	NaN	NaN
458	Utilities	Gas and Electric	NaN	...	NaN	NaN	NaN
459	Utilities	Gas and Electric	11/20/2020	...	86.44	NaN	86.44
460	Utilities	Gas and Electric	12/19/2020	...	114.09	NaN	200.53
462	Utilities	Telephone	NaN	...	NaN	NaN	NaN
463	Utilities	Telephone	11/19/2020	...	56.50	NaN	56.50
464	Utilities	Telephone	12/19/2020	...	74.36	NaN	130.86
468	Miscellaneous	NaN	NaN	...	NaN	NaN	NaN
469	Miscellaneous	NaN	11/28/2020	...	250	NaN	250.00
470	Miscellaneous	NaN	12/20/2020	...	666	NaN	916.00
471	Miscellaneous	NaN	12/28/2020	...	2000	NaN	2916.00

[11 rows x 10 columns]

4: Head back to “*Filter and Edit*” in chapter 17 for a reminder on how you can use `loc` to modify a `DataFrame` slice.

This type of group operation is common when working with any data. In the following chapters, you'll see how to run the fill-in operation above using pandas's `groupby` method (which is much easier to understand than the mouthful code above).

One last step in cleaning the general ledger is removing rows with non-empty values only in their `'Account'` or `'SubAccount'` columns (e.g., the first two rows above). You can do that by calling `dropna` and specifying the subset of columns you want pandas to check for empty values:

In [21]: `ledger_df = ledger_df.dropna(subset=ledger_df.columns[2:], how='all')`

Notice that instead of typing column names for the `subset` keyword argument, you can use a slice of `ledger_df`'s `columns` attribute, which already contains all column names in order (the first two names being `'Account'` and `'SubAccount'` that we want to ignore in this case).

After all this scrubbing, what can you do with `ledger_df`? For instance, you can easily slice it to get postings in various accounts or sub-accounts:

In [22]:

```
ledger_df[
    (ledger_df['Account'] == 'Landscaping Services') &
    (ledger_df['SubAccount'] == 'Fountains and Garden Lighting')
]
```

Out [22]:

	Account	SubAccount	Date	...	Debit	Credit	Balance
290	Landscaping Serv...	Fountains and Ga...	10/20/2020	...	NaN	275	275.0
291	Landscaping Serv...	Fountains and Ga...	10/27/2020	...	NaN	48	323.0
292	Landscaping Serv...	Fountains and Ga...	11/17/2020	...	NaN	275	598.0
293	Landscaping Serv...	Fountains and Ga...	11/17/2020	...	NaN	72	670.0
294	Landscaping Serv...	Fountains and Ga...	11/17/2020	...	NaN	75	745.0
295	Landscaping Serv...	Fountains and Ga...	12/13/2020	...	NaN	275	1020.0
296	Landscaping Serv...	Fountains and Ga...	12/21/2020	...	NaN	275	1295.0
297	Landscaping Serv...	Fountains and Ga...	01/01/2021	...	NaN	84	1379.0
298	Landscaping Serv...	Fountains and Ga...	01/02/2021	...	NaN	180	1559.0
299	Landscaping Serv...	Fountains and Ga...	01/02/2021	...	NaN	275	1834.0
300	Landscaping Serv...	Fountains and Ga...	01/03/2021	...	NaN	45	1879.0
301	Landscaping Serv...	Fountains and Ga...	01/03/2021	...	NaN	45	1924.0
302	Landscaping Serv...	Fountains and Ga...	01/03/2021	...	NaN	275	2199.0
303	Landscaping Serv...	Fountains and Ga...	01/04/2021	...	NaN	47.5	2246.5

[14 rows x 10 columns]

Or re-sort the table by any of its columns:

In [23]: `ledger_df.sort_values('Credit', ascending=False)`

Out [23]:

	Account	SubAccount	Date	...	Debit	Credit	Balance
234	Notes Payable	NaN	01/01/2021	...	NaN	25000	25000.00
239	Opening Balance ...	NaN	12/19/2020	...	NaN	13495	18495.00
238	Opening Balance ...	NaN	08/21/2020	...	NaN	5000	5000.00
230	Loan Payable	NaN	01/01/2021	...	NaN	4000	4000.00
169	Accounts Payable...	NaN	12/28/2020	...	NaN	2000	4213.62
..
464	Utilities	Telephone	12/19/2020	...	74.36	NaN	130.86
469	Miscellaneous	NaN	11/28/2020	...	250	NaN	250.00
470	Miscellaneous	NaN	12/20/2020	...	666	NaN	916.00
471	Miscellaneous	NaN	12/28/2020	...	2000	NaN	2916.00
475	Not Specified	NaN	12/18/2020	...	0	NaN	0.00

[335 rows x 10 columns]

Or compute summary values for any account in the table:

In [24]: `ledger_df[ledger_df['Account'] == 'Utilities']['Debit'].sum()`

Out [24]: 331.39

You can also save the scrubbed GL back to an Excel file that is much easier to work with by running:

In [25]: `ledger_df.to_excel('project_data/Clean QuickBooks GL.xlsx', index=False)`

The entire code needed to read and clean “QuickBooks GL.xlsx” is listed below. It might seem like a lot of code, but if you need to reformat a GL export manually every other week in Excel, code like this one can save you considerable time. As we go through more of pandas’s features in the following chapters, you’ll see how to simplify the code further and make it easier to work with.

```
In [26]: ledger_df = pd.read_excel('project_data/QuickBooks GL.xlsx', skiprows=4)          1
       ledger_df = ledger_df.dropna(how='all', axis='columns')                           2
       ledger_df = ledger_df.dropna(how='all', axis='rows')                             3
       ledger_df = ledger_df.rename(columns={'Acct': 'Account', 'SubAcct': 'SubAccount'}) 4
       ledger_df['Account'] = ledger_df['Account'].fillna(method='ffill')                5
                                         6
def is_subtotal(name):                                                               7
    return pd.notna(name) and 'Total' in name                                     8
                                         9
valid_accounts = [name for name in ledger_df['Account'].unique()]                  10
if not is_subtotal(name):
    valid_subaccounts = [name for name in ledger_df['SubAccount'].unique()]           11
    if not is_subtotal(name):
                                         12
        ledger_df = ledger_df[ledger_df['Account'].isin(valid_accounts)]            13
        ledger_df = ledger_df[ledger_df['SubAccount'].isin(valid_subaccounts)]         14
                                         15
for account in ledger_df['Account'].unique():
    ledger_df.loc[ledger_df['Account'] == account, 'SubAccount'] = ledger_df.loc[
        ledger_df['Account'] == account, 'SubAccount'].fillna(method='ffill')           16
                                         17
                                         18
ledger_df = ledger_df.dropna(subset=ledger_df.columns[2:], how='all')                 19
                                         20
                                         21
                                         22
                                         23
```

Summary

This quick project chapter showed you how to use pandas to read and clean a general ledger file. In the following chapters, you'll uncover more of pandas's tools that not only let you clean and reshape tables faster and with less code but also enable you to extract insights from your data. One such set of tools are pandas's text handling methods, which we look at next.

Working with text columns

A large part of working with data revolves around handling text — what Python calls strings. Python has many built-in tools¹ for manipulating strings, many of which have been adapted by `pandas` to work on entire columns of string values. Before we look at how you can use these tools with the data in `ledger_df`, let's quickly revisit Python strings.

1: Python's flexible tools for manipulating strings are a big contributor to its popularity.

Revisiting Python strings

You may remember from chapter 5 that you can create a new string variable in Python by assigning a sequence of (one or more) characters wrapped in quotes² to a name:

```
In [1]: message = 'hello, python for accounting'
```

The `message` variable created above is a Python string object. There are several methods available on string objects, most of which *return* a new string object. For instance, to get a new uppercase string that has the same characters as `message`, you can run:

```
In [2]: message.upper()
```

```
Out [2]: 'HELLO, PYTHON FOR ACCOUNTING'
```

Calling a string method does not modify the original string in place (i.e., strings in Python are immutable). If you want to update `message` and make it point to the uppercase text above, you need to assign the output of `upper` back to `message`:

```
In [3]: message = message.upper()
```

```
message
```

```
Out [3]: 'HELLO, PYTHON FOR ACCOUNTING'
```

Something I didn't mention in chapter 5 is that, because Python strings are sequences of characters, they can be sliced, just like you slice a regular Python list. For instance, if you want to select the first five characters in `message`, you can use:

```
In [4]: message[:5]
```

```
Out [4]: 'HELLO'
```

Another feature that both Python lists and strings have in common is that you can use the `in` keyword to check for membership of an item or a substring. For instance, you can check whether a piece of text is part of `message` with:

```
In [5]: 'PYTHON' in message
```

```
Out [5]: True
```

```
In [6]: 'pandas' in message
```

```
Out [6]: False
```

You can combine multiple strings together using the `+` operator:

```
In [7]: 'hello' + ', ' + 'python ' + 'for ' + 'accounting'
```

```
Out [7]: 'hello, python for accounting'
```

Or if you have several variables that you want to put into a single piece of text, you can use a Python f-string:

```
In [8]: first_name = "margaret"  
last_name = "hamilton"  
age = 83  
  
f'{first_name} {last_name} is {age} years old!'
```

```
Out [8]: margaret hamilton is 83 years old!
```

All of the operations above can be applied to string values that are part of `DataFrame` or `Series` objects, but with a slight `pandas` twist. Let's take a closer look.

String methods in pandas

A significant part of working with data involves handling text, which is why `pandas` equipped `Series` objects with several methods for manipulating string values. However, unlike the `Series` methods you've seen so far, you can only access these string methods by putting the term `str` in front of them — for example, to make all text values in `ledger_df`'s `'Channel'` column uppercase:

```
In [9]: ledger_df['Channel'].str.upper()
```

```
Out[9]: 0      SHOPPE.COM
        1      WALCART
        2      BULLSEYE
        3      BULLSEYE
        4      BULLSEYE
       ...
14049      BULLSEYE
14050      BULLSEYE
14051  UNDERSTOCK.COM
14052      IBAY.COM
14053  UNDERSTOCK.COM
Name: Channel, Length: 14054, dtype: object
```

If you try to call the `upper` method directly, without typing `str` in front of it, you'll get the following error message:

```
In [10]: ledger_df['Channel'].upper()
```

```
Out[10]: -----
AttributeError                               Traceback (most recent call last)
<ipython-input-8-5ec418284f6c> in <module>
      ...
-> 1 ledger_df['Channel'].upper()
...
AttributeError: 'Series' object has no attribute 'upper'
```

The `str` keyword is a `Series` attribute (i.e., as opposed to a `Series` method). It is a trick `pandas` uses to organize all string methods under a common name, so it's easier for you to find and use them.

You can call string methods on any column that contains text values — you will get an error if you call them on columns that have other types of data (e.g., numbers or mixed values). Besides the `upper` method above, there are many other string methods available; table 21.1 lists some of them, together with code examples. Most of these `Series` methods have the same name as Python's built-in string methods (e.g., `upper`, `lower`, `title`); if you discover an interesting method on a Python string, a `Series` equivalent that works on an entire column of strings is likely available.

Table 21.1: Some of the methods available on `Series` objects containing text values. These and more methods are described in the official `pandas` documentation available at pandas.pydata.org/pandas-docs/stable/user_guide/text.html.

Method	Example	Description
<code>contains</code>	<code>df['Product Name'].str.contains('shoe')</code>	Returns a boolean <code>Series</code> indicating whether a given string occurs in each value of the <code>Series</code> .
	<code>(df['Product Name'].str.contains('shoe', case=False))</code>	Same as above, but ignores string case.
<code>count</code>	<code>df['Product Name'].str.count('in')</code>	Returns a numerical <code>Series</code> with the number of times a given string (e.g., in this case ' <code>in</code> ') occurs in each value of the <code>Series</code> .
<code>startswith</code>	<code>df['Product Name'].str.startswith("Men's")</code>	Returns a boolean <code>Series</code> indicating whether each value of the <code>Series</code> starts with a given string.

Table 21.1: Some of the methods available on Series objects containing text values. These and more methods are described in the official pandas documentation available at pandas.pydata.org/pandas-docs/stable/user_guide/text.html.

Method	Example	Description
endswith	<code>df['Product Name'].str.endswith("gray")</code>	Returns a boolean Series indicating whether each value of a Series ends with a given string.
lower	<code>df['Product Name'].str.lower()</code>	Converts strings in the Series to lowercase.
upper	<code>df['Product Name'].str.upper()</code>	Converts strings in the Series to uppercase.
title	<code>df['Product Name'].str.title()</code>	Converts strings in a Series to title case (i.e., each word in the string starts with a capital letter).
strip	<code>df['Product Name'].str.strip()</code> <code>df['Product Name'].str.strip(to_strip='x')</code>	Removes leading and trailing characters in each value of a Series. By default, removes whitespace characters (i.e., spaces, tabs or newline characters). Same as above, but specifies which trailing character to remove (e.g., 'xShirt (x-small)x' becomes 'Shirt (x-small)').
replace	<code>(df['Product Name'].str.replace("w/", "with"))</code> <code>(df['Product Name'].str.replace("w/", "with", case=False))</code> <code>(df['Product Name'].str.replace("w/", "with", n=1))</code>	Replaces all occurrences of a pattern with another string in each value of the Series. Same as above, but ignores case when looking for characters to replace. Same as above, but specifies how many replacements to make. Here, only the first (i.e., left-most) occurrence of 'w/' in a string is replaced.
slice	<code>df['Product Name'].str.slice(10, 20)</code> <code>(df['Product Name'].str.slice(10, 20, step=2))</code>	Returns a slice from each string in the Series. Start and stop positions are specified as arguments. If the string is shorter than start characters, the returned value is an empty string. Same as above, but returns every second character.
split	<code>df['Product Name'].str.split()</code> <code>df['Product Name'].str.split(pat="x")</code>	Splits each string in the Series around a delimiter into a list of strings. By default, the delimiter is any (one or more) whitespace character. Same as above, but uses the x character as a delimiter.
isalpha	<code>df['Product Name'].str.isalpha()</code>	Returns a boolean Series indicating whether each value contains <i>only</i> alphabetic symbols (i.e., 'abc' will return <code>True</code> , whereas 'a.b.c.' or 'a2' will return <code>False</code>).
isnumeric	<code>df['Product Name'].str.isnumeric()</code>	Returns a boolean Series indicating whether each value contains <i>only</i> numeric symbols (i.e., '1' or '1/2' will return <code>True</code>).
isupper	<code>df['Product Name'].str.isupper()</code>	Returns a boolean Series indicating whether each value contains <i>only</i> uppercase letters (i.e., 'ABC' will return <code>True</code> , whereas 'AbC' will return <code>False</code>).

Table 21.1: Some of the methods available on Series objects containing text values. These and more methods are described in the official pandas documentation available at pandas.pydata.org/pandas-docs/stable/user_guide/text.html.

Method	Example	Description
islower	<code>df['Product Name'].str.isupper()</code>	Returns a boolean Series indicating whether each value contains <i>only</i> lowercase letters (i.e., 'abc' will return <code>True</code> , whereas 'aBc' will return <code>False</code>).

It's worth pointing out that these string methods skip empty values (i.e., NaNs or `<NA>`). For instance, to make all values in the 'Product Name' column uppercase, you can use the `upper` method as before — but notice in the output of `upper` that missing values remain unchanged (i.e., NaNs remain NaNs):

```
In [11]: ledger_df['Product Name']
```

```
Out [11]: 0      Cannon Water Bomb Balloons 100 Pack
           1      LEGO Ninja Turtles Stealth Shell in Purs...
           2                  NaN
           3      Transformers Age of Extinction Generatio...
           4      Transformers Age of Extinction Generatio...
           ...
          14049     AC Adapter/Power Supply&Cord for Lenovo ...
          14050     Cisco Systems Gigabit VPN Router (RV320K...
          14051     Philips AJ3116M/37 Digital Tuning Clock ...
          14052                  NaN
          14053     Sirius Satellite Radio XADH2 Home Access...
Name: Product Name, Length: 14054, dtype: object
```

```
In [12]: ledger_df['Product Name'].str.upper()
```

```
Out [12]: 0      CANNON WATER BOMB BALLOONS 100 PACK
           1      LEGO NINJA TURTLES STEALTH SHELL IN PURS...
           2                  NaN
           3      TRANSFORMERS AGE OF EXTINCTION GENERATIO...
           4      TRANSFORMERS AGE OF EXTINCTION GENERATIO...
           ...
          14049     AC ADAPTER/POWER SUPPLY&CORD FOR LENOVO ...
          14050     CISCO SYSTEMS GIGABIT VPN ROUTER (RV320K...
          14051     PHILIPS AJ3116M/37 DIGITAL TUNING CLOCK ...
          14052                  NaN
          14053     SIRIUS SATELLITE RADIO XADH2 HOME ACCESS...
Name: Product Name, Length: 14054, dtype: object
```

The same happens with string methods that return a boolean Series. For instance, if you want to check whether values in the 'Product Name' column contain the term 'LEGO', you can run:

```
In [13]: ledger_df['Product Name'].str.contains('LEGO')
```

```
Out [13]: 0      False
           1      True
           2      NaN
           3      False
           4      False
```

```

...
14049    False
14050    False
14051    False
14052     NaN
14053    False
Name: Product Name, Length: 14054, dtype: object

```

At times you might need to use `contains` to filter a `DataFrame`. For instance, to filter `ledger_df` and keep all rows where product names contain the term '`LEGO`', you might run something similar to the following code:

```
In [14]: ledger_df[ledger_df['Product Name'].str.contains('LEGO')]
```

```
Out [14]: -----
ValueError                                                 Traceback (most recent call last)
<ipython-input-26-0940b377f685> in <module>
      1 ledger_df[ledger_df['Product Name'].str.contains('LEGO')]
      2 ...
ValueError: Cannot mask with non-boolean array containing NA / NaN values
```

However, because the '`Product Name`' column has missing values, the output of `contains` also includes missing values — as I mentioned earlier, `NANs` in the input remain `NANs` in the output of pandas's string methods. You need to fill in these missing values in the output of `contains` (or in any other boolean `Series`) before you can use it for filtering:

```
In [15]: ledger_df[ledger_df['Product Name'].str.contains('LEGO').fillna(False)]
```

```
Out [15]:   InvoiceNo      Channel      Product Name ... Unit Price Quantity  Total
1          1533      Walcart      LEGO Ninja Turtles S... ...  6.70      1   6.70
43         1575      iBay.com      LEGO Star Wars Clone... ... 14.68      2  29.36
105        1637      Bullseye      LEGO LOTR 79006 The ... ...  7.67      6  46.02
176         1708      Shoppe.com      LEGO City Fire Chief... ... 24.95      1  24.95
228         1608      iBay.com      LEGO Star Wars Clone... ... 14.68      2  29.36
...
13525       15007  Understock.com      LEGO City Trains Hig... ...  7.41     18 133.38
13550       15082      Walcart      LEGO City Fire Chief... ... 25.11      8 200.88
13731       15131      Walcart      LEGO City Fire Chief... ... 25.11      8 200.88
13753       15285  Understock.com      LEGO Star Wars Clone... ... 14.84      2  29.68
14031       15468  Understock.com      LEGO Star Wars Clone... ... 14.84      2  29.68
```

[200 rows x 12 columns]

Notice you can chain the `fillna` method to the output of `contains` — in general, you can chain as many `Series` methods you need. In this case, I filled missing values in the boolean `Series` with `False`, because I wanted to discard rows with missing product names, but you can fill them in with `True` if you want to keep them.

Two of the string methods in table 21.1 are worth discussing in more detail: `replace` and `split`. The `replace` string method shares a name with another `Series` method, which might cause some

confusion, and the `split` method can be used to create multiple columns at once, which needs some explanations.

Replacing parts of text

There are two `replace` methods available on `Series` objects: one that replaces values entirely and a string `replace` method that substitutes a pattern in each value with a replacement. The first method works for any type of data, whereas the string `replace` method works only on `Series` that contain string values.

Whenever you need to replace a character (or multiple characters) *within* each value in a text column, you should use the string `replace` (i.e., type `str` in front of the method when calling it). For example, some of the values in the '`Channel`' column end in '`.com`'. To remove this ending, you can replace it with an empty string:

```
In [16]: ledger_df['Channel'].str.replace('.com', '')
```

```
Out [16]: 0      Shoppe
1      Walcart
2      Bullseye
3      Bullseye
4      Bullseye
...
14049    Bullseye
14050    Bullseye
14051    Understock
14052      iBay
14053    Understock
Name: Channel, Length: 14054, dtype: object
```

This replaces part of each value (i.e., a *substring*) in the '`Channel`' column with another string (in this case, with an empty string). Values in the '`Channel`' column that don't contain the '`.com`' substring are ignored and remain unchanged.

However, when you need to replace *entire values* in a text column, not just a substring in each, you should use the other `replace` method, without `str` in front of it. For example, to replace '`iBay.com`' with '`Amazon.com`', you can use:

```
In [17]: ledger_df['Channel'].replace('iBay.com', 'Amazon.com')
```

```
Out [17]: 0      Shoppe.com
1      Walcart
2      Bullseye
3      Bullseye
4      Bullseye
...
14049    Bullseye
14050    Bullseye
14051    Understock.com
```

```
14052      Amazon.com
14053      Understock.com
Name: Channel, Length: 14054, dtype: object
```

In some cases, you can get the same result with either method — but in those cases where you can't, knowing about the two `replace` methods will save you some headache.

Splitting text values into multiple columns

The `split` method is also worth highlighting because it returns a list of values — if you call it on a column with string values, it will return a `Series` of Python lists. Yes, a column of lists.

In `ledger_df`, values in the '`ProductID`' column contain a forward slash that separates two parts of the product identifier: a category identifier (i.e., the first part before the slash) and a unique item identifier (i.e., the part after).

```
In [18]: ledger_df[['Product Name', 'ProductID', 'Unit Price', 'Quantity', 'Total']]
```

```
Out [18]:
```

	Product Name	ProductID	Unit Price	Quantity	Total
0	Cannon Water Bomb Ba...	T&G/CAN-97509	20.11	14	281.54
1	LEGO Ninja Turtles S...	T&G/LEG-37777	6.70	1	6.70
2		NaN	11.67	5	58.35
3	Transformers Age of ...	T&G/TRA-20170	13.46	6	80.76
4	Transformers Age of ...	T&G/TRA-20170	13.46	6	80.76
...
14049	AC Adapter/Power Sup...	E/AC-63975	28.72	8	229.76
14050	Cisco Systems Gigabi...	E/CIS-74992	33.39	1	33.39
14051	Philips AJ3116M/37 D...	E/PHI-08100	4.18	1	4.18
14052		NaN	4.78	25	119.50
14053	Sirius Satellite Rad...	E/SIR-83381	33.16	2	66.32

[14054 rows x 5 columns]

To split product ID values on the forward slash character, run:

```
In [19]: ledger_df['ProductID'].str.split('/')
```

```
Out [19]:
```

0	[T&G, CAN-97509]
1	[T&G, LEG-37777]
2	[T&G, PET-14209]
3	[T&G, TRA-20170]
4	[T&G, TRA-20170]
...	...
14049	[E, AC-63975]
14050	[E, CIS-74992]
14051	[E, PHI-08100]
14052	[E, POL-61164]
14053	[E, SIR-83381]

Name: ProductID, Length: 14054, dtype: object

The output above is a `Series` of Python lists. Working with Python lists in a `Series` can get confusing. Ideally, each item in the lists returned by `split` should be a separate column in a new `DataFrame`, so that you can work with the split values using `DataFrame` methods. Luckily, you can get a `DataFrame` as the output of `split` by passing the `expand=True` keyword argument:

```
In [20]: ledger_df['ProductID'].str.split('/', expand=True)
```

```
Out [20]:      0          1
0    T&G  CAN-97509
1    T&G  LEG-37777
2    T&G  PET-14209
3    T&G  TRA-20170
4    T&G  TRA-20170
...
24019     E  AC-63975
24020     E  CIS-74992
24021     E  PHI-08100
24022     E  POL-61164
24023     E  SIR-83381
```

[24024 rows x 2 columns]

The code above returns a `DataFrame` with one column for each item in the list of splits (had the `'ProductID'` values contained multiple forward slash characters, the number of splits would have been greater and the `DataFrame` returned by `split` would have had more than two columns).

By default, the columns returned by `split` have numbers as their names (i.e., `0` and `1` above). You can assign the output of `split` to a new `DataFrame` variable to continue working with these split values (and you might want to rename its columns using the `rename` method). You can also assign the output of `split` back to `ledger_df` as two new columns:

```
In [21]: ledger_df[['CategoryID', 'ItemID']] = ledger_df['ProductID'].str.split('/', expand=True)

ledger_df
```

```
Out [21]:   InvoiceNo      Channel      Product Name ...   Total CategoryID      ItemID
0        1532  Shoppe.com  Cannon Water Bom... ...  281.54    T&G  CAN-97509
1        1533       Walcart  LEGO Ninja Turtl... ...   6.70    T&G  LEG-37777
2        1534      Bullseye           NaN ...   58.35    T&G  PET-14209
3        1535      Bullseye  Transformers Age... ...  80.76    T&G  TRA-20170
4        1535      Bullseye  Transformers Age... ...  80.76    T&G  TRA-20170
...
14049     15581      Bullseye  AC Adapter/Power... ... 229.76     E  AC-63975
14050     15582      Bullseye  Cisco Systems Gi... ... 33.39     E  CIS-74992
14051     15583  Understock.com  Philips AJ3116M/... ...   4.18     E  PHI-08100
14052     15584       iBay.com           NaN ... 119.50     E  POL-61164
14053     15585  Understock.com  Sirius Satellite... ...  66.32     E  SIR-83381
```

```
[14054 rows x 14 columns]
```

The two extra columns on the right of `ledger_df` (i.e., '`CategoryID`' and '`ItemID`') are the output of splitting '`ProductID`' values around the forward-slash character.

Concatenating text columns

Instead of splitting text values, you might want to create a new text column in a `DataFrame` by concatenating values from its other columns. Concatenating text columns is just as easy as putting two strings together with the `+` operator. Just like you can add a single number to an entire column of numbers, you can add a piece of text to all the values in a text column:

```
In [22]: 'Category ID is: ' + ledger_df['CategoryID']
```

```
Out [22]: 0      Category ID is: T&G
1      Category ID is: T&G
2      Category ID is: T&G
3      Category ID is: T&G
4      Category ID is: T&G
...
14049     Category ID is: E
14050     Category ID is: E
14051     Category ID is: E
14052     Category ID is: E
14053     Category ID is: E
Name: CategoryID, Length: 14054, dtype: object
```

You can concatenate text columns using the same operator. For instance, you can re-create product IDs from the new '`CategoryID`' and '`ItemID`' columns using:

```
In [23]: ledger_df['CategoryID'] + '/' + ledger_df['ItemID']
```

```
Out [23]: 0      T&G/CAN-97509
1      T&G/LEG-37777
2      T&G/PET-14209
3      T&G/TRA-20170
4      T&G/TRA-20170
...
14049     E/AC-63975
14050     E/CIS-74992
14051     E/PHI-08100
14052     E/POL-61164
14053     E/SIR-83381
Length: 14054, dtype: object
```

At times, you will need to create text columns from numerical values in a `DataFrame`. For instance, you might want to combine values from the '`InvoiceNo`' and '`AccountNo`' columns together, separated by a forward slash, to create a transaction ID (i.e., just

like you combined '`CategoryID`' and '`ItemID`' values above, to get a product ID). However, if you try to combine '`InvoiceNo`' and '`AccountNo`' as we did earlier, you will get an error:

```
In [24]: ledger_df['InvoiceNo'] + '/' + ledger_df['AccountNo']
```

```
Out [24]:
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-56-57c8a5186f50> in <module>
     1 ledger_df['InvoiceNo'] + '/' + ledger_df['AccountNo']
     ...
ValueError: unknown type str32
```

This happens because the '`InvoiceNo`' and '`AccountNo`' columns contain numbers, not strings. While the error message above isn't very informative, it is pandas's way of telling you that it can't combine integers and strings (i.e., numbers and the forward-slash character). To get the result you want, you need to first use the `astype` method and change column types to `string` before concatenating columns:

```
In [25]: ledger_df['InvoiceNo'].astype('string') + '/' + ledger_df['AccountNo'].astype('string')
```

```
Out [25]:
```

0	1532/5004
1	1533/5004
2	1534/5004
3	1535/5004
4	1535/5004
	...
14049	15581/5004
14050	15582/5004
14051	15583/5004
14052	15584/5004
14053	15585/5004

Length: 14054, dtype: string

The output above is a `Series` of string values you can assign to a new column in `ledger_df` if you want to and manipulate using the same string methods we've been using so far.

String data types in pandas

In the previous chapters, I mentioned that when you read data from an Excel spreadsheet with `read_excel`, pandas tries to figure out what type of data is in each column and assigns its best guess as `DataFrame` column types. Even though pandas has a `string` data type in its toolbox, it assigns³ the `object` data type to any column that contains string values:

```
In [26]: ledger_df = pd.read_excel('Q1Sales.xlsx')
```

```
ledger_df.info()
```

3: For the examples in this book, I'm using pandas version 1.0. Future versions of pandas will likely start using the `string` data type.

```
Out [26]: <class 'pandas.core.frame.DataFrame'>
RangeIndex: 14054 entries, 0 to 14053
Data columns (total 12 columns):
 #   Column      Non-Null Count Dtype  
--- 
 0   InvoiceNo   14054 non-null  int64  
 1   Channel     14054 non-null  object  
 2   Product Name 12362 non-null  object  
 3   ProductID   14054 non-null  object  
 4   Account     14054 non-null  object  
 5   AccountNo   14054 non-null  int64  
 6   Date        14054 non-null  datetime64[ns]
 7   Deadline    14054 non-null  object  
 8   Currency    14054 non-null  object  
 9   Unit Price  14054 non-null  float64 
 10  Quantity    14054 non-null  int64  
 11  Total       14054 non-null  float64 
dtypes: datetime64[ns](1), float64(2), int64(3), object(6)
memory usage: 1.3+ MB
```

Notice that all string columns (i.e., 'Channel', 'Product Name', 'ProductID', 'Account', and 'Currency') have the `object` Dtype. You can easily convert these columns to the `string` data type by adding a `convert_dtypes` call right after you read the file:

```
In [27]: ledger_df = pd.read_excel('Q1Sales.xlsx').convert_dtypes()

ledger_df.info()
```

```
Out [27]: <class 'pandas.core.frame.DataFrame'>
RangeIndex: 14054 entries, 0 to 14053
Data columns (total 12 columns):
 #   Column      Non-Null Count Dtype  
--- 
 0   InvoiceNo   14054 non-null  Int64  
 1   Channel     14054 non-null  string  
 2   Product Name 12362 non-null  string  
 3   ProductID   14054 non-null  string  
 4   Account     14054 non-null  string  
 5   AccountNo   14054 non-null  Int64  
 6   Date        14054 non-null  datetime64[ns]
 7   Deadline    14054 non-null  string  
 8   Currency    14054 non-null  string  
 9   Unit Price  14054 non-null  float64 
 10  Quantity    14054 non-null  Int64  
 11  Total       14054 non-null  float64 
dtypes: Int64(3), datetime64[ns](1), float64(2), string(6)
memory usage: 1.3 MB
```

Or by calling the `astype` method on any string column you want to convert explicitly:

```
In [28]: ledger_df['Product Name'].astype('string')
```

```
Out [28]: 0      Cannon Water Bomb Ba...
           1      LEGO Ninja Turtles S...
           2              <NA>
           3      Transformers Age of ...
           4      Transformers Age of ...
           ...
          14049     AC Adapter/Power Sup...
          14050     Cisco Systems Gigabi...
          14051     Philips AJ3116M/37 D...
          14052             <NA>
          14053     Sirius Satellite Rad...
Name: Product Name, Length: 14054, dtype: string
```

For all practical uses, columns with the `object` or `string` data type work the same way. However, keep in mind that columns with different types of values (e.g., strings, numbers, or booleans in the same Series) also get assigned the `object` data type. For instance, the following Series has the `object` data type:

```
In [29]: pd.Series([1011, '$1320', "$980", 645, 340])
```

```
Out [29]: 0    1011
           1    $1320
           2    $980
           3    645
           4    340
dtype: object
pd.Series([1011, '1320', "980", 645, 340])
```

Even though the Series above contains a mix of numbers and strings, the `str` methods we went over still work, but only on its values that are actual strings:

```
In [30]: pd.Series([1011, '$1320', "$980", 645, 340]).str.strip('$')
```

```
Out [30]: 0    NaN
           1    1320
           2    980
           3    NaN
           4    NaN
           5    NaN
dtype: object
```

The code above tries to strip the dollar sign in front of the two values that have it. However, because there is no `strip` method available on integers, the numbers in the Series turned into `NaN` values. If you want to keep them in the output, you first have to convert all values in the Series to the `string` data type:

```
In [31]: pd.Series([1011, '$1320', "$980", 645, 340]).astype('string').str.strip('$')
```

```
Out[31]: 0    1011
         1    1320
         2     980
         3     645
         4     340
dtype: string
```

Converting `DataFrame` text columns to the `string` data type prevents you from accidentally introducing `Nan`s into your data when using string methods. Columns with the `string` data type also tend to use less memory on your computer than `object` columns, so if you chain `convert_dtypes` after `read_excel`, your `DataFrame` variables will be slimmer and faster.

Overthinking: Regular expressions

Regular expressions are a powerful way to find and extract complex patterns from text values. In Python, a regular expression is just a string that contains specific symbols and operators which can be used to define patterns. For example, the following regular expression would find any text value that includes the terms '`Nikon`' and '`Camera`':

```
In [32]: pattern = '(Nikon) .*(Camera)'
```

All of the text values below would match this regular expression:

```
Nikon Camera
Nikon COOLPIX L26 16.1 MP Digital Camera with 5x Zoom
D3100 14.2MP Nikon Camera Digital SLR with 18-55mm f/3.5-5.6 VR
```

Regular expressions have their own symbols and operators that are different from Python's or `pandas`'s — you can think of regular expressions as an entirely different, highly specialized programming language that is embedded in Python. As such, this section can only be a concise introduction to this new language; you'll have to read more about regular expressions on your own if you want to add them to your toolkit.⁴

However, keep in mind that for most use cases (and with some creativity) you will be able to get the results you need with `pandas`'s string methods without using regular expressions at all. The regular expression above can be reproduced by combining multiple uses of the `contains` string method. Nevertheless, regular expressions are available in Python and `pandas`, and you can use them to find and extract complicated patterns from text whenever you need to.

For the code examples in this section, let's create a separate `DataFrame` with all camera sales in `ledger_df`. This new `DataFrame`

"Some people, when confronted with a problem, think «I know, I'll use regular expressions.» Now they have two problems."
– Jamie Zawinski

⁴: Python's official documentation includes a tutorial on regular expressions that is an excellent place to start learning more about them. It's available at docs.python.org/3/howto/regex.html#regex-howto.

will make it easier to illustrate how regular expressions simplify certain string operations in pandas.

```
In [33]: is_camera = ledger_df['Product Name'].str.contains('camera', case=False).fillna(False)

cameras_df = ledger_df[is_camera]
cameras_df = cameras_df[['ProductID', 'Product Name', 'Total']]

cameras_df
```

```
Out [33]:      ProductID          Product Name   Total
66    C&P/KID-94587    Kidz Digital Camera  275.64
117   C&P/KOD-01305  Kodak ZM1-NM 1 MP 1-Inch LCD CMOS Se...  64.10
154   C&P/FOS-95687  Foscam FI8910W White Wireless IP Cam...  669.20
265   C&P/Q-S-31839  Q-See QSC414D Outdoor Dome Color CCD...  7.82
287   C&P/KOD-01305  Kodak ZM1-NM 1 MP 1-Inch LCD CMOS Se...  64.10
...
13916  C&P/NEE-31972  NEEWER 10x25 Zoom LCD Binoculars Bu...  37.68
13950  T&G/LIG-86589  Lights, Camera, Action Decor        23.02
13954  C&P/DAH-04621  Dahua IPC-HFW2100 S 1.3MP Weatherpro...  126.00
13986  C&P/NIK-92147  Nikon D3100 14.2MP Digital SLR Camer...  54.10
13996  C&P/SEC-57209  Securityman Wi-Fi Interference Free ...  17.44

[443 rows x 3 columns]
```

The `cameras_df` DataFrame includes all rows in `ledger_df` where values in the `'Product Name'` column contain the term `'camera'`.

If you want to filter this DataFrame further, based on values in the `'Product Name'` column, to keep only sales of four different brands of cameras (i.e., *Nikon*, *Canon*, *Kodak*), you can use a regular expression (i.e., a pattern) with the `contains` method:

```
In [34]: pattern = '(Nikon|Canon|Kodak)'
cameras_df = cameras_df[cameras_df['Product Name'].str.contains(pattern)]

cameras_df
```

```
Out [34]:      ProductID          Product Name   Total
117   C&P/KOD-01305  Kodak ZM1-NM 1 MP 1-Inch LCD CMOS Se...  64.10
287   C&P/KOD-01305  Kodak ZM1-NM 1 MP 1-Inch LCD CMOS Se...  64.10
616   C&P/KOD-32137  Kodak EasyShare Z990 12 MP Digital C...  166.30
2151  C&P/CAN-50721  Canon PowerShot SX50 HS 12MP Digital...  6.45
2280  C&P/CAN-50721  Canon PowerShot SX50 HS 12MP Digital...  6.45
...
12459  C&P/CAN-12514  Canon EOS Rebel T2i DSLR Camera (Bod...  35.40
12522  C&P/CAN-12514  Canon EOS Rebel T2i DSLR Camera (Bod...  35.40
12905  C&P/KOD-32137  Kodak EasyShare Z990 12 MP Digital C...  332.60
13594  C&P/NIK-92147  Nikon D3100 14.2MP Digital SLR Camer...  54.10
13986  C&P/NIK-92147  Nikon D3100 14.2MP Digital SLR Camer...  54.10

[77 rows x 3 columns]
```

Notice the `pattern` variable on line 1 in the example above. It is just a Python string — but it contains a sequence of terms,

separated by a vertical bar, and surrounded by brackets. When used with pandas's `contains`, this string will find all values in the '`Product Name`' column of `cameras_df` that contain one of its terms. Why the vertical bars and the brackets around the separate terms? Those are just some of the symbols and operators you can use to define regular expression patterns — the rest, you will have to discover on your own.

Instead of the regular expression above, you could have combined several uses of the `contains` method and get the same result:

```
In [35]: cameras_df[
    (cameras_df['Product Name'].str.contains('Nikon'))
    | (cameras_df['Product Name'].str.contains('Canon'))
    | (cameras_df['Product Name'].str.contains('Kodak'))
]
```

```
Out [35]:      ProductID          Product Name   Total
117  C&P/KOD-01305  Kodak ZM1-NM 1 MP 1-Inch LCD CMOS Se...  64.10
287  C&P/KOD-01305  Kodak ZM1-NM 1 MP 1-Inch LCD CMOS Se...  64.10
616  C&P/KOD-32137  Kodak EasyShare Z990 12 MP Digital C...  166.30
2151 C&P/CAN-50721  Canon PowerShot SX50 HS 12MP Digital...  6.45
2280 C&P/CAN-50721  Canon PowerShot SX50 HS 12MP Digital...  6.45
...
12459 C&P/CAN-12514  Canon EOS Rebel T2i DSLR Camera (Bod...  35.40
12522 C&P/CAN-12514  Canon EOS Rebel T2i DSLR Camera (Bod...  35.40
12905 C&P/KOD-32137  Kodak EasyShare Z990 12 MP Digital C...  332.60
13594 C&P/NIK-92147  Nikon D3100 14.2MP Digital SLR Camer...  54.10
13986 C&P/NIK-92147  Nikon D3100 14.2MP Digital SLR Camer...  54.10
```

[77 rows × 3 columns]

While the result is identical, the regular expression example is much shorter. This is, perhaps, one of the main benefits of using regular expressions with pandas: they can simplify long text-based table filters, as the one above.

Besides finding patterns in text, regular expressions can also be used to extract pieces of information from text values. Notice that many of the product names above include a brand name (e.g., *Nikon*), as well as a model identifier (e.g., *D3100*). You can extract both the brand name and the model identifier from the product names using a simple regular expression and pandas's `extract` method:

```
In [36]: pattern = '(Nikon|Canon|Kodak)\s+(\w*)'

cameras_df['Product Name'].str.extract(pattern)
```

	0	1
117	Kodak	ZM1
287	Kodak	ZM1
616	Kodak	EasyShare
2151	Canon	PowerShot

```
2280  Canon  PowerShot
...
12459  Canon        EOS
12522  Canon        EOS
12905  Kodak  EasyShare
13594  Nikon        D3100
13986  Nikon        D3100
```

[77 rows x 2 columns]

The output above is a `DataFrame` with two columns created by extracting pieces of text from each value in `cameras_df`'s '`Product Name`' column.

The `pattern` used to create this `DataFrame` matches text values that include one of the brand names (i.e., `'(Nikon|Canon|Kodak)'`), followed by one or more whitespace characters (i.e., `'\s+'`), in turn followed by zero or more word characters (i.e., `'(\w*)'`). Notice that the `pattern` string also includes two sets of characters surrounded by brackets — these are called regular expression groups and define which parts of text get extracted (i.e., in this case, the brand name and the word that immediately follows it). The `extract` method checks each value in the '`Product Name`' column against this pattern, and if there is a match, it returns the pieces of text corresponding to each group in a `DataFrame`.

This quick example of using `pandas` string methods with regular expressions hopefully highlighted what they're good at. If you want to learn more about Python's regular expressions, you can read the introductory tutorial available in Python's official documentation.⁵ But remember that regular expressions can be challenging even for the most advanced programmers — if you find them frustrating or hard to figure out, you are in the company of everyone who has ever used them.

5: At <https://docs.python.org/3/howto/regex.html#regex-howto>.

Summary

This chapter showed you how to use string methods in `pandas`. Manipulating text is a large part of working with data; fortunately, `pandas` is well equipped for even the most specialized text-handling use cases. It is just as prepared for handling dates, which is what we turn to in the following chapter.

Working with date columns

Handling dates is another typical task when working with data. Fortunately, pandas comes with a wide range of tools for manipulating dates — this chapter explores some of the most useful ones. First, as we did with strings, let's quickly revisit Python dates.

Revisiting Python dates

In chapter 9, I briefly mentioned that the Python standard library includes a module called `datetime`, which provides functionality for working with date and time values through its four different types of objects:

- ▶ `date` objects for working with date values (e.g., `2020/09/30`);
- ▶ `time` objects for working with time values (e.g., `13:04`);
- ▶ `datetime` objects for working with values that contain both date and time parts (e.g., `2020-11-04 00:05:23`);¹
- ▶ `timedelta` objects to represent arbitrary time intervals for date and time arithmetic (e.g., a number of days to add to a `date` variable).

The code from chapter 9 illustrating how you can use the `datetime` module and its objects is shown (and slightly expanded) below:

```
In [1]: from datetime import date, time, datetime, timedelta

today = date(2020, 9, 30)
now = datetime(2020, 9, 30, 12, 33)
birthday = date(1989, 3, 21)

days_of_holiday = timedelta(days=14)
minutes_of_nap = timedelta(minutes=30)
```

The `datetime` values above are just as easy to work with as Python strings or numbers. You can use comparison operators with `date` or `datetime` values, or perform any kind of date arithmetic by adding and subtracting `timedelta` values to or from dates:

```
In [2]: today + days_of_holiday
Out [2]: datetime.date(2020, 10, 14)

In [3]: now + minutes_of_nap
Out [3]: datetime.datetime(2020, 9, 30, 13, 3)

In [4]: today > birthday
```

1: To keep you on your toes, the `date` `time` module includes a `datetime` object type.

```
Out[4]: True
In[5]: today - birthday
Out[5]: datetime.timedelta(days=11516)
```

In addition to date arithmetic, you can transform `datetime` objects into Python strings that look friendlier than the output above. For instance, to transform the `birthday` variable above into a human-readable string, you can use the `strftime` method:

```
In[6]: birthday.strftime('%d/%m/%y')
Out[6]: '21/03/89'
In[7]: birthday.strftime('%A, %d %B, %Y')
Out[7]: 'Tuesday, 21 March, 1989'
```

The argument passed to `strftime` in the examples above specifies the output format (i.e., they are different combinations of date *format specifiers*). There are many of these format specifiers available — unfortunately, they're a bit cryptic and, based on my experience, impossible to remember. Table 22.1 below lists some of the most useful ones, but you can find all of them in Python's official documentation.²

2: At docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior.

Table 22.1: Some of the format specifiers available for `datetime` objects. These are described in the `datetime` documentation available at docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior. Examples in this table are based on the following `datetime` value: `datetime(2020, 9, 30, 7, 6, 5)` — which represents 7:06 AM on September 30th 2020.

Specifier	Description	Output
'%a'	Weekday abbreviated name.	Mon
'%A'	Weekday full name.	Monday
'%d'	Day of the month as a zero-padded whole number.	30
'%-d'	Day of the month as a whole number.	30
'%b'	Abbreviated month name.	Sep
'%B'	Full month name.	September
'%m'	Month as zero-padded whole number.	09
'%-m'	Month as a whole number.	9
'%y'	Year without century as zero-padded whole number.	13
'%Y'	Year with century as whole number.	2013
'%H'	Hour (24-hour clock) as zero-padded whole number.	07
'%-H'	Hour (24-hour clock) as whole number.	7
'%I'	Hour (12-hour clock) as zero-padded whole number.	07
'%-I'	Hour (12-hour clock) as whole number.	7
'%p'	AM or PM.	AM
'%M'	Minute as a zero-padded whole number.	06
'%-M'	Minute as whole number.	6
'%S'	Second as zero-padded whole number.	05
'%-S'	Second as whole number.	5

The reverse operation, converting a Python string to a date or time value, can be achieved using the `strptime` function³ available directly on the `datetime` module:

```
In [8]: my_date = '30/09/20'
datetime.strptime(my_date, '%d/%m/%y')
```

```
Out [8]: datetime.datetime(2020, 9, 30, 0, 0)
```

As with outputting dates to strings, you need to tell `strptime` what format the date string you want to transform into a date object is in, with the same specifiers listed in table 22.1 — here, because the date format is day/month/year, the second argument passed to `strptime` is `'%d/%m/%y'`.

All this is great, but you didn't make it so far to work with date values one-by-one. You can easily apply the operations above to date columns in a `DataFrame`, but as usual, with a slight `pandas` twist. Let's take a closer look.

Date columns

Whenever you read an Excel file with `read_excel`, `pandas` will recognize Excel date columns⁴ and store them as `datetime64` columns in the resulting `DataFrame`. In addition, `pandas` recognizes date values even if they are in “General” columns in Excel (if all values in those columns have the same date format). However, if your date columns are stored as “Text” in Excel, `pandas` will keep them as strings (you'll have to convert them yourself if you want to use them as `datetime` values).

If you take a look at `ledger_df`, two of its columns contain dates: `'Date'`, which stores the date of each sale, and `'Deadline'`, which stores product aging thresholds used for inventory management and reporting (i.e., products still in inventory beyond this date are considered aged):

```
In [9]: ledger_df.info()
```

```
Out [9]: <class 'pandas.core.frame.DataFrame'>
RangeIndex: 14054 entries, 0 to 14053
Data columns (total 12 columns):
 #   Column        Non-Null Count  Dtype  
--- 
 0   InvoiceNo     14054 non-null   int64  
 1   Channel       14054 non-null   object  
 2   Product Name  12362 non-null   object  
 3   ProductID    14054 non-null   object  
 4   Account       14054 non-null   object  
 5   AccountNo    14054 non-null   int64  
 6   Date          14054 non-null   datetime64[ns]
 7   Deadline      14054 non-null   object
```

³: The `f` in `strftime` stands for *format*, whereas the `p` in `strptime` stands for *parse*.

⁴: Columns that are formatted as “Short Date”, “Long Date” or “Time” in Excel.

```
8   Currency      14054 non-null  object
9   Unit Price    14054 non-null  float64
10  Quantity      14054 non-null  int64
11  Total         14054 non-null  float64
dtypes: datetime64[ns](1), float64(2), int64(3), object(6)
memory usage: 1.3+ MB
```

Notice that the '`Date`' column (column 6 in the listing above) is assigned the `datetime64` data type. Why `datetime64` and not just `datetime`, as with plain Python? Because pandas uses a different type of date object than Python, one designed to be more efficient when working with lots of date values at once. If you take a closer look at values in the '`Date`' column, you'll see that they are actually `Timestamp` objects:

```
In [10]: ledger_df['Date'].iloc[0]
Out [10]: Timestamp('2020-01-01 00:00:00')
```

`Timestamp` objects are the pandas equivalent of Python's `datetime` (they extend the `datetime64` data type that comes from `numpy`, one of the Python libraries pandas uses in its internal code). `Timestamp` and `datetime` are interchangeable in any code you write. You can create a pandas `Timestamp` and a Python `datetime` to convince yourself that they're the same thing:

```
In [11]: timestamp_date = pd.Timestamp(2020, 1, 1)
datetime_date = datetime(2020, 1, 1)

timestamp_date == datetime_date
```

```
Out [11]: True
```

The reason pandas uses `Timestamp` and `datetime64` is that they use less memory, are faster to work with, and include a few extra methods that Python's `datetime` objects don't have. These details can only be confusing, but all you need to remember is that `Timestamp` and `datetime` mean the same thing anywhere you see them in pandas code.

The other column in `ledger_df` that contains dates is '`Deadline`'. In the output of `info` above, notice the '`Deadline`' column is assigned the `object` data type instead of `datetime64` (pandas thinks values in the '`Deadline`' column are strings, not `Timestamp` values). If you take a look at its values, you'll see why — the '`Date`' column is neatly formatted, whereas dates in the '`Deadline`' column have different formats:

```
In [12]: ledger_df[['Date', 'Deadline']]
```

```
Out [12]:
```

	Date	Deadline
0	2020-01-01	11/23/19
1	2020-01-01	06/15/20
2	2020-01-01	05/07/20
3	2020-01-01	12/22/19
4	2020-01-01	12/22/19
...
14049	2020-01-31	February 23 2020
14050	2020-01-31	January 21 2020
14051	2020-01-31	March 22 2020
14052	2020-01-31	June 25 2020
14053	2020-01-31	February 01 2020

[14054 rows x 2 columns]

Often data is sourced from separate systems that use different date formats and put together in one common dataset — fixing date formats is a common data cleaning issue. Let's see how you can use pandas to fix values in the '`Deadline`' column.

Converting strings to dates

Values in the '`Deadline`' column are strings (i.e., text values) that look like dates in various formats (e.g., '`2-03-20`', '`04/23/20`', '`Sun Mar 29 00:00:00 2020`', '`August 02 2020`' are all values from the '`Deadline`' column). While you may think it takes a lot of work to convert these strings to date values, pandas doesn't — to convert them, you can use pandas's `to_datetime` function:

```
In [13]: pd.to_datetime(ledger_df['Deadline'])
```

```
Out [13]:
```

0	2019-11-23
1	2020-06-15
2	2020-05-07
3	2019-12-22
4	2019-12-22
...	
24019	2020-02-23
24020	2020-01-21
24021	2020-03-22
24022	2020-06-25
24023	2020-02-01

Name: Deadline, Length: 24024, dtype: datetime64[ns]

The output above is a neatly formatted Series of `datetime` values. The reason `to_datetime` didn't complain or show an error is that pandas tries to figure out the format of each date in the '`Deadline`' column, and if it finds a valid format that matches your dates, it will use it when converting date-strings to date-values. If it doesn't, it will prompt you with an error message.

However, `to_datetime` makes certain assumptions about the format of your dates that you should be aware of. For instance, `pandas` converts '`05/07/20`' to a date representing the 7th of May 2020 (i.e., `pandas` assumes the date is in `mm/dd/yy` format). If you want to be explicit about date format, you will have to use the `format` keyword argument with `to_datetime` and tell `pandas` how to interpret date-strings. Consider the following examples, where I use `to_datetime` to convert the same date-strings to different date-values:

```
In [14]: dates = pd.Series(['05/07/20', '05/08/20', '05/09/20'])

pd.to_datetime(dates)
```

```
Out [14]: 0    2020-05-07
1    2020-05-08
2    2020-05-09
dtype: datetime64[ns]
```

```
In [15]: pd.to_datetime(dates, format='%d/%m/%y')
```

```
Out [15]: 0    2020-07-05
1    2020-08-05
2    2020-09-05
dtype: datetime64[ns]
```

The first example above converts date-strings to date-values set in May 2020. In the second example, I told `pandas` to interpret the first part of each date-string as a day number (by passing an explicit date format to `to_datetime` with the `format` keyword argument) and it converts date-strings to date-values set in July, August, and September 2020.

Because of the assumptions it makes when parsing date-strings, if you're not explicit about date formats, `pandas` might get things wrong and convert a date-string to an incorrect date value.⁵ Checking each date might be needed in some cases, but you can also check the output of `to_datetime` using the `describe` method to see how spread the date values that come out of this conversion process are:

```
In [16]: pd.to_datetime(ledger_df['Deadline']).describe()
```

```
Out [16]: count    24024
mean      2020-03-31 01:03:46.573434112
min       2019-11-03 00:00:00
25%       2020-01-23 00:00:00
50%       2020-03-31 00:00:00
75%       2020-06-06 00:00:00
max       2020-08-26 00:00:00
Name: Deadline, dtype: object
```

In this case, after the conversion, the earliest date is `2019-11-03` and the latest is `2020-08-26` (i.e., `min` and `max` in the output above). In general, if the range of dates output by `describe` seems off, you

5: Date format guessing can also take a long time. For large datasets, you will start noticing this extra time. If you know the format your dates are in, passing a `format` argument to `to_datetime` speeds things up.

might want to investigate why. For now, these dates seem correct, and you can use them to fix values in the 'Deadline' column by assigning the output of `to_datetime` back to the column:

```
In [17]: ledger_df['Deadline'] = pd.to_datetime(ledger_df['Deadline'])
```

1

If you run `info` again on `ledger_df`, you will see that the 'Deadline' column now has the `datetime64` data type (column 7 below):

```
In [18]: ledger_df.info()
```

```
Out [18]: <class 'pandas.core.frame.DataFrame'>
RangeIndex: 14054 entries, 0 to 14053
Data columns (total 12 columns):
 #   Column      Non-Null Count  Dtype  
---  -- 
 0   InvoiceNo    14054 non-null   int64  
 1   Channel      14054 non-null   object  
 2   Product Name 12362 non-null   object  
 3   ProductID    14054 non-null   object  
 4   Account      14054 non-null   object  
 5   AccountNo    14054 non-null   int64  
 6   Date         14054 non-null   datetime64[ns]
 7   Deadline     14054 non-null   datetime64[ns]
 8   Currency     14054 non-null   object  
 9   Unit Price   14054 non-null   float64 
 10  Quantity     14054 non-null   int64  
 11  Total        14054 non-null   float64 
dtypes: datetime64[ns](2), float64(2), int64(3), object(5)
memory usage: 1.3+ MB
```

```
In [19]: ledger_df[['Date', 'Deadline']]
```

```
Out [19]:      Date  Deadline
 0   2020-01-01 2019-11-23
 1   2020-01-01 2020-06-15
 2   2020-01-01 2020-05-07
 3   2020-01-01 2019-12-22
 4   2020-01-01 2019-12-22
 ...
 ...
 14049 2020-01-31 2020-02-23
 14050 2020-01-31 2020-01-21
 14051 2020-01-31 2020-03-22
 14052 2020-01-31 2020-06-25
 14053 2020-01-31 2020-02-01

[14054 rows x 2 columns]
```

You can use `to_datetime` to convert any column that pandas doesn't recognize as containing dates. Converting strings to dates (i.e., `object` or `string` columns to `datetime64` columns) is important because it allows you to perform date arithmetic with those columns and enables a range of date-specific methods. Let's take a look at some of these date methods next.

Pandas date methods

Just like you use the `str` attribute to access string methods on a `Series` of string values, you can use the `dt` attribute on `datetime64` columns to access date-specific attributes or methods. For instance, if you want to extract the year from each value in the '`Deadline`' column, you can use:

```
In [20]: ledger_df['Deadline'].dt.year
```

```
Out [20]: 0      2019
1      2020
2      2020
3      2019
4      2019
...
14049    2020
14050    2020
14051    2020
14052    2020
14053    2020
Name: Deadline, Length: 14054, dtype: int64
```

This output is a `Series` of integers, representing the year component of each date value in the '`Deadline`' column of `ledger_df`.

The `dt` attribute (or `dt` accessor) is just a pandas trick to keep all date-related methods and attributes together, under a common name, so you know where to find them. It is available on any `Series` object that contains `datetime64` values.

In the code example above, notice that `year` is an attribute, not a method (i.e., it isn't followed by a set of brackets). There are several date attributes that you can access in the same style — table 22.2 below lists them.

Table 22.2: Attributes available on date columns. These attributes are documented in the official pandas documentation available at pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html.

Attribute name	Example	Description
<code>year</code>	<code>ledger_df['Deadline'].dt.year</code>	Returns the year of each date as a <code>Series</code> of integers.
<code>month</code>	<code>ledger_df['Deadline'].dt.month</code>	Returns the month of each date as a <code>Series</code> of integers.
<code>day</code>	<code>ledger_df['Deadline'].dt.day</code>	Returns the day of each date as a <code>Series</code> of integers.
<code>date</code>	<code>ledger_df['Deadline'].dt.date</code>	Returns the date component (without time) of each date as a <code>Series</code> of Python <code>date</code> values.
<code>time</code>	<code>ledger_df['Deadline'].dt.time</code>	Returns the time component (without date) of each date as a <code>Series</code> of Python <code>time</code> values.
<code>dayofyear</code>	<code>ledger_df['Deadline'].dt.dayofyear</code>	Returns the day of year for each date as a <code>Series</code> of integers (from 1 to 366).

weekofyear	<code>ledger_df['Deadline'].dt.weekofyear</code>	Returns the week of year for each date as a Series of integers (from 1 to 53).
dayofweek	<code>ledger_df['Deadline'].dt.dayofweek</code>	Returns the day-of-the-week number for each date as a Series of integers (0 for Monday, through 6 for Sunday).
quarter	<code>ledger_df['Deadline'].dt.quarter</code>	Returns the quarter of each date as a Series of integers (1 for Jan-Mar, 2 for Apr-Jun, etc.).
is_quarter_start	<code>ledger_df['Deadline'].dt.is_quarter_start</code>	Returns a boolean for each date indicating whether it is the starting date of the quarter.
is_quarter_end	<code>ledger_df['Deadline'].dt.is_quarter_end</code>	Returns a boolean for each date indicating whether it is the ending date of the quarter.

Besides the date attributes mentioned above, there are several methods you can access on date columns as well. Some of these methods are listed in table 22.3 below. For example, to get the day name for all values in the '`Deadline`' column, you can use:

```
In [21]: ledger_df['Deadline'].dt.day_name()
```

```
Out [21]: 0      Saturday
           1      Monday
           2    Thursday
           3     Sunday
           4     Sunday
           ...
          14049    Sunday
          14050   Tuesday
          14051    Sunday
          14052   Thursday
          14053   Saturday
Name: Deadline, Length: 14054, dtype: object
```

With `day_name`, you can even specify the language you want your day names in — for instance, to get German day names for all dates in the '`Deadline`' column:

```
In [22]: ledger_df['Deadline'].dt.day_name('de_DE')
```

```
Out [22]: 0      Samstag
           1      Montag
           2    Donnerstag
           3     Sonntag
           4     Sonntag
           ...
          14049    Sonntag
          14050   Dienstag
          14051    Sonntag
          14052   Donnerstag
          14053   Samstag
Name: Deadline, Length: 14054, dtype: object
```

Table 22.3: Methods available on date columns. These (and more) methods are documented in the official pandas documentation available at pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html.

Method name	Example	Description
day_name	<code>df['Product Deadline'].dt.day_name("de_DE")</code>	Return the day names of the dates, using the locale code specified as an argument (example returns day names in German). By default, uses system locale.
month_name	<code>(df['Product Deadline'].dt.month_name("en_US"))</code>	Same as above, but returns month names in U.S. English.
normalize	<code>df['Product Deadline'].dt.normalize()</code>	Converts the time component of each value to midnight (i.e., 00:00:00). This is useful when you want to remove time from date time values.
strftime	<code>(df['Product Deadline'].dt.strftime("%m/%d/%Y"))</code>	Converts dates to strings (same as <code>strftime</code> for single <code>datetime</code> values). Format needs to be specified as an argument.

Why are some of these attributes and some methods? Attributes are pieces of information (e.g., day number, year number) that are stored directly in the date object, whereas methods *produce* date-related information by performing some calculations using date attributes (and occasionally some external information).

Filtering date columns

You'll often need to filter tables based on date ranges or date values. Fortunately, filtering a `DataFrame` on one of its date columns is as straightforward as working with any other type of data.

Conditional statements on date columns in `pandas` return a boolean `Series` that you can then use for filtering. You can specify conditional statements on date columns by using date-strings as reference values. For example, to filter `ledger_df` and select all sales after mid-January 2020, you can use:

```
In [23]: ledger_df[ledger_df['Date'] > '15/01/2020']
```

```
Out [23]:   InvoiceNo      Channel      Product Name ... Unit Price Quantity  Total
7224        8756    Bullseye     Miele Type U AirCl... ...  11.71       23  269.33
7225        8757  Shoppe.com    Verizon LG G2 Chev... ...  22.21        5  111.05
7226        8758  Understock.com Coleman 5620B718G ... ...  9.67        3  29.01
7227        8759  Understock.com 12-Inch & 9-Inch S... ...  25.85        1  25.85
7228        8760      iBay.com   Coaster Oriental S... ...  2.40        2   4.80
...
14049       15581    Bullseye  AC Adapter/Power S... ...  28.72        8  229.76
14050       15582    Bullseye  Cisco Systems Giga... ...  33.39        1  33.39
14051       15583  Understock.com Philips AJ3116M/37... ...  4.18        1   4.18
14052       15584      iBay.com                   NaN ...  4.78       25 119.50
14053       15585  Understock.com  Sirius Satellite R... ...  33.16        2   66.32
```

[6830 rows x 12 columns]

The date-string above is in dd/mm/yyyy format, but you can specify reference dates in almost any other format and pandas will figure things out — all of the following examples are equivalent:

```
ledger_df[ledger_df['Date'] > '15th of January, 2020']
ledger_df[ledger_df['Date'] > '15/1/2020']
ledger_df[ledger_df['Date'] > '2020, Jan 15']
ledger_df[ledger_df['Date'] > '1-15-20']
ledger_df[ledger_df['Date'] > '2020/1/15']
```

Instead of date-strings as reference values, you can also use Python `datetime` variables for filtering. Using `datetime` variables instead of date-strings is often a good idea because you can perform date arithmetic while filtering, if you need to (whereas you can't with date-strings). For instance, you can do something like this:

```
In [25]: import datetime as dt

start_date = dt.datetime(2020, 1, 15)
end_date = dt.datetime(2020, 1, 20)

shift = dt.timedelta(days=1)

ledger_df[
    (ledger_df['Date'] > start_date - shift) &
    (ledger_df['Date'] < end_date + shift)
]
```

```
Out [25]:   InvoiceNo      Channel      Product Name ... Unit Price Quantity   Total
6748       8280  Shoppe.com  Fender 005-3191-00... ... 43.45      6  260.70
6749       8281 Understock.com  3M 6897 Black Head... ... 4.40      14  61.60
6750       8282 Understock.com Tarantula Sleeve W... ... 10.57      6  63.42
6751       8283        Walcart  Hubsan X4 H107C 2.... ... 5.46      1  5.46
6752       8284 Understock.com Reusable Particula... ... 20.68      2  41.36
...
...
9331      10800 Understock.com Samsung Galaxy S3 ... ... 12.28      17  208.76
9332      10818      iBay.com  Bushnell Velocity ... ... 5.26      3  15.78
9333      10821      iBay.com Vivitar V69379-SIL... ... 7.37      6  44.22
9334      10823 Understock.com Cat People / The C... ... 14.82     76 1126.32
9335      10833 Understock.com                  BANG ... 10.08      6  60.48
```

[2588 rows x 12 columns]

The benefit of this approach is that if you want to change the range used for filtering, you can simply modify the `shift` variable and re-run the cell, instead of having to edit date-strings manually.

For more complex date filters, you can even use the date attributes and methods I mentioned earlier. For instance, to filter `ledger_df` and keep all sales that age on Thursdays, in the fourth quarter of 2018 or 2019, you can use:

```
In [26]: ledger_df[
    (ledger_df['Deadline'].dt.year.isin([2018, 2019])) &
    (ledger_df['Deadline'].dt.quarter == 4) &
    (ledger_df['Deadline'].dt.day_name() == 'Thursday')
]
```

	InvoiceNo	Channel	Product Name	Unit Price	Quantity	Total
6	1538	Understock.com	Nerf N-Sports Weat...	31.36	9	282.24
22	1554	iBay.com	DR Strings Nickel ...	25.95	3	77.85
38	1570	Walcart	Tork Dispenser Nap...	13.08	1	13.08
65	1597	Walcart	Battery Back Door ...	5.23	17	88.91
85	1617	Walcart	Small Ooze Tube - ...	23.86	4	95.44
...
13806	15338	Understock.com	Nokia Lumia 520 8G...	16.99	3	50.97
13873	15236	Shoppe.com	The Sephra 16-Inch...	3.90	12	46.80
13900	15250	Shoppe.com	The Sephra 16-Inch...	3.90	12	46.80
13943	15475	Understock.com	Small Ooze Tube - ...	5.95	7	41.65
14000	15388	Understock.com	Nan	11.98	23	275.54

[550 rows x 12 columns]

Notice in the example above that you can chain regular Series methods to the output of dt methods — because many dt methods or attributes return numbers or strings, not datetime64 values.

Converting dates back to strings

When you write a DataFrame that has datetime64 columns back to an Excel spreadsheet with `to_excel`, all date values in the spreadsheet will be in the `yyyy-mm-dd hh:mm:ss` format. If you need a different date format in your Excel spreadsheet, you can convert date values back to strings with the `strftime` method — the same method that you can use to convert plain Python `datetime` objects to strings:

```
In [27]: ledger_df['Deadline'].dt.strftime("%A, %d %B, %Y")
```

0	Saturday, 23 November, 2019
1	Monday, 15 June, 2020
2	Thursday, 07 May, 2020
3	Sunday, 22 December, 2019
4	Sunday, 22 December, 2019
...	
14049	Sunday, 23 February, 2020
14050	Tuesday, 21 January, 2020
14051	Sunday, 22 March, 2020
14052	Thursday, 25 June, 2020
14053	Saturday, 01 February, 2020

Name: Deadline, Length: 14054, dtype: object

The date format specifiers you can use with `strftime` are the same ones listed earlier in table 22.1.

The output above is a `Series` of string values (so date methods don't work on this `Series` anymore). You can assign this `Series` back to the '`Deadline`' column if you want to keep deadline values in this new format, and you can easily convert it back to a `datetime64` `Series` using `to_datetime` whenever you need to use date methods on it again.

Pandas date arithmetic

Working with dates often means having to perform date arithmetic (e.g., subtracting two dates, adding a week to a sequence of dates, shifting all dates in a column to the last Friday of the month, etc.) or modifying date components in-place (e.g., changing the time on all dates in a column from 1PM to 2PM). Because these types of date transformations are common, `pandas` comes with the tools to make them straightforward.

There are several `pandas` objects you can use to transform date values in arbitrary ways:

- ▶ `Timedelta` for representing arbitrary differences in date or time like 35 days, 12 hours or 3 weeks (equivalent and interchangeable with Python's `timedelta` object I mentioned earlier in this chapter).
- ▶ Date offset objects for representing common changes to dates or times (e.g., `BusinessDay`, `BusinessHour`). Date offsets are a friendlier alternative to `Timedelta` and are useful when you want to do some date arithmetic without figuring out the exact number of days you need to add or subtract.
- ▶ `Period` for representing recurring periods of time. Periods are particularly useful for working with business quarters in different year-end setups.

Let's take a look at `Timedelta` objects first.

Timedeltas

The simplest operation you can do with date columns is to add or subtract an arbitrary period of time to or from them. This arbitrary period of time can be represented as a `Timedelta` object, which is part of the `pandas` library (i.e., just like `DataFrame` or `Series` objects are). For example, if you want to add 2 days to each value in the '`Deadline`' column, you can use:

```
In [28]: ledger_df['Deadline']
```

```
Out [28]: 0      2019-11-23
          1      2020-06-15
          2      2020-05-07
          3      2019-12-22
          4      2019-12-22
          ...
         14049    2020-02-23
         14050    2020-01-21
         14051    2020-03-22
         14052    2020-06-25
         14053    2020-02-01
Name: Deadline, Length: 14054, dtype: datetime64[ns]
```

```
In [29]: ledger_df['Deadline'] + pd.Timedelta(days=2)
```

```
Out [29]: 0      2019-11-25
          1      2020-06-17
          2      2020-05-09
          3      2019-12-24
          4      2019-12-24
          ...
         14049    2020-02-25
         14050    2020-01-23
         14051    2020-03-24
         14052    2020-06-27
         14053    2020-02-03
Name: Deadline, Length: 14054, dtype: datetime64[ns]
```

Timedelta arithmetic only works on `datetime64` Series, and the output is another `datetime64` Series. To remove 4 weeks from each value, you can subtract a `Timedelta` just as easily:

```
In [30]: ledger_df['Deadline'] - pd.Timedelta(weeks=4)
```

```
Out [30]: 0      2019-10-26
          1      2020-05-18
          2      2020-04-09
          3      2019-11-24
          4      2019-11-24
          ...
         14049    2020-01-26
         14050    2019-12-24
         14051    2020-02-23
         14052    2020-05-28
         14053    2020-01-04
Name: Deadline, Length: 14054, dtype: datetime64[ns]
```

The arguments available when you create a `Timedelta` object are `weeks`, `days`, `hours`, `minutes` and `seconds`⁶ and you can use any combination of arguments to define the time interval you need:

6: And `milliseconds`, `microseconds`, `nanoseconds` if you need to be that precise with your dates.

```
In [31]: ledger_df['Deadline'] + pd.Timedelta(weeks=4, days=3, hours=2, minutes=1)
```

```
Out[31]: 0      2019-12-24 02:01:00
         1      2020-07-16 02:01:00
         2      2020-06-07 02:01:00
         3      2020-01-22 02:01:00
         4      2020-01-22 02:01:00
         ...
        14049    2020-03-25 02:01:00
        14050    2020-02-21 02:01:00
        14051    2020-04-22 02:01:00
        14052    2020-07-26 02:01:00
        14053    2020-03-03 02:01:00
Name: Deadline, Length: 14054, dtype: datetime64[ns]
```

Besides adding or removing arbitrary time intervals, `Timedelta` objects are also used to represent the difference between date values. For example, if you subtract the '`Date`' and '`Deadline`' columns in `ledger_df`, you will get a Series of `Timedelta` values:

```
In [32]: ledger_df['Date'] - ledger_df['Deadline']
```

```
Out[32]: 0      39 days
         1     -166 days
         2     -127 days
         3      10 days
         4      10 days
         ...
        14049    -23 days
        14050    10 days
        14051    -51 days
        14052   -146 days
        14053    -1 days
Length: 14054, dtype: timedelta64[ns]
```

The Series output by this operation has the `timedelta64` data type, but if you inspect any of its values, you will see that it is just a `Timedelta` object — `timedelta64` is the data type assigned by pandas to columns with `Timedelta` objects.

```
In [33]: (ledger_df['Date'] - ledger_df['Deadline']).iloc[0]
```

```
Out[33]: Timedelta('39 days 00:00:00')
```

An interesting feature of `Timedelta` objects is that you can perform arithmetic with them. For instance, you can add two `Timedelta` objects together, to get a different time interval:

```
In [34]: pd.Timedelta(days=2) + pd.Timedelta(weeks=1)
```

```
Out[34]: Timedelta('9 days 00:00:00')
```

Perhaps more useful is that you can divide `Timedelta` objects to get a time interval in a different unit. For instance, if you subtract the two date columns in `ledger_df`, you get a time difference between dates expressed in days, as you saw above (days is the default time interval for `Timedelta` objects). To express this difference in hours,

you can simply divide the output of subtracting the two columns by a `Timedelta` that uses the hour time interval:

```
In [35]: (ledger_df['Date'] - ledger_df['Deadline']) / pd.Timedelta(hours=1)
```

```
Out [35]: 0      936.0
1     -3984.0
2     -3048.0
3      240.0
4      240.0
...
14049    -552.0
14050    240.0
14051   -1224.0
14052   -3504.0
14053    -24.0
Length: 14054, dtype: float64
```

Now you have the difference between the two date columns expressed in hours instead of days. Notice, however, that the output is a numerical Series (i.e., it has the `float64` data type), and is no longer a Series of `Timedelta` values.

Although flexible, date transformations with `Timedelta` objects are only useful when you know exactly how many weeks, days, or hours you want to add or subtract from your dates. When you want to shift all dates in a column by two business days using `Timedelta`, you need to keep track of weekends and business hours, and it can all get messy fast. Luckily, `pandas` comes with a few date offset objects that can handle the details for you.

Date offsets

The first offset object we'll look at is called, appropriately, `DateOffset`. `DateOffset` is similar to `Timedelta`: you access it in the same way, it accepts similar keyword arguments, and produces the same results. For instance, to add 2 days to each value in the '`Deadline`' column you can use:

```
In [36]: # equivalent to
# ledger_df['Deadline'] + pd.Timedelta(days=2)
ledger_df['Deadline'] + pd.DateOffset(days=2)
```

```
Out [36]: 0      2019-11-25
1      2020-06-17
2      2020-05-09
3      2019-12-24
4      2019-12-24
...
14049    2020-02-25
14050    2020-01-23
14051    2020-03-24
14052    2020-06-27
```

```
14053    2020-02-03
Name: Deadline, Length: 14054, dtype: datetime64[ns]
```

The `DateOffset` object has a few more features than `Timedelta`, which is why it gets a separate section in this chapter. One of these features is that you can also use the `years` and `months` keyword arguments to create longer time intervals if you need to — `Timedelta` only accepts `weeks` as its longest unit of time:

```
In [37]: ledger_df['Deadline'] + pd.DateOffset(years=2, months=5)
```

```
Out [37]: 0      2022-04-23
1      2022-11-15
2      2022-10-07
3      2022-05-22
4      2022-05-22
...
14049   2022-07-23
14050   2022-06-21
14051   2022-08-22
14052   2022-11-25
14053   2022-07-01
Name: Deadline, Length: 14054, dtype: datetime64[ns]
```

Even more useful is that you can use it to modify part of your dates. For instance, if you want to set the year of all dates in the '`Date`' column to 1999, you can use the following `DateOffset` operation:

```
In [38]: ledger_df['Deadline'] + pd.DateOffset(year=1999)
```

```
Out [38]: 0      1999-11-23
1      1999-06-15
2      1999-05-07
3      1999-12-22
4      1999-12-22
...
14049   1999-02-23
14050   1999-01-21
14051   1999-03-22
14052   1999-06-25
14053   1999-02-01
Name: Deadline, Length: 14054, dtype: datetime64[ns]
```

This returns a new `datetime64` Series with all dates set in 1999. The slight difference from the previous example where you added 2 years to each date is that the keyword argument used with `pd.DateOffset` above is in the singular: `year` not `years`.

Using `DateOffset` this way is the easiest way to modify part of your dates (i.e., without calculating how much you need to shift each date and then using a `Timedelta`). You can use almost all the keyword arguments in the singular — the one that doesn't work in the singular is `week` (i.e., you can shift a date by a number of weeks, but you cannot set a date's week number). The keyword arguments

that are available in the singular form are `year`, `day`, `hour`, `minute`, `second`, `microsecond` and `nanosecond` — and, as before, you can use any combination to get the results you need.

For more human-friendly date arithmetic, `pandas` comes with several predefined date offset objects. These predefined offsets extend `DateOffset` and are part of a nested submodule in `pandas`. To access them, you need to `import` the following submodule:⁷

7: `tseries` stands for *time series*.

```
In [39]: import pandas.tseries.offsets as offsets
```

After you run the code above, remember you can use JupyterLab's autocomplete feature to see what predefined offsets are available — in a separate cell, type `offsets.` and press the TAB key:

```
In [40]: offsets.<TAB>
```

One of the available offsets that I mentioned earlier is `BusinessDay`. You can use it to shift dates by a number of business days without worrying about business hours or weekends. Let's quickly make a reference date variable, to show an example of using the `BusinessDay` offset:

```
In [41]: reference_date = pd.Timestamp(2020, 9, 30)

reference_date.day_name()
```

```
Out [41]: 'Wednesday'
```

This reference date is a Wednesday. If you add three business days to it, the result will be, as expected, the next Monday:

```
In [42]: (reference_date + offsets.BusinessDay(3)).day_name()
```

```
Out [42]: 'Monday'
```

As with `Timedelta` or `DateOffset`, you can also subtract predefined offsets to go back in time. For instance, to shift `reference_date` to Easter, three years ago, without having to figure out leap years and changing Easter dates, you can use:

```
In [43]: reference_date - offsets.Easter(3)
```

```
Out [43]: Timestamp('2018-04-01 00:00:00')
```

The same kind of date arithmetic with offsets works on entire `DataFrame` columns as well. To shift dates from the '`Deadline`' column forwards by 3 business days, you can use:

```
In [44]: ledger_df['Deadline'] + offsets.BusinessDay(3)
```

```
Out[44]: 0    2019-11-27
1    2020-06-18
2    2020-05-12
3    2019-12-25
4    2019-12-25
...
14049 2020-02-26
14050 2020-01-24
14051 2020-03-25
14052 2020-06-30
14053 2020-02-05
Name: Deadline, Length: 14054, dtype: datetime64[ns]
```

Besides the offsets used above, there are several other predefined date offsets you can use (and each accepts different keyword arguments) — table 22.4 lists some of them.

Table 22.4: Predefined DateOffset types available from the `pandas.tseries.offsets` submodule. These offset types (and more) are documented in the official `pandas` documentation available at pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#dateoffset-objects.

All examples use `date = pd.Timestamp(year=2020, month=3, day=12, hour=12, minute=30)` which represents 12:30 AM on March 12th 2020.

Offset type	Example	Description
<code>BMonthBegin</code>	<code>date + offsets.BMonthBegin()</code> <code>Timestamp('2020-04-01 12:30:00')</code>	Date offset to nearest beginning of business month in the future.
<code>BMonthEnd</code>	<code>date + offsets.BMonthEnd(3)</code> <code>Timestamp('2020-05-29 12:30:00')</code>	Date offset to third nearest end of business month in the future.
<code>BYearBegin</code>	<code>date + offsets.BYearBegin()</code> <code>Timestamp('2021-01-01 12:30:00')</code>	Date offset to nearest beginning of business year in the future.
	<code>date + BYearBegin(month=3)</code> <code>Timestamp('2021-03-01 12:30:00')</code>	Same as above, but specifies starting month of business year.
<code>BYearEnd</code>	<code>date + offsets.BYearEnd()</code> <code>Timestamp('2020-12-31 12:30:00')</code>	Same as above, but offsets to nearest end of business year in the future (<code>month</code> parameter can be used).
<code>BusinessDay</code>	<code>date - offsets.BusinessDay(7)</code> <code>Timestamp('2020-03-03 12:30:00')</code>	Offsets by seven business days in the past.
<code>BusinessHour</code>	<code>date + offsets.BusinessHour()</code> <code>Timestamp('2020-03-13 12:30:00')</code>	Offsets by one business hour in the future.
	<code>date + offsets.BusinessHour(start='08:00', end='13:00')</code> <code>Timestamp('2020-03-13 08:30:00')</code>	Same as above, but specifies custom business hours.
<code>Easter</code>	<code>date + offsets.Easter()</code> <code>Timestamp('2020-03-13 12:30:00')</code>	Offsets to nearest Easter in the future.

WeekOfMonth	<pre>date + offsets.WeekOfMonth(week=2, weekday=4) Timestamp('2020-03-20 12:30:00')</pre>	Offsets to nearest Friday of the second week of the calendar month. <code>week</code> and <code>weekday</code> can be used to specify which week day or week number (in a month) to offset to.
-------------	---	--

Keep in mind that if you try to use these predefined offsets with a date column that contains missing values (i.e., `NaN`, `NaT` or `NA`), you will get an error, whereas `Timedelta` date arithmetic will work, but keeps missing values in the output.

Periods

Besides dates and date offsets, you can also create regular intervals of time using pandas's `Period` object (i.e., a number of business days, a calendar month, a quarter, etc.). For accounting, `Period` objects are particularly useful for converting calendar dates to fiscal quarters in different year-end setups.

The easiest way to use `Period` objects is by converting a `Series` of dates into a `Series` of periods with the `to_period` date method. For instance, to convert all dates in the '`Deadline`' column into a `Series` of quarterly periods, you can use:

```
In [45]: ledger_df['Deadline'].dt.to_period(freq='Q')
```

```
Out [45]: 0      2019Q4  
1      2020Q2  
2      2020Q2  
3      2019Q4  
4      2019Q4  
...  
14049    2020Q1  
14050    2020Q1  
14051    2020Q1  
14052    2020Q2  
14053    2020Q1  
Name: Deadline, Length: 14054, dtype: period[Q-DEC]
```

The output above is a `Series` of `Period` objects. So what are these periods exactly? They are labels for a span of time in a given calendar: in this case, they show which quarter each date in the '`Deadline`' column belongs to in a standard calendar year (i.e., a calendar year that ends in December). If you want to specify a different year-end month, you can pass a different value to the `freq` keyword argument when using `to_period`:

```
In [46]: ledger_df['Deadline'].dt.to_period(freq='Q-SEP')
```

	2020-10-24																	
freq='M'	JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC						
freq='Q-DEC'	2020Q1			2020Q2			2020Q3			2020Q4								
freq='Q-SEP'	2020Q2			2020Q3			2020Q4			2021Q1								
freq='A-DEC'	2020																	
freq='A-SEP'	2020									2021								

Figure 22.1: Illustration of converting 2020-10-24 to periods. Each row represents a succession of periods (i.e., time spans) that have different lengths and year-end setups. The dotted line crosses each row to show the period representation of 2020-10-24 when using different frequency values with `to_period`.

```
Out [46]: 0      2020Q1
1      2020Q3
2      2020Q3
3      2020Q1
4      2020Q1
...
14049   2020Q2
14050   2020Q2
14051   2020Q2
14052   2020Q3
14053   2020Q2
Name: Deadline, Length: 14054, dtype: period[Q-SEP]
```

Now the quarter labels above are based on a year (i.e., a 12-month period) that ends in September. By default, December is used as the year-end month, however, any other month label is valid, as long as it is specified in the same style (i.e., '`'Q-JAN'` through '`'Q-DEC'`).

The values in both `Series` output above are quarter labels, but you can use different frequencies with `to_period` to convert dates to other time intervals. For instance, to convert dates in the '`Deadline`' column to yearly period labels:

```
In [47]: # A stands for annual
ledger_df['Deadline'].dt.to_period(freq='A')
```

```
Out [47]: 0      2019
1      2020
2      2020
3      2019
4      2019
...
14049   2020
14050   2020
```

```
14051    2020
14052    2020
14053    2020
Name: Deadline, Length: 14054, dtype: period[A-DEC]
```

And as with quarters, you can also specify the year-end month:

```
In [48]: ledger_df['Deadline'].dt.to_period(freq='A-MAR')
```

```
Out [48]: 0      2020
1      2021
2      2021
3      2020
4      2020
...
14049    2020
14050    2020
14051    2020
14052    2021
14053    2020
Name: Deadline, Length: 14054, dtype: period[A-MAR]
```

Because `period` Series don't contain dates, you can't manipulate them using the date arithmetic tools from the previous section. However, you can transform them back into date-values using the `to_timestamp` method:

```
In [49]: periods = ledger_df['Deadline'].dt.to_period(freq='Q-SEP')
```

```
periods
```

```
Out [49]: 0      2020Q1
1      2020Q3
2      2020Q3
3      2020Q1
4      2020Q1
...
14049    2020Q2
14050    2020Q2
14051    2020Q2
14052    2020Q3
14053    2020Q2
Name: Deadline, Length: 14054, dtype: period[Q-SEP]
```

```
In [50]: periods.dt.to_timestamp()
```

```
Out [50]: 0      2019-10-01
1      2020-04-01
2      2020-04-01
3      2019-10-01
4      2019-10-01
...
14049    2020-01-01
14050    2020-01-01
14051    2020-01-01
14052    2020-04-01
```

```
14053    2020-01-01
Name: Deadline, Length: 14054, dtype: datetime64[ns]
```

The `to_timestamp` method will convert each period to the first date of the period (in the example above, each quarter becomes the first date of the quarter). Its output is a familiar `datetime64 Series` (i.e., a column of dates), that you can manipulate using date offsets or any other `pandas` date methods.

The `to_period` and `to_timestamp` methods used in this section are only available through the `dt` attribute (just like the other date-specific `Series` methods).

Periods can be used for complex mappings of dates to calendar intervals (e.g., business-hours, recurring holidays, weekly shifts). If you work with financial reports that include specific time intervals, definitely consider reading more about periods in `pandas`.⁸

8: At pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#time-span-representation.

Overthinking: Timezones

Something you may come across in your work is reconciling data with dates from different timezones. While this might not be a common issue, particularly when working with standard accounting datasets such as a general ledger, some online companies store real-time ledgers where timezone information can be used to distinguish entries.

Excel can't store timezones with any of its dates: all dates in your spreadsheets are in the same timezone as your computer. You can, of course, have timezone information as a separate column in a spreadsheet, but if you have lots of different timezones flying around, things get tricky fast.

In `pandas`, you can store timezone information directly with your date values. By default, when you read data from Excel, date values are timezone agnostic (they contain no information about their timezone), but you can assign them a timezone with the `tz_localize`⁹ method available for `datetime64` columns:

9: `tz` for timezone.

```
In [51]: us_pacific_deadlines = ledger_df['Deadline'].dt.tz_localize('US/Pacific')
```

```
us_pacific_deadlines
```

```
Out [51]: 0      2019-11-23 00:00:00-08:00
1      2020-06-15 00:00:00-07:00
2      2020-05-07 00:00:00-07:00
3      2019-12-22 00:00:00-08:00
4      2019-12-22 00:00:00-08:00
...
14049    2020-02-23 00:00:00-08:00
14050    2020-01-21 00:00:00-08:00
```

```
14051 2020-03-22 00:00:00-07:00
14052 2020-06-25 00:00:00-07:00
14053 2020-02-01 00:00:00-08:00
Name: Deadline, Length: 14054, dtype: datetime64[ns, US/Pacific]
```

The data type associated with the Series above now shows you what timezone dates are in (i.e., datetime64[ns, US/Pacific]). Note, however, that this doesn't shift the original dates (or times) to the new timezone in any way; it merely assigns them a timezone. The dates are the same as before, but with an extra label that says they're in the 'US/Pacific' timezone.

If you need to convert timezone-aware dates to a different timezone, you can use `tz_convert`:¹⁰

```
In [52]: us_pacific_deadlines.dt.tz_convert('US/Eastern')
```

```
Out [52]: 0      2019-11-23 03:00:00-05:00
1      2020-06-15 03:00:00-04:00
2      2020-05-07 03:00:00-04:00
3      2019-12-22 03:00:00-05:00
4      2019-12-22 03:00:00-05:00
...
14049 2020-02-23 03:00:00-05:00
14050 2020-01-21 03:00:00-05:00
14051 2020-03-22 03:00:00-04:00
14052 2020-06-25 03:00:00-04:00
14053 2020-02-01 03:00:00-05:00
Name: Deadline, Length: 14054, dtype: datetime64[ns, US/
Eastern]
```

```
In [53]: us_pacific_deadlines.dt.tz_convert('Europe/Berlin')
```

```
Out [53]: 0      2019-11-23 09:00:00+01:00
1      2020-06-15 09:00:00+02:00
2      2020-05-07 09:00:00+02:00
3      2019-12-22 09:00:00+01:00
4      2019-12-22 09:00:00+01:00
...
14049 2020-02-23 09:00:00+01:00
14050 2020-01-21 09:00:00+01:00
14051 2020-03-22 08:00:00+01:00
14052 2020-06-25 09:00:00+02:00
14053 2020-02-01 09:00:00+01:00
Name: Deadline, Length: 14054, dtype: datetime64[ns, Europe/Berlin]
```

Conversely, if you need to remove timezone information from a datetime64 column, you can use `tz_localize` and pass `None` as the argument:

```
In [54]: us_pacific_deadlines.dt.tz_localize(None)
```

¹⁰ To list all the timezone labels you can use with `tz_convert`, run `import pytz; pytz.common_timezones` in a separate cell.

```
Out [54]: 0      2019-11-23
1      2020-06-15
2      2020-05-07
3      2019-12-22
4      2019-12-22
...
14049  2020-02-23
14050  2020-01-21
14051  2020-03-22
14052  2020-06-25
14053  2020-02-01
Name: Deadline, Length: 14054, dtype: datetime64[ns]
```

Once you add timezone information to your dates, you won't be able to compare or subtract them from other dates that are timezone agnostic. For instance, if you try to compare timezone-aware sale dates to the original deadline date, you will see a fairly explicit error message:

```
In [55]: ledger_df['Date'].dt.tz_localize('US/Pacific') > ledger_df['Deadline']
```

```
Out [55]: -----
TypeError                                 Traceback (most recent call last)
<ipython-input-170-229733c76b64> in <module>
----> 1 pd.to_datetime('2020-10-1') == pd.to_datetime('2020-10-1').tz_localize('US/Pacific')
...
TypeError: Cannot compare tz-naive and tz-aware timestamps
```

Summary

This chapter showed you how to use date methods in `pandas`. Cleaning, converting, or formatting dates is just as common when working with data as handling text. Fortunately, `pandas` is well equipped for even the most specialized date-handling use cases.

Even though `pandas` has many built-in functions, you'll sometimes need to craft and use custom functions on your data. Let's see how you can define a custom Python function and apply it to the rows or columns of a `DataFrame` next.

Applying custom functions

You've already seen by now that pandas has a lot of functions in its toolkit — for working with strings or dates, for complex table filters, for reading and writing Excel files, etc. However, even a toolkit as diverse as this can't have everything you need: you will often have to write your own data transformation functions to get the results you want. Luckily, pandas makes it straightforward to use custom functions on the rows or columns of a `DataFrame` through the `apply` method.

This chapter explores how you can use the `apply` method in its various configurations — let's start by applying a custom function on a single `DataFrame` column.

Applying functions to columns

You might remember from chapter 8 that you can create custom functions in Python using the `def` keyword:¹

```
In [1]: def process_channel(channel):
    return 'Name: ' + channel.upper()
```

The `process_channel` function above is just a block of code with a name. You can use this name to *call* the function and run its code as many times as you need to:

```
In [2]: process_channel('Shoppe.com')
Out [2]: 'Name: SHOPPE.COM'
In [3]: process_channel('Bullseye')
Out [3]: 'Name: BULLSEYE'
```

The value you pass between the parentheses when calling the function is called the function argument. This value gets bound to the `channel`² variable inside the function body. This value is used throughout the function code and any result gets returned as the function output. The `process_channel` function above is fairly basic, but you can craft any custom function using the Python building blocks we covered in part 1 of the book.

If you wanted to call this function for each value in the '`Channel`' column, you could use a loop — something similar to:

1: Remember that you need to indent the code that is in the function block. You also need to put a colon (i.e., `:`) after the function name and the parentheses.

2: `channel` is just a name I chose, you can use any variable or parameter name inside the function definition, as long as it is a valid Python name (i.e., it starts with a letter and doesn't contain spaces).

```
for channel in ledger_df['Channel']:
    process_channel(channel)
```

However, pandas makes it much easier to get the same result without a loop — you can call the `process_channel` function for each value in a `DataFrame` column at once using the `apply` method:

```
In [5]: ledger_df['Channel'].apply(process_channel)
```

```
Out [5]: 0           Name: SHOPPE.COM
1           Name: WALCART
2           Name: BULLSEYE
3           Name: BULLSEYE
4           Name: BULLSEYE
...
14049       Name: BULLSEYE
14050       Name: BULLSEYE
14051       Name: UNDERSTOCK.COM
14052       Name: IBAY.COM
14053       Name: UNDERSTOCK.COM
Name: Channel, Length: 14054, dtype: object
```

It looks like magic, but it's just pandas.

When you apply a function on a `DataFrame` column, pandas goes through the column values one-by-one, calls your custom function, and passes each value as an argument to your function. The output from each function call is then put in a new `Series` object — that has the same number of rows as the column you called `apply` on — which is the result you see above. Using `apply` is much faster than `for` loops because pandas is optimized for working with entire sequences of values at once.

Notice in the example above that just the *function name* is passed to `apply`, without any parentheses after it (i.e., `apply`'s argument is `process_channel`, not `process_channel()`). This is because pandas handles calling the function for you; you just need to tell it which function to call.

You can apply custom functions this way to any `DataFrame` column, regardless of its data type. However, when defining a custom function, you need to be mindful of the values you want to apply it to. In this example, I knew I wanted to apply the `process_channel` function to a column of strings. As such, I knew that each value passed to the function will be a Python string, and that Python strings have an `upper` method. If you try to apply `process_channel` on the '`AccountNo`' column, you'll get an error because values in that column are not strings but integers, and `process_channel` is designed to work only with strings:

```
In [6]: ledger_df['AccountNo'].apply(process_channel)
```

Out [6]:

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-14-481209cb6e12> in <module>
      1 ledger_df['AccountNo'].apply(process_channel)
      2
      3
<ipython-input-8-c918cb524d81> in process_channel(value)
      1 def process_channel(value):
      2     return 'Name: ' + value.upper()

AttributeError: 'int' object has no attribute 'upper'
```

Even if you are mindful of the data you apply your custom function on, you'll still get an error if the data contains `NaN` values. Consider this example:

In [7]:

```
def process_product(product):
    return 'Product: ' + product.upper()

ledger_df['Product Name'].apply(process_product)
```

Out [7]:

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-17-9437884125ac> in <module>
      1 ledger_df['Product Name'].apply(process_product)
      2
      3
<ipython-input-16-9d5a94895fe8> in process_product(product)
      1 def process_product(product):
      2     return 'Product: ' + product.upper()

AttributeError: 'float' object has no attribute 'upper'
```

While the `process_product` function above is identical to the `process_channel` function, and both '`Product Name`' and '`Channel`' columns contain string values, the '`Product Name`' also contains `NaN` values. These `NaN` values don't have an `upper` method associated with them (how do you make a missing value uppercase?!). The last line in the error message above tells you exactly that: `NaN` values (which `pandas` considers `float` values) don't have an `upper` attribute associated with them. You need to keep this in mind whenever using `apply` on columns that contain `NANs` or different types of values.

To fix `process_product`, you can make³ it handle `NANs` explicitly:

In [8]:

```
def process_product(product):
    if pd.isna(product):
        return 'EMPTY PRODUCT NAME'
    else:
        return 'Product: ' + product.upper()
```

3: Or you can fill in missing values in the '`Product Name`' column with a dummy string, using `fillna`, and then apply the function.

Now the `process_product` function handles missing values: if the argument passed to it is a `NaN` value, the function returns a message string; otherwise, it returns a processed product name. Now, if you

apply the new version of `process_product` to the '`Product Name`' column, you get:

```
In [9]: ledger_df['Product Name'].apply(process_product)
```

```
Out [9]: 0      Product: CANNON WA...
1      Product: LEGO NINJ...
2      EMPTY PRODUCT NAME
3      Product: TRANSFORM...
4      Product: TRANSFORM...
...
14049    Product: AC ADAPTE...
14050    Product: CISCO SYS...
14051    Product: PHILIPS A...
14052    EMPTY PRODUCT NAME
14053    Product: SIRIUS SA...
Name: Product Name, Length: 14054, dtype: object
```

Notice that all missing values in the '`Product Name`' column have been replaced with `EMPTY PRODUCT NAME`.

You saw in the previous chapters that there are a lot of string and date methods built-into `pandas`. Many of those methods can be written as custom functions and applied to your data's string or date columns. There are many roads through the `pandas` woods and which one you choose is up to you. However, keep in mind that `pandas`'s built-in methods, such as the string or date methods I mentioned in the previous chapters, typically skip missing values for you (i.e., they return `NaN` whenever they get a `NaN` as input), whereas with your own functions, you have to handle `NANs` yourself.

Overthinking: Functions without a name

You'll often need to apply one-off, concise transformations to your columns: defining a new one-line function every time you need some specific transformation can get annoying (and uses up a lot of space in your notebooks). Instead of having many one-line functions that are useful once, you can use anonymous functions. For example, instead of using `process_channel` with `apply`, you can get the same result by calling:

```
In [10]: ledger_df['Channel'].apply(lambda channel: 'Name: ' + channel.upper())
```

```
Out [10]: 0      Name: SHOPPE.COM
1      Name: WALCART
2      Name: BULLSEYE
3      Name: BULLSEYE
4      Name: BULLSEYE
...
14049    Name: BULLSEYE
14050    Name: BULLSEYE
```

```
14051    Name: UNDERSTOCK.COM
14052        Name: IBAY.COM
14053    Name: UNDERSTOCK.COM
Name: Channel, Length: 14054, dtype: object
```

You define anonymous functions using the `lambda` keyword. After `lambda`, you need to specify a parameter name followed by a colon and a return value. The following functions do the same thing:

```
# regular function definition
def process_channel(channel):
    return 'Name: ' + channel.upper()

# anonymous function definition
lambda channel: 'Name: ' + channel.upper()
```

Anonymous functions are convenient for one-off, short transformations of data. They have some limitations (you can only have one function parameter), so you'll need to use regular Python functions for more complex code. Still, they're often a convenient shortcut.

Applying functions to rows

The two functions you defined in the previous section can only be applied to one column at a time. But what if you need to use values from several columns in your custom function? For instance, you might want to calculate a different tax amount for each sale item, depending on what channel it occurred in — which means you need both `'Total'` and `'Channel'` values in your function. In that case, you can define a custom function and apply it to each row in your `DataFrame`, instead of applying it to a column.

To figure out how you can do that, let's start by selecting the first row in `ledger_df` (using `iloc`) and assigning it to a variable:

```
In [12]: first_row = ledger_df.iloc[0]
```

```
first_row
```

```
Out [12]: InvoiceNo          1532
Channel           Shoppe.com
Product Name     Cannon Water Bomb ...
ProductID         T&G/CAN-97509
Account           Sales
AccountNo        5004
Date      2020-01-01 00:00:00
Deadline        11/23/19
Currency          USD
Unit Price       20.11
Quantity          14
Total            281.54
Name: 0, dtype: object
```

Just like columns, DataFrame rows are Series objects — `first_row` is a Series object. You can access values from `first_row` using square brackets and column names:

```
In [13]: first_row['Total']
```

```
Out [13]: 281.54
```

```
In [14]: first_row['Channel']
```

```
Out [14]: 'Shoppe.com'
```

Because we want to calculate a different tax value for each row in `ledger_df` depending on its sale channel, we need a custom function that works with entire rows (not single values). Let's create a new function that takes a row (i.e., a Series object) as its argument and computes this conditional tax amount:

```
In [15]: def calculate_tax(row):
    if row['Channel'] == 'Shoppe.com':
        return row['Total'] * (16 / 100)
    elif row['Channel'] == 'iBay.com':
        return row['Total'] * (11 / 100)
    elif row['Channel'] == 'Understock.com':
        return row['Total'] * (9 / 100)
    else:
        return 0
```

The function above calculates a 16% tax value on '`Shoppe.com`' sales, an 11% tax on '`iBay.com`' sales, and a 9% tax on '`Understock.com`' sales. The tax for sales in any other channel is set to 0 (notice the last branch of the `if-else` statement). You can call this function on any row in `ledger_df`:

```
In [16]: calculate_tax(first_row)
```

```
Out [16]: 45.0464
```

```
In [17]: calculate_tax(ledger_df.iloc[10])
```

```
Out [17]: 0
```

```
In [18]: calculate_tax(ledger_df.iloc[100])
```

```
Out [18]: 3.355
```

You could loop through each row in `ledger_df` and call this function yourself (e.g., using a `for` loop) but it's much easier to use the `apply` method. To `apply` `calculate_tax` on each row in `ledger_df`, you can use:

```
In [19]: ledger_df.apply(calculate_tax, axis='columns')
```

```
Out[19]: 0      45.0464
1      0.0000
2      0.0000
3      0.0000
4      0.0000
...
14049    0.0000
14050    0.0000
14051    0.3762
14052    13.1450
14053    5.9688
Length: 14054, dtype: float64
```

The result above is a `Series` with the conditional tax amounts we wanted to calculate. You can assign this `Series` to a new column in `ledger_df` if you need to keep it in your table.

When you `apply` a custom function on the rows of `DataFrame` like we did here, `pandas` goes through each row in the `DataFrame`, one-by-one, and calls your function, passing the row as an argument. As with using `apply` on a single column, the output from each function call is put in a separate `Series` with the same number of rows as your `DataFrame`.

In this example, `pandas` knows to pass entire rows as arguments for your custom function (instead of column values) because of the `axis='columns'` keyword argument used with `apply`. As before, notice that just your function's name is passed to `apply`, without any parentheses after it.⁴

4: `pandas` handles the function calling; you just need to tell it which function you want it to call.

Overthinking: Other function parameters

The `calculate_tax` function above is not very flexible: right now, tax levels are baked into the function code. If you decide to use different tax levels for each channel, you need to edit the function and run `apply` again. Similarly, if you want to add other tax levels (e.g., for '`Walcart`' and '`Bullseye`'), you need to edit the function and add new branches to the `if-else` statement. In the real world, you might also need to apply different versions of this function on other datasets, in which case you might end up with several similar functions in your notebook: `calculate_tax2`, `calculate_tax3`, `calculate_tax_v4_final` — I've been there.

You can make your functions more flexible (and more general) by adding more parameters to their definition. For instance, to add channel-specific tax levels as a separate parameter to `calculate_tax`, you can use:

```
In [20]: def calculate_tax(row, levels={}):
    channel = row['Channel']
    total = row['Total']

    tax = 0
    if channel in levels:
        tax = levels[channel]

    return total * tax
```

Although it took the same number of lines to write, this version of `calculate_tax` is more flexible than the previous one. There are a few changes worth highlighting:

- ▶ **Line 1** adds another parameter to the function called `levels`, with an empty Python dictionary as its default value. Later, when applying `calculate_tax` to `ledger_df`, you'll set `levels` to a dictionary mapping channel names to their associated tax level.
- ▶ **Lines 2 and 3** create two variables inside the function body to store the row values we want to use later. This is not strictly necessary but makes the code easier to read.
- ▶ **Line 5** creates a `tax` variable with a default value of 0.
- ▶ **Line 6 and 7** represent the actual innovation: instead of using an `if-else` statement to determine tax levels, the `levels` dictionary is used to look up the channel's tax level. Right now, the `level` dictionary is empty (that is its default value), but when applying `calculate_tax` to `ledger_df`, we will give it a more useful value.
- ▶ **Line 9** returns the calculated tax value.

You can now apply this version of `calculate_tax` using:

```
In [21]: ledger_df.apply(calculate_tax, axis='columns')
```

```
Out [21]: 0      0.0
1      0.0
2      0.0
3      0.0
4      0.0
...
14049   0.0
14050   0.0
14051   0.0
14052   0.0
14053   0.0
Length: 14054, dtype: float64
```

This is clearly not the same result as before. The output is all zeros because we didn't pass a dictionary mapping channel names to tax rates to `calculate_tax`. To make our function work as before,

you need to pass the following Python dictionary as the `levels` keyword argument when using `apply`:

```
In [22]: ledger_df.apply(
    calculate_tax,
    levels={
        'Shoppe.com': (16 / 100),
        'iBay.com': (11 / 100),
        'Understock.com': (9 / 100),
    },
    axis='columns'
)
```

```
Out [22]: 0      45.0464
1      0.0000
2      0.0000
3      0.0000
4      0.0000
...
14049     0.0000
14050     0.0000
14051     0.3762
14052     13.1450
14053     5.9688
Length: 14054, dtype: float64
```

Now the output is the same as before. The benefit of this version of `calculate_tax` is that you can easily change or add new tax rates when applying it, without changing its code at all. For instance, if you want to add different tax rates for all channels in our data, you can use:

```
In [23]: ledger_df.apply(
    calculate_tax,
    levels={
        'Shoppe.com': (16 / 100),
        'iBay.com': (11 / 100),
        'Understock.com': (9 / 100),
        'Bullseye': (6 / 100),
        'Walcart': (4 / 100),
    },
    axis='columns'
)
```

```
Out [23]: 0      45.0464
1      0.2680
2      3.5010
3      4.8456
4      4.8456
...
14049     13.7856
14050     2.0034
14051     0.3762
14052     13.1450
14053     5.9688
```

```
Length: 14054, dtype: float64
```

The custom functions you use with `apply` (whether on columns or rows) can be designed with multiple parameters. When using `apply`, you will need to set values for these extra parameters, as I did above with `levels`. However, there is one constraint: your custom function's first parameter is always a value or a row set by `pandas` when applying the function. Any other parameters that you want to set yourself when applying the function need to come second and after in the order of parameters.

In general, it is a good idea to make functions that you want to use frequently more flexible by adding extra parameters. However, there is always a trade-off between flexibility and how easy it is to call the function when you need it (e.g., notice that calling `calculate_tax` now takes up several lines of code) — as with everything, finding a balance takes practice.

Summary

This chapter showed you how to apply custom Python functions to the rows or columns of a `DataFrame`. In both cases, you use the `apply DataFrame` method. However, depending on how you want to apply your custom function, you need to design it to accept a single value or a `Series` as its first argument.

The last few chapters have introduced some of `pandas`'s more specialized tools. Let's see how you can use these tools to extract insights from product reviews next.

Project: Mining product reviews

Free-form¹ text is standard in data — even the QuickBooks general ledger you saw in chapter 20 had a *Memo* column with comments about each record in the ledger. This project chapter shows you how to use the `pandas` tools you've been reading about in the last few chapters to extract insights from a product reviews dataset.

To get started, launch JupyterLab and open this chapter's notebook — you'll find the notebook in your *Python for Accounting* workspace in the “P2 - Working with tables” folder. As always, the first line of code you need to run is:

```
In [1]: import pandas as pd
```

The reviews dataset you'll use for this project is in the “*project_data*” folder. The dataset is a CSV file, so you'll need `pandas`'s `read_csv` function instead of `read_excel` to load the data into a `DataFrame`:

```
In [2]: reviews_df = pd.read_csv('project_data/reviews.csv')
```

```
reviews_df
```

```
Out [2]:
```

	ProductID	Product Name	Rating	Review	Date
0	K&D/WMF-26982	WMF Manaos / Bistro Ic...	5	I love it	43660
1	K&D/CUI-82621	Cuisinart DCC-2600 Bre...	5	Worth the money	42600
2	K&D/HAM-06147	Hamilton Beach 22708 T...	5	toaster/oven	43738
3	K&D/FOX-77328	Fox Run Salt and Peppe...	5	Nice Shaker Set.	44118
4	K&D/OST-12291	Oster TSSTTVVG01 4-Sli...	5	PRETTY & EASY TO USE	43758
...
73902	K&D/BUN-34597	BUNN ST Velocity Brew ...	4	High quality construct...	42920
73903	K&D/CHE-65796	Chefmaster KTGR5 13-In...	4	Excellent for your he...	43639
73904	K&D/FRE-21195	French Press Coffee Ma...	3	Didn't know there was ...	43685
73905	K&D/CUI-32446	Cuisinart CPT-142 Comp...	2	Wildly Inaccurate Sett...	43888
73906	K&D/NOR-01856	Nordic Ware 60120 Micr...	4	Imperfect, but So Handy!	43961

[73907 rows x 5 columns]

Each row in `reviews_df` above represents an online review for a *Kitchen & Dining* product supplied by the same wholesale distributor that generated the sales data you've been working with in the previous chapters.

There are five columns in `reviews_df`: the '`ProductID`' and '`Product Name`' columns should be familiar by now, as they're also part of the sales data; the '`Rating`' column is a value from 1 (lowest rating) to 5 (highest rating) assigned by the buyer to a product; the '`Review`' column contains the free-form text you'll be analyzing to see what buyers like or dislike about the products they purchase; and the

1: As opposed to structured text, like product IDs or sales channel names.

'Date' column is the date of each review, stored as an Excel date serial number.

You can call the `info` method on `reviews_df` to see what data type each column is and if there are any missing values:

```
In [3]: reviews_df.info()
```

```
Out [3]: <class 'pandas.core.frame.DataFrame'>
RangeIndex: 73907 entries, 0 to 73906
Data columns (total 5 columns):
 #   Column      Non-Null Count  Dtype  
--- 
 0   ProductID    73907 non-null   object  
 1   Product Name 73907 non-null   object  
 2   Rating        73907 non-null   int64  
 3   Review         73907 non-null   object  
 4   Date          73907 non-null   int64  
dtypes: int64(2), object(3)
memory usage: 2.8+ MB
```

There are three parts to this project: first, you'll need to find the most recent product reviews. There are a lot of reviews in the table above, and many of them are outdated. Second, you'll need to process each review and turn it into a list of descriptive words. Last, you'll count review words to figure out what makes people like or dislike the products they purchase.

Let's start by making the 'Date' column in `reviews_df` easier to understand.

Keeping recent reviews

Right now, values in the 'Date' column represent the number of days between the 1st of January 1900 and each review's date:

```
In [4]: reviews_df['Date']
```

```
Out [4]: 0      43660
1      42600
2      43738
3      44118
4      43758
...
73902    42920
73903    43639
73904    43685
73905    43888
73906    43961
Name: Date, Length: 73907, dtype: int64
```

This is how Excel stores dates “under-the-hood” so they can be used in calculations with other numerical values.² At times, Excel’s dates get saved as numbers and you’ll have to turn them back into dates yourself. If you open “*reviews.csv*” in Excel and set the ‘**Date**’ column to the *Long Date* column type, you’ll see its values turn into human-readable dates. In `pandas`, to turn these numbers into `datetime` values, you’ll need to run:

```
In [5]: pd.to_datetime(reviews_df['Date'], unit='D', origin='1899-12-30')
```

```
Out [5]: 0      2019-07-14
1      2016-08-18
2      2019-09-30
3      2020-10-14
4      2019-10-20
...
73902  2017-07-04
73903  2019-06-23
73904  2019-08-08
73905  2020-02-27
73906  2020-05-10
Name: Date, Length: 73907, dtype: datetime64[ns]
```

You’ve seen the `to_datetime` function in action before. In this example, you use the `unit` keyword arguments to tell `pandas` that the values you want to convert represent a number of days (as opposed to a number of nanoseconds, which is the default unit). Similarly, the `origin` keyword argument tells `pandas` that days are counted up from a starting date of 30th of December 1899. Why didn’t I set the `origin` keyword argument to the 1st of January 1900 if Excel stores dates as the number of days that have elapsed since then? Because `pandas` expects the `origin` keyword argument to be the day before day 0 — for Excel, the 1st of January 1900 is day number 1, which means day number 0 is the 31st of December 1899, and the day before (which `to_datetime` needs as its `origin` keyword argument) is the 30th of December 1899. If you ever find Excel’s date numbers in your data and want to convert them to Python `datetime` values, the code above is what you need to run.

Now that you know how to convert Excel’s date-numbers to `date` `time` values, let’s assign them back to the ‘**Date**’ column:

```
In [6]: reviews_df['Date'] = pd.to_datetime(reviews_df['Date'], unit='D', origin='1899-12-30')
```

You can use several of `pandas`’s date methods to check how spread review dates are. One option is to extract the year from each date, then count how many times each year appears in the data:

```
In [7]: reviews_df['Date'].dt.year.value_counts().sort_index()
```

2: In fact, Python and `pandas` use a similar approach. In Python, `date time` values represent the number of nanoseconds that have elapsed since the 1st of January 1970.

```
Out [7]: 2006      6
2007     28
2008     33
2009     86
2010     80
2011    151
2012    322
2013    794
2014   1285
2015   1722
2016   2680
2017   4483
2018   7229
2019  20279
2020 33308
2021 1421
Name: Date, dtype: int64
```

There are many outdated reviews — the earliest ones are from 2006, which wouldn't tell you much about buyers' current preferences. Let's filter `reviews_df` and keep reviews from 2020 or later only:

```
In [8]: reviews_df = reviews_df[reviews_df['Date'].dt.year >= 2020]

reviews_df
```

```
Out [8]:      ProductID      Product Name  Rating      Review      Date
3      K&D/FOX-77328  Fox Run Salt and...      5  Nice Shaker Set. 2020-10-14
7      K&D/PRO-00930  Progressive Inte...      5  Works great 2020-09-08
8      K&D/HIG-61873  Highwin P1001-8 ...      5  Loved it :) 2020-12-11
9      K&D/BAR-93217  Baratza Encore -...      5  Great deal 2020-04-30
12     K&D/BAM-78441  Bamboo Cutting B...      5  I'm oiling them ... 2020-10-15
...
...
73895  K&D/TRU-30514  Trudeau Melamine...      1  Broke easily 2020-11-26
73896  K&D/MOR-18621  Morning Mug (1)      1  NOT GOOD 2020-07-15
73900  K&D/NIF-37634  Nifty Ice Cream ...      4  Works well 2020-02-11
73905  K&D/CUI-32446  Cuisinart CPT-14...      2  Wildly Inaccurat... 2020-02-27
73906  K&D/NOR-01856  Nordic Ware 6012...      4  Imperfect, but S... 2020-05-10
```

[34729 rows x 5 columns]

There are plenty of reviews left to work with; let's now use pandas to dig into what people are saying.

Processing review text

As you may have noticed, there's a lot of variation in the way people write reviews (i.e., what words they use, how long their reviews are, how they use punctuation, letter case, etc.). Most free-form text is like this; therefore, the first thing you need to do before analyzing text is making it more uniform. Making text uniform typically involves the following steps:

- ▶ Converting text to lowercase only;
- ▶ Removing trailing whitespace;
- ▶ Removing punctuation characters;
- ▶ Splitting text into its constituent words;
- ▶ Removing words that don't carry meaning (i.e., "a", "the", etc.).

There are other steps you can take when preparing text (e.g., extracting the stem of each word so that "loving it" and "love it" become the same string), but for now, let's apply the steps above to the reviews data using `pandas` and Python.

To make all review text lowercase and remove trailing whitespace characters you can use `pandas`'s string methods:

```
In [9]: reviews_df['Review'] = reviews_df['Review'].str.lower()
reviews_df['Review'] = reviews_df['Review'].str.strip()
```

1
2

To remove punctuation, let's define a custom function and apply it to the `'Review'` column. To start, you'll need a list of punctuation characters that you want to remove. You can type the characters yourself, but Python's `string` module (which is part of Python's standard library, so you already have it installed) has a punctuation variable you can use instead:

```
In [10]: import string

string.punctuation
```

```
Out [10]: '!"#%&\'()*+,-./:;=>?@[\\]^_`{|}~'
```

The `string.punctuation` variable is a string containing common punctuation characters. Python strings are sequences of characters, so you can use them as you use Python lists (i.e., go over their characters using a `for` loop). Let's use the punctuation characters in a custom function that takes as input a single review and replaces each punctuation character in the review with an empty string:

```
In [11]: def remove_punctuation(review):
    for character in string.punctuation:
        review = review.replace(character, '')

    return review
```

If you call this function and pass a string value as its argument, you'll get the same string back but without any punctuation characters:

```
In [12]: remove_punctuation('Great quality!!! Much nicer than expected -- but expensive.')
```

```
Out [12]: 'Great quality much nicer than expected but expensive'
```

You can now apply this custom function to all reviews with:

```
In [13]: reviews_df['Review'] = reviews_df['Review'].apply(remove_punctuation)
```

Next you need to split each review into a list of words. You can do that with pandas's `split` string function — the code below assigns the list of words to a new column in `reviews_df`:

```
In [14]: reviews_df['Review Words'] = reviews_df['Review'].str.split()
```

```
In [15]: reviews_df[['Review', 'Review Words']]
```

```
Out [15]:
```

	Review	Review Words
3	nice shaker set	[nice, shaker, set]
7	works great	[works, great]
8	loved it	[loved, it]
9	great deal	[great, deal]
12	im oiling them now nice lavish and...	[im, oiling, them, now, nice, lavish...]
...
73895	broke easily	[broke, easily]
73896	not good	[not, good]
73900	works well	[works, well]
73905	wildly inaccurate settings	[wildly, inaccurate, settings]
73906	imperfect but so handy	[imperfect, but, so, handy]

[34729 rows x 2 columns]

The reviews seem cleaner now. However, you still have a lot of words that don't carry meaning (e.g., "it", "but", "so", etc.). In natural language processing, these words are called *stopwords* — they're noise, not information. Just like you used a list of characters to remove punctuation, you can use a list of stopwords to remove unhelpful words from the `'Review Words'` column. I've added a stopwords dataset in the `"project_data"` folder so you don't have to create one yourself — to load it into a Python list, you can run:

```
In [16]: stopwords = pd.read_csv('project_data/stopwords.csv', squeeze=True)
stopwords = list(stopwords)
```

The `squeeze` keyword argument used with `read_csv` above tells pandas to turn the DataFrame returned by `read_csv` into a Series object if there's only one column (which is the case here). You can check what words are in the list by printing it or by running:

```
In [17]: 'it' in stopwords
```

```
Out [17]: True
```

```
In [18]: 'product' in stopwords
```

```
Out [18]: False
```

As with punctuation, let's define a custom function that removes stopwords from a word list:

```
In [19]: def remove_stopwords(word_list):
    return [w for w in word_list if w not in stopwords]
```

The function above uses a Python list comprehension to go through a sequence of words (i.e., the list assigned to the `word_list` parameter when calling the function) and returns a new list of words, with only those words that aren't in the `stopwords` dataset. If you pass it a list of words, you get:

```
In [20]: remove_stopwords(['imperfect', 'but', 'so', 'handy'])
```

```
Out [20]: ['imperfect', 'handy']
```

You can now apply the `remove_stopwords` function to all values in the 'Review Words' column using:

```
In [21]: reviews_df['Review Words'] = reviews_df['Review Words'].apply(remove_stopwords)

reviews_df[['Review', 'Review Words']]
```

```
Out [21]:
```

	Review	Review Words
3	nice shaker set	[nice, shaker, set]
7	works great	[works, great]
8	loved it	[loved]
9	great deal	[great, deal]
12	im oiling them now nice lavish and...	[oiling, nice, lavish, durable]
...
73895	broke easily	[broke, easily]
73896	not good	[good]
73900	works well	[works]
73905	wildly inaccurate settings	[wildly, inaccurate, settings]
73906	imperfect but so handy	[imperfect, handy]

[34729 rows x 2 columns]

Now that you have a clean list of informative words for each review let's see why people like or dislike the products they bought by counting words.

Counting words

Counting words to figure out what text is about may seem simple, but it's the most common technique used to unravel human language with computers — even advanced natural language processing algorithms count words one way or another.

Before you start counting words, let's first break the reviews dataset into two separate `DataFrame` variables: one with the most positive reviews and another with the most negative. It's easier to understand why someone liked a product when you're sure they liked it (based on the product rating). Luckily, for `reviews_df` you can use values in the 'Rating' column to determine reviewers'

sentiment about the product they bought, and create the two DataFrame variables:

```
In [22]: negative_reviews_df = reviews_df.loc[reviews_df['Rating'] == 1]
positive_reviews_df = reviews_df.loc[(reviews_df['Rating'] == 5)]
```

To count review words, let's create two Series variables with all the words used for the negative and positive reviews, respectively:

```
In [23]: negative_words = pd.Series(negative_reviews_df['Review Words'].sum())
positive_words = pd.Series(positive_reviews_df['Review Words'].sum())
```

You can now use these variables to count how often different words appear in the product reviews:

```
In [24]: negative_words.value_counts()
```

```
Out [24]: money      213
junk        205
poor        196
use         190
work        188
...
these       1
bumps       1
note         1
decanter    1
candydeep   1
Length: 3490, dtype: int64
```

```
In [25]: positive_words.value_counts()
```

```
Out [25]: great        5602
love         2123
good         1452
works        1430
perfect      1419
...
stariner     1
analog       1
slicershreddergrinder  1
autosiphon   1
oneida       1
Length: 7330, dtype: int64
```

It's not surprising to see words like *money*, *junk*, and *poor* used frequently with negative reviews, and words like *great*, *love*, and *good* used for positive reviews. However, you can use this simple count to dive deeper into the reviews and find the ones that speak about specific product features or qualities.

For instance, you can try to find out what's so '*great*' about the products that people review. To do that, you can use a simple regular expression and extract all words that follow '*great*' in positive reviews, with the following pandas code:

```
In [26]: positive_pattern = '(great .*)'

positive_reviews_df['Review'].str.extract(positive_pattern).value_counts()
```

```
Out [26]: great product           353
great price                   63
great value                   57
great quality                 52
great item                    50
...
great for picnics             1
great for picnics and barbeques 1
great for pizza                1
great zucchini noodles        1
```

The regular expression above (i.e., the `positive_pattern` string variable) finds the word '`great`' in all reviews and extracts the words that follow it. The `value_counts` method counts how often different pieces of text are found in the reviews — it seems buyers like the "`price`" and "`quality`" of the products they buy.

The same idea works for negative reviews, where you can see why people think products are '`poor`':

```
In [27]: negative_pattern = '(poor .*)'

negative_reviews_df['Review'].str.extract(negative_pattern).value_counts()
```

```
Out [27]: poor quality            49
poor design                  12
poor execution                7
poor quality control          6
poor product                  5
...
poor frying experience        1
poor for bread making         1
poor fit into standard size   1
poor workmanship for cuisinart 1
Length: 103, dtype: int64
```

You can dig deeper into the reviews and design more complicated regular expressions that match different product features or qualities (e.g., '`(great coffee .*)`').

Summary

This quick project chapter showed you how to use `pandas`'s date and string functions to analyze product reviews.³ Now, let's get back to uncovering more of `pandas`'s features and see how you can use it to combine, merge, and pivot tables.

³: The `nltk` and `spaCy` Python libraries are worth exploring if you work with text often.

Concatenating tables

Working with a single file or dataset can only get you so far. To make the most of your data, you will often need to combine multiple spreadsheets or Excel files into a single `DataFrame` and leverage connections between datasets.

The two main tools for combining tables in `pandas` are its `concat` (short for *concatenate*) and `merge` functions. While `concat` is the `pandas` equivalent of copy-pasting rows (or columns) from one sheet to another, `merge` is similar to Excel's VLOOKUP function. This chapter shows you how to combine multiple tables into a single `DataFrame` using the `concat` function. The following chapter takes a look at joining tables on their common columns with `pandas`'s `merge` function.

Row-wise concatenation

You can use the `concat` function for row-wise or column-wise concatenation of `DataFrame` or `Series` objects. Row-wise concatenation is the `pandas` equivalent of copying rows from one spreadsheet and pasting them at the bottom of another.

For the code examples in this section, let's read a small subset of our sales data into several `DataFrame` variables and then concatenate them using `concat`. To read each sheet of "Q1Sales.xlsx" as a separate `DataFrame`, you can use:

```
In [1]: cols = ['ProductID', 'Quantity', 'Total']  
1  
2  
jan_df = pd.read_excel('Q1Sales.xlsx', sheet_name='January', usecols=cols, nrows=5)  
3  
feb_df = pd.read_excel('Q1Sales.xlsx', sheet_name='February', usecols=cols, nrows=5)  
4  
mar_df = pd.read_excel('Q1Sales.xlsx', sheet_name='March', usecols=cols, nrows=5)  
5
```

In the code above, you use several keyword arguments with `read_excel`. On line 1, you define a list of columns to read from each sheet of the sales Excel file. You then pass this list of columns to each `read_excel` call, using the `use_cols` keyword argument. You also use `nrows=5` and tell `pandas` to read only the top five rows of each sheet (we are keeping these `DataFrame` objects small to help illustrate how `concat` works).

After you run the code above, you should have three `DataFrame` variables in your notebook, each with five rows and three columns:

```
In [2]: jan_df
```

```
Out [2]:
```

	ProductID	Quantity	Total
0	T&G/CAN-97509	14	281.54
1	T&G/LEG-37777	1	6.70
2	T&G/PET-14209	5	58.35
3	T&G/TRA-20170	6	80.76
4	T&G/TRA-20170	6	80.76

```
In [3]: feb_df
```

```
Out [3]:
```

	ProductID	Quantity	Total
0	C&P/NEW-62681	7	17.08
1	I&S/RUB-56368	2	8.70
2	M&T/7TH-34490	3	13.98
3	H&PC/MED-46454	14	90.02
4	T&G/100-26579	1	12.69

```
In [4]: mar_df
```

```
Out [4]:
```

	ProductID	Quantity	Total
0	MI/VIC-46664	6	134.34
1	MI/ARC-23043	1	25.65
2	MI/AKG-35546	2	17.96
3	CP&A/LE-28028	9	74.97
4	K&D/STA-02514	80	1436.80

You can now row-wise concatenate these variables into a single DataFrame using the `concat` function.¹

```
In [5]: pd.concat([jan_df, feb_df, mar_df])
```

```
Out [5]:
```

	ProductID	Quantity	Total
0	T&G/CAN-97509	14	281.54
1	T&G/LEG-37777	1	6.70
2	T&G/PET-14209	5	58.35
3	T&G/TRA-20170	6	80.76
4	T&G/TRA-20170	6	80.76
0	C&P/NEW-62681	7	17.08
1	I&S/RUB-56368	2	8.70
2	M&T/7TH-34490	3	13.98
3	H&PC/MED-46454	14	90.02
4	T&G/100-26579	1	12.69
0	MI/VIC-46664	6	134.34
1	MI/ARC-23043	1	25.65
2	MI/AKG-35546	2	17.96
3	CP&A/LE-28028	9	74.97
4	K&D/STA-02514	80	1436.80

¹: Because `concat` can be used with multiple DataFrame objects at once, it is a top-level function in pandas (i.e., it is not a DataFrame method).

The argument passed to `concat` is a Python list containing the separate DataFrame objects you want to put together. The output above is a new DataFrame with the same rows as `jan_df`, `feb_df` and `mar_df` put together.

Notice that row labels are preserved from the original DataFrame objects (i.e., row labels in the output above are not consecutive). If you need to make row labels consecutive and discard the original labels, you can use the `ignore_index=True` keyword argument:

```
In [6]: pd.concat([jan_df, feb_df, mar_df], ignore_index=True)
```

```
Out [6]:   ProductID  Quantity  Total
0    T&G/CAN-97509      14  281.54
1    T&G/LEG-37777       1    6.70
2    T&G/PET-14209       5   58.35
3    T&G/TRA-20170       6   80.76
4    T&G/TRA-20170       6   80.76
5    C&P/NEW-62681        7   17.08
6    I&S/RUB-56368        2    8.70
7    M&T/7TH-34490        3   13.98
8    H&PC/MED-46454      14  90.02
9    T&G/100-26579        1   12.69
10   MI/VIC-46664        6  134.34
11   MI/ARC-23043        1   25.65
12   MI/AKG-35546        2   17.96
13   CP&A/LE-28028        9   74.97
14   K&D/STA-02514       80 1436.80
```

When you concatenate DataFrame objects, there's nothing that tells you what DataFrame each row is from (in the output above, you don't know which rows are from `mar_df` and which ones from `jan_df`). If you need to keep a record of where each row is originally from, instead of passing a list of DataFrame variables to `concat`, you can pass a Python dictionary:

```
In [7]: pd.concat({'Jan': jan_df, 'Feb': feb_df, 'Mar': mar_df})
```

```
Out [7]:   ProductID  Quantity  Total
Jan 0    T&G/CAN-97509      14  281.54
         1    T&G/LEG-37777       1    6.70
         2    T&G/PET-14209       5   58.35
         3    T&G/TRA-20170       6   80.76
         4    T&G/TRA-20170       6   80.76
Feb 0    C&P/NEW-62681        7   17.08
         1    I&S/RUB-56368        2    8.70
         2    M&T/7TH-34490        3   13.98
         3    H&PC/MED-46454      14  90.02
         4    T&G/100-26579        1   12.69
Mar 0    MI/VIC-46664        6  134.34
         1    MI/ARC-23043        1   25.65
         2    MI/AKG-35546        2   17.96
         3    CP&A/LE-28028        9   74.97
         4    K&D/STA-02514       80 1436.80
```

This DataFrame looks slightly different from all the other ones you've seen so far: each row has two associated labels (notice the two left-most columns above). These row labels are stored in a *MultiIndex* — you can assign the DataFrame above to another variable and inspect its `index` to see what it looks like:

```
In [8]: df = pd.concat({'Jan': jan_df, 'Feb': feb_df, 'Mar': mar_df})
```

```
df.index
```

```
Out[8]: MultiIndex([(Jan, 0),
                     (Jan, 1),
                     (Jan, 2),
                     (Jan, 3),
                     (Jan, 4),
                     (Feb, 0),
                     (Feb, 1),
                     (Feb, 2),
                     (Feb, 3),
                     (Feb, 4),
                     (Mar, 0),
                     (Mar, 1),
                     (Mar, 2),
                     (Mar, 3),
                     (Mar, 4)],
                    )
```

With this `MultiIndex DataFrame`, if you want to access certain rows or values, you need to use `loc` with one or two row labels (i.e., one for each level of the index). For instance, to select all rows that have the label '`Mar`', you can use:

```
In[9]: df.loc['Mar']
```

```
Out[9]:   ProductID  Quantity    Total
0  MI/VIC-46664        6  134.34
1  MI/ARC-23043        1   25.65
2  MI/AKG-35546        2   17.96
3  CP&A/LE-28028        9   74.97
4  K&D/STA-02514       80 1436.80
```

Or if you want to access a single row, you can use:

```
In[10]: df.loc['Mar', 0]
```

```
Out[10]: ProductID    MI/VIC-46664
Quantity          6
Total            134.34
Name: (Mar, 0), dtype: object
```

As you continue working with `pandas`, you'll come across `DataFrame` objects with multi-level row labels, such as the one above, more and more — and even `DataFrame` objects with multi-level column labels. There's nothing special about these multi-level labels: just like regular labels, they identify rows or columns in a table.

Going back to `concat`, notice that all the previous examples combined `DataFrame` objects with the same set of columns (i.e., each one of the three `DataFrame` objects had the same columns). If you concatenate `DataFrame` objects that have different columns, the resulting `DataFrame` will have a union of columns from each separate `DataFrame`. For example, you can select a subset of columns from two of the `DataFrame` variables you created earlier and concatenate them with:

```
In [11]: pd.concat([
    jan_df[['ProductID', 'Quantity']],
    feb_df[['ProductID', 'Total']]
])
```

```
Out [11]:      ProductID  Quantity  Total
0   T&G/CAN-97509     14.0    NaN
1   T&G/LEG-37777      1.0    NaN
2   T&G/PET-14209      5.0    NaN
3   T&G/TRA-20170      6.0    NaN
4   T&G/TRA-20170      6.0    NaN
0   C&P/NEW-62681      NaN  17.08
1   I&S/RUB-56368      NaN   8.70
2   M&T/7TH-34490      NaN  13.98
3   H&PC/MED-46454      NaN  90.02
4   T&G/100-26579      NaN  12.69
```

By default, the `concat` function doesn't mind if the tables you pass as input have the same columns: the resulting table will have all columns, from all input `DataFrame` objects, regardless of whether they are shared. Where the input tables don't have common columns, the output is filled with `Nan`s, as above. If you want the resulting `DataFrame` to keep only those columns that are common among all the separate input `DataFrame` variables, you can use the `join='inner'` keyword argument with `concat`:

```
In [12]: pd.concat([
    jan_df[['ProductID', 'Quantity']],
    feb_df[['ProductID', 'Total']]
],
join='inner',
ignore_index=True
)
```

```
Out [12]:      ProductID
0   T&G/CAN-97509
1   T&G/LEG-37777
2   T&G/PET-14209
3   T&G/TRA-20170
4   T&G/TRA-20170
5   C&P/NEW-62681
6   I&S/RUB-56368
7   M&T/7TH-34490
8   H&PC/MED-46454
9   T&G/100-26579
```

Now the output of `concat` contains just the one column that appears in each of the input `DataFrame` variables (i.e., the '`ProductID`' column).

Column-wise concatenation

You can use `concat` to combine `DataFrame` variables column-wise as well. To concatenate the three monthly `DataFrame` variables column-wise, you can pass the familiar `axis='columns'` argument to `concat`:

```
In [13]: pd.concat([jan_df, feb_df, mar_df], axis='columns')
```

```
Out [13]:      ProductID  Quantity   Total ...      ProductID  Quantity   Total
0  T&G/CAN-97509        14  281.54 ...  MI/VIC-46664        6  134.34
1  T&G/LEG-37777         1    6.70 ...  MI/ARC-23043        1   25.65
2  T&G/PET-14209         5   58.35 ...  MI/AKG-35546        2   17.96
3  T&G/TRA-20170         6   80.76 ...  CP&A/LE-28028        9   74.97
4  T&G/TRA-20170         6   80.76 ...  K&D/STA-02514       80  1436.80

[5 rows x 9 columns]
```

The result now is a `DataFrame` with nine columns (and with triplicate column names). While this is not a particularly useful table, in some cases column-wise concatenation might be the solution you need, which is why I'm mentioning it here.

As with row-wise concatenation, column-wise concatenation results in a union of all input row labels. For example, if you concatenate the top three rows of `jan_df` with the bottom three rows of `mar_df` you'll get:

```
In [14]: pd.concat([jan_df.head(3), mar_df.tail(3)], axis='columns')
```

```
Out [14]:      ProductID  Quantity   Total      ProductID  Quantity   Total
0  T&G/CAN-97509     14.0  281.54        NaN        NaN        NaN
1  T&G/LEG-37777      1.0    6.70        NaN        NaN        NaN
2  T&G/PET-14209      5.0   58.35  MI/AKG-35546      2.0   17.96
3          NaN        NaN        NaN  CP&A/LE-28028      9.0   74.97
4          NaN        NaN        NaN  K&D/STA-02514     80.0  1436.80
```

The result above contains a union of all row labels and `NaN` values where there is a mismatch. The only row label that appears in both input `DataFrame` variables is 2 — which is why the middle row has values in all columns. As with row-wise concatenation, you can use the `join='inner'` keyword argument to keep only the row labels that are common to all inputs:

```
In [15]: pd.concat([jan_df.head(3), mar_df.tail(3)], axis='columns', join='inner')
```

```
Out [15]:      ProductID  Quantity   Total      ProductID  Quantity   Total
2  T&G/PET-14209      5   58.35  MI/AKG-35546      2   17.96
```

The lesson here is that using `concat` is somewhat different from copy-pasting rows in a spreadsheet because `concat` also aligns its input `DataFrame` objects on their common row or column labels.

Appending rows to a DataFrame

Instead of using `concat`, you can also use the `append` method to add rows or columns to an existing `DataFrame`. For example, to add the rows from `feb_df` to `jan_df`, you can use:

```
In [16]: jan_df.append(feb_df)
```

```
Out [16]:      ProductID  Quantity   Total
0    T&G/CAN-97509       14  281.54
1    T&G/LEG-37777        1   6.70
2    T&G/PET-14209       5  58.35
3    T&G/TRA-20170       6  80.76
4    T&G/TRA-20170       6  80.76
0    C&P/NEW-62681        7  17.08
1    I&S/RUB-56368        2   8.70
2    M&T/7TH-34490       3  13.98
3    H&PC/MED-46454       14  90.02
4    T&G/100-26579        1  12.69
```

The `append` method does not modify the calling `DataFrame` *in-place*² but rather returns a new `DataFrame` with a concatenation of rows — it is just a shortcut for `concat`. If you want to update `jan_df` with rows from `feb_df`, you need to assign the output of `append` back to `jan_df`. Or if you want to add rows from both `feb_df` and `mar_df` to `jan_df`, you can use:

```
In [17]: jan_df.append([feb_df, mar_df], ignore_index=True)
```

```
Out [17]:      ProductID  Quantity   Total
0    T&G/CAN-97509       14  281.54
1    T&G/LEG-37777        1   6.70
2    T&G/PET-14209       5  58.35
3    T&G/TRA-20170       6  80.76
4    T&G/TRA-20170       6  80.76
5    C&P/NEW-62681        7  17.08
6    I&S/RUB-56368        2   8.70
7    M&T/7TH-34490       3  13.98
8    H&PC/MED-46454       14  90.02
9    T&G/100-26579        1  12.69
10   MI/VIC-46664         6 134.34
11   MI/ARC-23043         1  25.65
12   MI/AKG-35546         2  17.96
13   CP&A/LE-28028         9  74.97
14   K&D/STA-02514        80 1436.80
```

2: Unlike the `append` method available on Python lists.

Notice the `ignore_index` keyword argument which is also available when using `append` (but the `join` keyword argument, available with the `concat` function, is not). If you combine `DataFrame` objects that have different columns using `append`, as with `concat`, the result will have a union of all input columns filled with `NaN` values where the columns don't match.

The `append` method is particularly useful when you want to add a few rows to an existing `DataFrame`. For instance, you can add a subtotal row to `feb_df` with:

```
In [18]: feb_df.append([{
    'ProductID': 'Subtotal',
    'Quantity': feb_df['Quantity'].sum(),
    'Total': feb_df['Total'].sum()
}], ignore_index=True)
```

```
Out [18]:   ProductID  Quantity  Total
0  C&P/NEW-62681        7  17.08
1  I&S/RUB-56368        2   8.70
2  M&T/7TH-34490        3  13.98
3  H&PC/MED-46454       14  90.02
4  T&G/100-26579        1  12.69
5      Subtotal         27 142.47
```

You can create rows using Python dictionaries (with column names as keys) and pass them in a list to `append` — pandas will transform the dictionaries into `Series` objects (i.e., `DataFrame` rows). Note that if you use `append` this way, you must specify `ignore_index=True`.

However, using `concat` is faster and more flexible (i.e., it accepts both the `axis` and the `join` keyword arguments, whereas `append` doesn't) especially when concatenating many large `DataFrames` or when combining `DataFrames` in a `for` loop — but `append` is there if you need it.

Summary

This chapter showed you how to combine `DataFrame` variables with the `concat` function. Using `concat` is similar to copy-pasting rows or columns from one spreadsheet to another. In addition to `concat`, pandas also enables you to join `DataFrame` objects on their common columns — let's see how.

Joining tables

When working with multiple related datasets, you will often want to combine them into a single `DataFrame`, based on the values in one or more of the columns they have in common. Consider the two tables below:

ProductID	Unit Price	Quantity	ProductID	Product Name	Brand
T&G/CAN-97509	20.11	14	T&G/LEG-37777	LEGO Ninja T...	LEGO
T&G/LEG-37777	6.70	1	T&G/TRA-20170	Transformers...	Transformers
T&G/PET-14209	11.67	5	T&G/PET-14209	Pete the Cat...	Merry Makers
T&G/TRA-20170	13.46	6	T&G/CAN-97509	Cannon Water...	Fun To Collect

Both tables have a '`ProductID`' column (and if you take a closer look, the same product ID values appear in both, but in a different order). If you want to combine the two tables into one `DataFrame`, matching rows from one table to rows in the other based on product IDs, simply concatenating columns won't help — you will have to *join* them. In Excel, you can express this type of conditional joining of tables using the `VLOOKUP` function.¹ In pandas, you can use the `merge` function.²

Table joins can be puzzling at first. If you get confused at any point in this chapter, it's because you're trying to understand something complicated. The mental workout is, I think, worth it: once joins click, you'll see how powerful and straightforward they are.

The next section briefly introduces how table joins work in general — you'll see how to use pandas's `merge` function right after.

1: Or its variants: `HLOOKUP` or `XLOOKUP`.

2: If you have used SQL before, the `merge` function is the pandas equivalent to SQL's `JOIN` operator. Even though the pandas function is called `merge`, I use the term "*join*" (instead of "*merge*") because it is commonly used in other domains for this type of table operation (e.g., it's used in SQL).

How joins work

Table joins take as input two tables with one or more columns in common (let's call them the left and right table) and produce a single output table. The way joins work changes slightly depending on whether you have duplicate values in the joining column (or columns) in each table or not. For now, let's take the two tables above as our first example: these tables can be joined on their common column (i.e., '`ProductID`') — and because both tables have unique product IDs in their respective '`ProductID`' columns (i.e., there are no duplicate product IDs in either table), joining them is an example of a *one-to-one* join.

One-to-one joins are straightforward: rows that have the same value in the join column are merged as one row in the output table. The first step of the join starts with finding two rows (one in each table) with the same product ID:

ProductID	Unit Price	Quantity	ProductID	Product Name	Brand
T&G/CAN-97509	20.11	14	T&G/LEG-37777	LEGO Ninja T...	LEGO
T&G/LEG-37777	6.70	1	T&G/TRA-20170	Transformers...	Transformers
T&G/PET-14209	11.67	5	T&G/PET-14209	Pete the Cat...	Merry Makers
T&G/TRA-20170	13.46	6	T&G/CAN-97509	Cannon Water...	Fun To Collect

After that, values from these two rows get copied as the first row in the output table — the value they have in common (i.e., 'ProductID') is copied just once. At the end of this first step, the output table looks like this:

ProductID	Unit Price	Quantity	Product Name	Brand
T&G/CAN-97509	20.11	14	Cannon Water Bomb Balloons	Fun To Collect

The next step in the join is identical: pandas goes to the following product ID in the left table, finds the corresponding row on the right, then copies a merged version of the two rows to the output table.

ProductID	Unit Price	Quantity	ProductID	Product Name	Brand
T&G/CAN-97509	20.11	14	T&G/LEG-37777	LEGO Ninja T...	LEGO
T&G/LEG-37777	6.70	1	T&G/TRA-20170	Transformers...	Transformers
T&G/PET-14209	11.67	5	T&G/PET-14209	Pete the Cat...	Merry Makers
T&G/TRA-20170	13.46	6	T&G/CAN-97509	Cannon Water...	Fun To Collect

ProductID	Unit Price	Quantity	Product Name	Brand
T&G/CAN-97509	20.11	14	Cannon Water Bomb Balloons	Fun To Collect
T&G/LEG-37777	6.70	1	LEGO Ninja Turtles Stealth	LEGO

And so the join continues for all remaining rows. The entire join operation and its resulting table are shown below:

Input table (left):			Input table (right):		
ProductID	Unit Price	Quantity	ProductID	Product Name	Brand
T&G/CAN-97509	20.11	14	T&G/LEG-37777	LEGO Ninja T...	LEGO
T&G/LEG-37777	6.70	1	T&G/TRA-20170	Transformers...	Transformers
T&G/PET-14209	11.67	5	T&G/PET-14209	Pete the Cat...	Merry Makers
T&G/TRA-20170	13.46	6	T&G/CAN-97509	Cannon Water...	Fun To Collect

Output table:					
ProductID	Unit Price	Quantity	Product Name	Brand	
T&G/CAN-97509	20.11	14	Cannon Water Bomb Balloons	Fun To Collect	
T&G/LEG-37777	6.70	1	LEGO Ninja Turtles Stealth	LEGO	
T&G/PET-14209	11.67	5	Pete the Cat and His Four...	Merry Makers	
T&G/TRA-20170	13.46	6	Transformers Age of Extin...	Transformers	

In a nutshell, this is what joins are all about: merging two tables on the values they have in common. However, there is one detail we still need to iron out: what happens when you have duplicate values in the join column?

Many-to-one joins

You might need to join two tables on a common column, but one of the tables has duplicate values in that column. Consider the following example:

ProductID	Unit Price	Quantity	ProductID	Product Name	Brand
T&G/GRE-17530	5.82	1	T&G/GRE-17530	Green Toys Tea Set	Green Toys
T&G/PAP-51200	38.89	6	T&G/PAP-51200	Papo Brachiosau...	Papo
T&G/PAP-51200	39.14	6			
T&G/GRE-17530	5.81	1			

The table on the left has duplicate values in its 'ProductID' column, whereas the table on the right has unique product IDs. Joining these two tables is an example of a *many-to-one* join.³

As before, the first step starts with finding rows in the two tables that share a common product ID value beginning with the first value in the 'ProductID' column of the left table. In this case, there are two rows in the left table and one row in the right — these rows are highlighted above.

3: If you swap the two tables, you get a *one-to-many* join, which works the same way.

As with the previous example, both left and right rows get merged and copied in the output table. However, the difference here is that *each* row on the left gets merged with its corresponding row on the right. After the first step, the output table looks like this:

ProductID	Unit Price	Quantity	Product Name	Brand
T&G/GRE-17530	5.82	1	Green Toys Tea Set	Green Toys
T&G/GRE-17530	5.81	1	Green Toys Tea Set	Green Toys

The output above has two rows because there are two rows in the left table where the product ID is T&G/GRE-17530, and each row on the left gets merged with its corresponding row on the right.

The next steps in the join are the same — the entire join and the resulting table are shown below:

Input table (left):			Input table (right):		
ProductID	Unit Price	Quantity	ProductID	Product Name	Brand
T&G/GRE-17530	5.82	1	T&G/GRE-17530	Green Toys Tea Set	Green Toys
T&G/PAP-51200	38.89	6	T&G/PAP-51200	Papo Brachiosau...	Papo
T&G/PAP-51200	39.14	6			
T&G/GRE-17530	5.81	1			

Output table:					
ProductID	Unit Price	Quantity	Product Name	Brand	
T&G/GRE-17530	5.82	1	Green Toys Tea Set	Green Toys	
T&G/GRE-17530	5.81	1	Green Toys Tea Set	Green Toys	
T&G/PAP-51200	38.89	6	Papo Brachiosaurus Toy Figure	Papo	
T&G/PAP-51200	39.14	6	Papo Brachiosaurus Toy Figure	Papo	

Notice that in the output table, '`ProductID`' values are grouped (i.e., duplicate product IDs are placed one after the other), which means the original order of rows, from both tables, is lost in the output table (this is a feature of joins).

Many-to-many joins

The other special case we need to take a look at is the *many-to-many* join. As you can probably tell, this join happens when you have duplicate values in the join column in both left and right tables — the example below illustrates this case:

ProductID	Unit Price	Quantity	ProductID	Review	Rating
T&G/TRA-20170	13.66	21	T&G/LEG-37777	Fun, affordable ...	4.0
T&G/LEG-37777	6.66	1	T&G/LEG-37777	Priced right	5.0
T&G/TRA-20170	13.66	1	T&G/TRA-20170	One Cool Autobot	5.0
T&G/TRA-20170	13.48	4	T&G/TRA-20170	Pretty-well made...	3.0
T&G/LEG-37777	6.69	1	T&G/LEG-37777	A Fantastic Set!...	4.0

As with the previous cases, joining these two tables starts with finding rows in each table that share a common product ID (e.g., the highlighted rows above) and merging them. After merging the first set of rows, the output table looks like this:

ProductID	Unit Price	Quantity	Summary	Rating
T&G/TRA-20170	13.66	21	One Cool Autobot	5.0
T&G/TRA-20170	13.66	21	Pretty-well made but Frust...	3.0
T&G/TRA-20170	13.66	1	One Cool Autobot	5.0
T&G/TRA-20170	13.66	1	Pretty-well made but Frust...	3.0
T&G/TRA-20170	13.48	4	One Cool Autobot	5.0
T&G/TRA-20170	13.48	4	Pretty-well made but Frust...	3.0

Here, the first row in the left table gets merged with *each* of its corresponding rows in the right table. (i.e., rows three and four). There are two more rows on the left with the same product ID (i.e., the third and fourth row), and they both get merged with the same rows from the right table as well.

The following steps in the join are identical — the entire operation is illustrated below:

Input table (left):			Input table (right):		
ProductID	Unit Price	Quantity	ProductID	Review	Rating
T&G/TRA-20170	13.66	21	T&G/LEG-37777	Fun, affordable ...	4.0
T&G/LEG-37777	6.66	1	T&G/LEG-37777	Priced right	5.0
T&G/TRA-20170	13.66	1	T&G/TRA-20170	One Cool Autobot	5.0
T&G/TRA-20170	13.48	4	T&G/TRA-20170	Pretty-well made...	3.0
T&G/LEG-37777	6.69	1	T&G/LEG-37777	A Fantastic Set!...	4.0

Output table:					
ProductID	Unit Price	Quantity		Summary	Rating
T&G/TRA-20170	13.66	21		One Cool Autobot	5.0
T&G/TRA-20170	13.66	21	Pretty-well made but Frustrating		3.0
T&G/TRA-20170	13.66	1		One Cool Autobot	5.0
T&G/TRA-20170	13.66	1	Pretty-well made but Frustrating		3.0
T&G/TRA-20170	13.48	4		One Cool Autobot	5.0
T&G/TRA-20170	13.48	4	Pretty-well made but Frustrating		3.0
T&G/LEG-37777	6.66	1	Fun, affordable Lego TMNT set		4.0
T&G/LEG-37777	6.66	1	Priced right		5.0
T&G/LEG-37777	6.66	1	A Fantastic Set!! And the Turtle minifig ...		4.0
T&G/LEG-37777	6.69	1	Fun, affordable Lego TMNT set		4.0
T&G/LEG-37777	6.69	1	Priced right		5.0
T&G/LEG-37777	6.69	1	A Fantastic Set!! And the Turtle minifig ...		4.0

As with many-to-one joins, notice that product IDs get grouped in the output table and that row order from the input tables is lost. Many-to-many joins usually create a lot of rows in the output table, which is why it is a good idea to use them with small input tables (or when you really need to create a lot of rows).

All the examples above illustrate how the join operation works in general — pandas or any other tool that lets you join tables implements them as described above. Let's see how you make them happen with code next.

Joins in pandas

For the code examples in this section, you'll need to use two datasets: the sales data from “Q1Sales.xlsx” that you're now familiar with and the products dataset from “products.csv” that I briefly introduced at the beginning of part two. To read the two datasets, run:

```
In [1]: import pandas as pd

ledger_df = pd.read_excel('Q1Sales.xlsx')
products_df = pd.read_csv('products.csv')
```

From these datasets, let's create two smaller DataFrame variables that will help illustrate one-to-one joins in pandas:

```
In [2]: left_df = ledger_df.head(5) 1
left_df = left_df[['ProductID', 'Unit Price', 'Quantity']] 2
left_df = left_df.drop_duplicates(subset=['ProductID']) 3
right_df = products_df[products_df['ProductID'].isin(left_df['ProductID'].unique())] 4
right_df = right_df[['ProductID', 'Product Name', 'Brand']] 5
right_df = right_df[['ProductID', 'Product Name', 'Brand']] 6
```

The code above should be familiar:

- ▶ **Line 1**: you select the top 5 rows of the ledger DataFrame and assign them to a new variable called `left_df`.
- ▶ **Line 2**: you select three of the columns in the `left_df` and reassign the selection to `left_df`.
- ▶ **Line 3**: you remove duplicate rows based on values in the `'ProductID'` column.
- ▶ **Line 5**: from the products dataset, you select only those products that appear in `left_df`, based on product IDs, and assign those rows to a new variable called `right_df`.
- ▶ **Line 6**: you select three columns from `right_df` and assign the selection back to `right_df`.

```
In [3]: left_df
```

```
Out [3]:   ProductID  Unit Price  Quantity
0  T&G/CAN-97509      20.11       14
1  T&G/LEG-37777       6.70        1
2  T&G/PET-14209      11.67        5
3  T&G/TRA-20170      13.46        6
```

```
In [4]: right_df
```

```
Out [4]:          ProductID           Product Name      Brand
7606  T&G/LEG-37777  LEGO Ninja Turtles...      LEGO
7813  T&G/TRA-20170  Transformers Age o...  Transformers
9143  T&G/PET-14209  Pete the Cat and H...  Merry Makers
10190 T&G/CAN-97509  Cannon Water Bomb ...  Fun To Collect
```

The two DataFrame variables have the same data as the example tables you saw earlier. They have a common column (i.e., `'ProductID'`) with unique values in both `left_df` and `right_df`. To join them on their common column, you can use pandas's `merge` function:

```
In [5]: pd.merge(left_df, right_df, on='ProductID')
```

```
Out [5]:          ProductID  Unit Price  Quantity           Product Name      Brand
0  T&G/CAN-97509      20.11       14  Cannon Water Bomb Ba...  Fun To Collect
1  T&G/LEG-37777       6.70        1  LEGO Ninja Turtles S...      LEGO
2  T&G/PET-14209      11.67        5  Pete the Cat and His...  Merry Makers
3  T&G/TRA-20170      13.46        6  Transformers Age of ...  Transformers
```

The first two arguments passed to `merge` are the `DataFrame` variables you want to join. The `on` keyword argument specifies what column to use for joining the two tables.⁴ The output is a new `DataFrame`.⁵

The `merge` function maps each row in `left_df` to a row in `right_df`, based on their corresponding `'ProductID'` values (as illustrated in the previous section). It then merges the two datasets into a new `DataFrame`, which has the same columns as the left and right tables used as input. This is a join in `pandas`: even though the join operation can be complex, the `pandas` code to run it is simple.

Whether your join is *one-to-one* or *many-to-many*, you still use `pandas`'s `merge` function the same way (i.e., as in the code example above). The columns you use for joining and whether they have duplicate values influence the output table, not how you use the `merge` function.

4: You don't have to specify a column, but it's a good idea to be explicit about the join column with the `on` keyword argument. If you don't use the `on` argument, the two tables are joined on all the columns they have in common (i.e., all columns with the same name in both tables).

5: The original row labels from either `left_df` or `right_df` are not kept in the output `DataFrame`.

Overthinking: Validating joins

One common issue when joining tables is assuming that the joining column values are unique (in one or both input tables) when they aren't. This typically leads to a many-to-many join, which creates a lot of rows and takes a long time to run — and for large datasets, it can freeze your computer because it uses up all of its memory.

To prevent unwanted joins, you can use the `validate` keyword argument with `merge` to tell `pandas` what kind of join you are expecting. When you use `validate`, `pandas` checks key uniqueness in both input tables before combining them, which can save you a lot of time and frustration. If keys are not as unique as you thought they were, `pandas` prompts you with an error message.

As an example, let's duplicate the last row in `left_df` and `right_df`:

```
In [6]: left_df = left_df.append(left_df.iloc[-1])
right_df = right_df.append(right_df.iloc[-1])
```

```
In [7]: left_df
```

```
Out[7]:    ProductID  Unit Price  Quantity
0  T&G/CAN-97509      20.11       14
1  T&G/LEG-37777       6.70        1
2  T&G/PET-14209      11.67        5
3  T&G/TRA-20170      13.46        6
3  T&G/TRA-20170      13.46        6
```

```
In [8]: right_df
```

Out [8]:

	ProductID	Product Name	Brand
7606	T&G/LEG-37777	LEGO Ninja Turtles Stealth...	LEGO
7813	T&G/TRA-20170	Transformers Age of Extinc...	Transformers
9143	T&G/PET-14209	Pete the Cat and His Four ...	Merry Makers
10190	T&G/CAN-97509	Cannon Water Bomb Balloons...	Fun To Collect
10190	T&G/CAN-97509	Cannon Water Bomb Balloons...	Fun To Collect

Now both tables have duplicate values in their common column.

If you use `merge` with the `validate` keyword argument set to '`one_to_one`', you will get an error:

In [9]: `pd.merge(left_df, right_df, on='ProductID', validate='one_to_one')`

Out [9]:

```
MergeError                                         Traceback (most recent call last)
<ipython-input-103-82d81cf2304c> in <module>
    ---> 1 pd.merge(left_df, right_df, on='ProductID', validate='one_to_one')
...
MergeError: Merge keys are not unique in either left or right dataset;
not a one-to-one merge
```

The error message tells you exactly that keys are not unique (in either dataset) and that the merge is not one-to-one. You can set the value of `validate` to one of:

- ▶ '`one_to_one`' to check if join values are unique in both left and right tables;
- ▶ '`many_to_one`' to check if join values are unique in the right table;
- ▶ '`one_to_many`' to check if join values are unique in the left table;

Joins on large tables can take a long time to run, which is why using the `validate` argument is a good idea: it can stop unwanted joins early (i.e., joins on values that you think are unique but aren't).

Inner, outer, left, and right joins

The examples we looked at previously had the same values in the '`ProductID`' column, with or without duplicates, in both left and right tables. This perfect alignment is rarely present in real-world tables — so what happens if your tables have different values in the column you want to join on? Consider the tables below:

ProductID	Unit Price	Quantity	ProductID	Product Name	Brand
T&G/THE-82687	5.36	6	T&G/PLA-85805	Playskool Mrs. P...	Mr Potato Head
T&G/PLA-85805	3.31	1	T&G/THO-09600	Thomas the Train...	Fisher-Price
T&G/DIS-51236	14.47	12	T&G/PLA-29969	Plan Toy Pull-Al...	Plan Toys
T&G/LEG-60816	21.40	1	T&G/LEG-60816	LEGO Star Wars M...	LEGO

These two tables both have a '`ProductID`' column. However, they don't have the same set of values in that column: the product IDs in the yellow rows above appear in both tables, whereas the ones in blue rows appear only in the left table, and the ones in red rows appear only in the right table.

By default, when you join these tables on the '`ProductID`' column, the output will contain only those product IDs that are shared between the two tables:

ProductID	Unit Price	Quantity	Product Name	Brand
T&G/PLA-85805	3.31	1	Playskool Mrs. Potato Head	Mr Potato Head
T&G/LEG-60816	21.40	1	LEGO Star Wars Mandalorian Battle ...	LEGO

This type of join is called an **inner** join.

However, in `pandas` you can change this default behavior with the `how` keyword argument, and setting it to either '`left`', '`right`', or '`outer`' when using `merge`. Before we look at some code that does that, let's quickly see what the other types of joins look like.

A **left join**⁶ keeps all rows from the *left* table in the output. However, because not all left rows have a corresponding row in the right table, only a subset of them get merged with values from the right table, whereas rows that don't have a match get filled with `Nan`s. Left joining the two tables above produces the following output:

6: Sometimes called a *left outer join*.

ProductID	Unit Price	Quantity	Product Name	Brand
T&G/THE-82687	5.36	6	NaN	NaN
T&G/PLA-85805	3.31	1	Playskool Mrs. Potato Head	Mr Potato Head
T&G/DIS-51236	14.47	12	NaN	NaN
T&G/LEG-60816	21.40	1	LEGO Star Wars Mandalorian Battle ...	LEGO

A **right join**⁷ keeps all rows from the *right* table in the output. Like before, because not all rows have a corresponding row in the left table, only a subset of them get merged, whereas rows that don't have a match get filled with `Nan`s. Right joining the two tables produces the following output:

7: Sometimes called a *right outer join*.

ProductID	Unit Price	Quantity	Product Name	Brand
T&G/PLA-85805	3.31	1.0	Playskool Mrs. Potato Head	Mr Potato Head
T&G/THO-09600	NaN	NaN	Thomas the Train: My First Thomas	Fisher-Price
T&G/PLA-29969	NaN	NaN	Plan Toy Pull-Along Snail	Plan Toys
T&G/LEG-60816	21.40	1.0	LEGO Star Wars Mandalorian Battle ...	LEGO

An **outer join**⁸ keeps all rows from both *left* and *right* tables

8: Sometimes called a *full outer join*.

in the output. Like left and right joins, only those rows with a corresponding row in the other table get merged, whereas missing values get filled with NaNs. The output of an outer join on the two example tables is shown below:

ProductID	Unit Price	Quantity	Product Name	Brand
T&G/THE-82687	5.36	6.0	NaN	NaN
T&G/PLA-85805	3.31	1.0	Playskool Mrs. Potato Head	Mr Potato Head
T&G/DIS-51236	14.47	12.0	NaN	NaN
T&G/LEG-60816	21.40	1.0	LEGO Star Wars Mandalorian Battle ...	LEGO
T&G/THO-09600	NaN	NaN	Thomas the Train: My First Thomas	Fisher-Price
T&G/PLA-29969	NaN	NaN	Plan Toy Pull-Along Snail	Plan Toys

In all the examples above, notice that the order of rows in the output depends on the type of join (e.g., the output of a left join has product IDs in the same order as the left table).

Let's get back to `pandas` and see how to change the default join behavior. First, let's create two `DataFrame` variables to use as an example:

```
In [10]: left_ids = ['T&G/LEG-60816', 'T&G/PLA-85805', 'T&G/DIS-51236', 'T&G/THE-82687']
right_ids = ['T&G/THO-09600', 'T&G/PLA-29969', 'T&G/LEG-60816', 'T&G/PLA-85805']

left_df = ledger_df[ledger_df['ProductID'].isin(left_ids)]
left_df = left_df[['ProductID', 'Unit Price', 'Quantity']]

right_df = products_df[products_df['ProductID'].isin(right_ids)]
right_df = right_df[['ProductID', 'Product Name', 'Brand']]
```

```
In [11]: left_df
```

```
Out [11]:   ProductID  Unit Price  Quantity
3687    T&G/THE-82687      5.36        6
3938    T&G/PLA-85805      3.31        1
11413   T&G/DIS-51236     14.47       12
13021   T&G/LEG-60816     21.40        1
```

```
In [12]: right_df
```

```
Out [12]:   ProductID          Product Name      Brand
7528    T&G/PLA-85805  Playskool Mrs. Potato Head  Mr Potato Head
11225   T&G/THO-09600  Thomas the Train: My First Thomas  Fisher-Price
13943   T&G/PLA-29969  Plan Toy Pull-Along Snail  Plan Toys
14628   T&G/LEG-60816  LEGO Star Wars Mandalorian Battle Pa...  LEGO
```

By design, these two `DataFrames` are identical to the two example tables at the beginning of this section — and they have only two common values in the `'ProductID'` column (i.e., `'T&G/PLA-85805'` and `'T&G/LEG-60816'`). You can join these tables using the same `merge` function you used before:

```
In [13]: pd.merge(left_df, right_df, on='ProductID')
```

```
Out [13]:   ProductID  Unit Price  Quantity          Product Name      Brand
0  T&G/LEG-73192      2.37       12  LEGO Technic 42006 Excavator      LEGO
1  T&G/PLA-29969      6.15        1    Plan Toy Pull-Along Snail  Plan Toys
```

As mentioned earlier, by default, `merge` performs an inner join between tables and discards all the rows that don't have a corresponding row in the other table.

However, discarding rows is not always what you need. You can use the `how` keyword argument to tell `merge` to use one of the joining behaviors illustrated above:

```
In [14]: pd.merge(left_df, right_df, on='ProductID', how='left')
```

```
Out [14]:   ProductID  Unit Price  Quantity          Product Name      Brand
0  T&G/MY-14365      11.84       2                  NaN      NaN
1  T&G/LEG-73192      2.37       12  LEGO Technic 42006 Excavator      LEGO
2  T&G/PLA-29969      6.15        1    Plan Toy Pull-Along Snail  Plan Toys
3  T&G/PAP-59850      22.85       1                  NaN      NaN
```

```
In [15]: pd.merge(left_df, right_df, on='ProductID', how='right')
```

```
Out [15]:   ProductID  Unit Price  Quantity          Product Name      Brand
0  T&G/PLA-85805      NaN       NaN  Playskool Mrs. Potato Head  Mr Potato Head
1  T&G/LEG-73192      2.37      12.0  LEGO Technic 42006 Excavator      LEGO
2  T&G/THO-09600      NaN       NaN  Thomas the Train: My First Thomas  Fisher-Price
3  T&G/PLA-29969      6.15      1.0    Plan Toy Pull-Along Snail  Plan Toys
```

```
In [16]: pd.merge(left_df, right_df, on='ProductID', how='outer')
```

```
Out [16]:   ProductID  Unit Price  Quantity          Product Name      Brand
0  T&G/MY-14365      11.84      2.0                  NaN      NaN
1  T&G/LEG-73192      2.37      12.0  LEGO Technic 42006 Excavator      LEGO
2  T&G/PLA-29969      6.15      1.0    Plan Toy Pull-Along Snail  Plan Toys
3  T&G/PAP-59850      22.85      1.0                  NaN      NaN
4  T&G/PLA-85805      NaN       NaN  Playskool Mrs. Potato Head  Mr Potato Head
5  T&G/THO-09600      NaN       NaN  Thomas the Train: My First Thomas  Fisher-Price
```

It might seem like all the details discussed in this chapter are fairly specific (and not that common when working with real data). You'd be surprised how often joins — in their various configurations — can solve real problems, which is why I think learning what they do is worth your time.

Overthinking: The source of each row

In some cases, when using a left, right, or outer join, you might want to know if a given row appears in both, left or right tables. You can use the `indicator` keyword argument with `merge` to add a new column in the output table that tells you where each row is originally from:

```
In [17]: pd.merge(left_df, right_df, on='ProductID', how='outer', indicator='Source')
```

	ProductID	Unit Price	Quantity	Product Name	Brand	Source
0	T&G/THE-82687	5.36	6.0	NaN	NaN	left_only
1	T&G/PLA-85805	3.31	1.0	Playskool Mrs. Potato Head	Mr Potato Head	both
2	T&G/DIS-51236	14.47	12.0	NaN	NaN	left_only
3	T&G/LEG-60816	21.40	1.0	LEGO Star Wars Manda...	LEGO	both
4	T&G/THO-09600	NaN	NaN	Thomas the Train: My...	Fisher-Price	right_only
5	T&G/PLA-29969	NaN	NaN	Plan Toy Pull-Along ...	Plan Toys	right_only

As with the `validate` keyword argument, the indicator column⁹ can help you double-check that the assumptions you had before the merge are correct. For instance, you can assign the output of `merge` above to a new variable and check how many rows originated from each of the two input variables:

```
In [18]: merged_df = pd.merge(left_df, right_df, on='ProductID', how='outer', indicator='Source')

merged_df['Source'].value_counts()
```

```
Out [18]: both      2
right_only    2
left_only     2
Name: Source, dtype: int64
```

By design, our two tables had two product IDs that appeared in both, two product IDs that appeared only in `left_df`, and two that appeared only in `right_df` — the merge output seems correct.

More joining options

Joins can be performed on more than one column and on columns that have different names in the two tables you want to merge. Let's take the bottom five rows of `ledger_df` and separate them into two `DataFrame` variables for this example:

```
In [19]: left_df = ledger_df[['ProductID', 'Channel', 'Unit Price']].tail(5)          1
right_df = ledger_df[['ProductID', 'Channel', 'Deadline']].tail(5)                   2
```

```
In [20]: left_df
```

```
Out [20]:   ProductID      Channel  Unit Price
14049  E/AC-63975  Bullseye      28.72
14050  E/CIS-74992  Bullseye      33.39
14051  E/PHI-08100  Understock.com      4.18
14052  E/POL-61164  iBay.com       4.78
14053  E/SIR-83381  Understock.com      33.16
```

```
In [21]: right_df
```

```
Out [21]:   ProductID      Channel      Deadline
14049  E/AC-63975  Bullseye  February 23 2020
14050  E/CIS-74992  Bullseye  January 21 2020
14051  E/PHI-08100  Understock.com  March 22 2020
14052  E/POL-61164  iBay.com    June 25 2020
14053  E/SIR-83381  Understock.com  February 01 2020
```

9: You can pass any string value to the `indicator` argument, which will then be used as the indicator column name. I used '`Source`' here, but any string value is valid.

These two DataFrame objects share *two* columns: '`ProductID`' and '`Channel`'. You can merge tables on values from multiple columns by using the `merge` function, just as we did in the previous sections, and passing a list of column names to the `on` keyword argument instead of a single column name:

```
In [22]: pd.merge(left_df, right_df, on=['ProductID', 'Channel'])
```

```
Out [22]:   ProductID      Channel  Unit Price      Deadline
0    E/AC-63975    Bullseye    28.72  February 23 2020
1    E/CIS-74992    Bullseye    33.39   January 21 2020
2    E/PHI-08100  Understock.com    4.18    March 22 2020
3    E/POL-61164     iBay.com    4.78     June 25 2020
4    E/SIR-83381  Understock.com   33.16  February 01 2020
```

The mechanics of joining on multiple columns is the same as joining on one column, but in this case, each *combination of values* in the two columns is used to match rows from one table to the other. If you wanted to, you could create a separate column in each table by combining values from the '`ProductID`' and '`Channel`' column and using that as the join column — but you would get the same result with more effort. All the other keyword arguments you can use with `merge` (i.e., `validate`, `how`, `indicator`) work the same way as before.

In addition to joining on multiple columns, you can also join tables on columns that don't have the same name in each input table (although they do need to share some common values in those columns for the join to work). For example, let's change the column names in `right_df` to lowercase:

```
In [23]: right_df.columns = ['productid', 'channel', 'deadline']
```

```
right_df
```

```
Out [23]:   productid      channel      deadline
14049    E/AC-63975    Bullseye  February 23 2020
14050    E/CIS-74992    Bullseye  January 21 2020
14051    E/PHI-08100  Understock.com  March 22 2020
14052    E/POL-61164     iBay.com    June 25 2020
14053    E/SIR-83381  Understock.com  February 01 2020
```

You get the same DataFrame as before, but with different column names. If you want to run the join above, you can rename the columns back to their original names (which wouldn't really help with this example), or you can specify the left and right columns to use for the join separately, using:

```
In [24]: pd.merge(
    left_df, right_df,
    left_on=['ProductID', 'Channel'],
    right_on=['productid', 'channel']
)
```

	ProductID	Channel	Unit Price	productid	channel	deadline
0	E/AC-63975	Bullseye	28.72	E/AC-63975	Bullseye	February 23 2020
1	E/CIS-74992	Bullseye	33.39	E/CIS-74992	Bullseye	January 21 2020
2	E/PHI-08100	Understock.com	4.18	E/PHI-08100	Understock.com	March 22 2020
3	E/POL-61164	iBay.com	4.78	E/POL-61164	iBay.com	June 25 2020
4	E/SIR-83381	Understock.com	33.16	E/SIR-83381	Understock.com	February 01 2020

However, when you join tables on differently named columns, both the left and right join columns are kept in the output, which is not particularly useful (at least not in this case). You can remove them after the join using `drop`:

```
In [25]: pd.merge(
    left_df, right_df,
    left_on=['ProductID', 'Channel'],
    right_on=['productid', 'channel']
).drop(['productid', 'channel'], axis='columns')
```

	ProductID	Channel	Unit Price	deadline
0	E/AC-63975	Bullseye	28.72	February 23 2020
1	E/CIS-74992	Bullseye	33.39	January 21 2020
2	E/PHI-08100	Understock.com	4.18	March 22 2020
3	E/POL-61164	iBay.com	4.78	June 25 2020
4	E/SIR-83381	Understock.com	33.16	February 01 2020

Summary

Working with data often requires you to leverage connections between datasets. To connect tables on their shared values, you have to join them. This chapter showed you how table joins work in general and how to make them happen using `pandas` code.

It might seem like joins are specialized tools, not that useful when dealing with accounting data. However, you'd be surprised how often joins can solve real-world problems. One problem they can solve is filling in missing product names in `ledger_df`, which is what we look at in the following project chapter.

Project: Filling missing product names in the sales data

27

The sales data you've been working with have a specific problem that table joins can help with: the 'Product Name' column has a lot of missing values. This quick project chapter shows you how to fill in missing product names in "Q1Sales.xlsx" with values from the "products.csv" file.

To get this project started, open the project notebook (which should be in your *Python for Accounting* workspace) and `import pandas`. Load the sales and products data by running:

```
In [1]: import pandas as pd  
sales_df = pd.concat(pd.read_excel('Q1Sales.xlsx', sheet_name=None), ignore_index=True)  
products_df = pd.read_csv('products.csv')
```

Line 3 above may seem strange, but it uses pandas's concat and read_excel functions, both of which you've seen before. Passing `sheet_name=None` to read_excel makes the function read data from *all sheets* in an Excel file. However, when you use `sheet_name=None`, read_excel no longer returns a single DataFrame, but a Python dictionary mapping sheet names to DataFrame objects. If you run the function in a separate cell, you'll see this dictionary:

```
In [2]: pd.read_excel('Q1Sales.xlsx', sheet_name=None)
```

	InvoiceNo	Channel	Product Name	Unit Price	Quantity
'January': Total					
0	1532	Shoppe.com	Cannon Water Bom...	20.11	14 281.54
1	1533	Walcart	LEGO Ninja Turtl...	6.70	1 6.70
2	1534	Bullseye	Nan ...	11.67	5 58.35
3	1535	Bullseye	Transformers Age...	13.46	6 80.76
4	1535	Bullseye	Transformers Age...	13.46	6 80.76
...
14049	15581	Bullseye	AC Adapter/Power...	28.72	8 229.76
14050	15582	Bullseye	Cisco Systems Gi...	33.39	1 33.39
14051	15583	Understock.com	Philips AJ3116M/...	4.18	1 4.18
14052	15584	iBay.com	Nan ...	4.78	25 119.50
14053	15585	Understock.com	Sirius Satellite...	33.16	2 66.32
...
'March': Total					
0	29486	Walcart	Vic Firth Americ...	22.39	6 134.34
1	29487	Walcart	Archives Spiral ...	25.65	1 25.65
2	29488	Bullseye	AKG WMS40 Mini D...	8.98	2 17.96
3	29489	Shoppe.com	LE Blue Case for...	8.33	9 74.97
4	29490	Understock.com	STARFISH Cookie ...	17.96	80 1436.80
...
9749	39235	iBay.com	Nature's Bounty ...	5.55	2 11.10

9750	39216	Shoppe.com	Funko Wonder Wom...	...	28.56	1	28.56
9751	39219	Shoppe.com	MONO GS1 GS1-BTY...	...	3.33	1	3.33
9752	39238	Shoppe.com		Nan	34.76	10	347.60
9753	39239	Understock.com	3 Collapsible Bo...	...	6.39	15	95.85

[9754 rows x 12 columns]}

The `concat` function works with dictionaries (like the one above) as well as lists of `DataFrame` objects. If you pass the output of `read_excel` above to `concat`, you get a single `DataFrame` with all the data in your Excel file. The `ignore_index` keyword argument makes `concat` discard row labels so that labels in the output `DataFrame` are consecutive numbers. You'll often need to read all sheets in an Excel file and keep their data in a single `DataFrame`; the shortest way to do that is with code like the one above.

You can now check both `DataFrame` variables for missing values:

In [3]: `sales_df.isna().sum()`

Out [3]:

InvoiceNo	0
Channel	0
Product Name	4566
ProductID	0
Account	0
AccountNo	0
Date	0
Deadline	0
Currency	0
Unit Price	0
Quantity	0
Total	0
dtype: int64	

As promised, the '`Product Name`' column contains over 4000 missing values. On the other hand, the products data seem complete:

In [4]: `products_df.isna().sum()`

Out [4]:

ProductID	0
Product Name	0
Brand	0
Category	0
dtype: int64	

You'll join the two datasets on their common '`ProductID`' column. However, you only need '`Product Name`' values from `products_df`; let's discard its '`Brand`' and '`Category`' columns:

In [5]: `products_df = products_df[['ProductID', 'Product Name']]`

The products dataset doesn't have any missing values, but that doesn't mean all the products in the sales data appear in the products data. Let's quickly check if all product IDs in `sales_df` have a corresponding ID in `products_df`:

```
In [6]: sales_df['ProductID'].isin(products_df['ProductID'])
```

```
Out [6]: 0      True
1      True
2      True
3      True
4      True
...
37703  True
37704  True
37705  True
37706  True
37707  True
Name: ProductID, Length: 37708, dtype: bool
```

The code above runs the product ID check, but it returns a long Series, with a boolean value for each row in sales_df. To quickly check whether all values in this Series are True, you can use the all Series method:

```
In [7]: sales_df['ProductID'].isin(products_df['ProductID']).all()
```

```
Out [7]: True
```

This tells you that all product IDs in sales_df have a corresponding ID in products_df. Now you can go ahead with joining the two datasets:

```
In [8]: pd.merge(
    sales_df,
    products_df,
    on='ProductID',
    suffixes=['-Sales', '-Products'],
    validate='many_to_one'
)
```

```
Out [8]:   InvoiceNo     Channel  Product Name-Sales ...   Total  Product Name-Products
0        1532  Shoppe.com  Cannon Water Bom... ...  281.54  Cannon Water Bom...
1        1949       Walcart           NaN ...   60.72  Cannon Water Bom...
2        5401  Understock.com  Cannon Water Bom... ... 101.00  Cannon Water Bom...
3        8601  Understock.com  Cannon Water Bom... ... 161.60  Cannon Water Bom...
4        9860  Understock.com  Cannon Water Bom... ... 101.00  Cannon Water Bom...
...
37703     ...          ...  New Waterproof S... ...   42.89  New Waterproof S...
37704     38956  Understock.com  Violinsmart 3/4 ... ... 52.12  Violinsmart 3/4 ...
37705     39053  Understock.com  Violinsmart 3/4 ... ... 52.12  Violinsmart 3/4 ...
37706     39030  Understock.com  Violinsmart 3/4 ... ... 52.12  Violinsmart 3/4 ...
37707     39038  Understock.com  Violinsmart 3/4 ... ... 52.12  Violinsmart 3/4 ...
37708     39045  Understock.com  Violinsmart 3/4 ... ... 52.12  Violinsmart 3/4 ...

[37708 rows x 13 columns]
```

The example above uses pandas's merge function to join the two datasets on their common 'ProductID' column. Because both tables have a 'Product Name' column that isn't used as a joining column in the merge operation, their join will have two 'Product Name'

columns (one from the left table, another from the right one). The `suffixes` keyword argument above tells pandas to add a different suffix to each '`Product Name`' column and make the joined table easier to understand. If you don't pass custom suffixes as I did above, pandas uses '`x`' and '`y`' as default suffixes ('`Product Name-Sales`' seems easier to understand than '`Product Namex`').

In the code above, you also set the `validate='many_to_one'` keyword argument that triggers an error in case product IDs in `products_df` (i.e., in the right table) are not unique (they are unique in this case).

There are two '`Product Name-`' columns in the output table above: '`Product Name-Sales`' contains the original product names from "`Q1Sales.xlsx`" (with missing values), and '`Product Name-Products`' contains product names from "`products.csv`". Let's combine the two columns into a single '`Product Name`' column; first, assign the joined table back to `sales_df`:

```
In [9]: sales_df = pd.merge(
    sales_df,
    products_df,
    on='ProductID',
    suffixes=['-Sales', '-Products'],
    validate='many_to_one'
)
```

To combine the two columns into a single one, let's define a custom function that choose a valid product name from the two options:

```
In [10]: def combine_product_names(row):
    if pd.notna(row['Product Name-Sales']):
        return row['Product Name-Sales']
    else:
        return row['Product Name-Products']
```

The function above is designed to accept a `sales_df` row (i.e., a `Series` object) as its input and return one of the two product name options. If the value in '`Product Name-Sales`' is `NaN`, the function will return the product name in '`Product Name-Products`'.¹ You can apply this function to `sales_df` by running:

```
In [11]: sales_df.apply(combine_product_names, axis='columns')
```

```
Out [11]: 0      Cannon Water Bomb Balloons 100 Pack
1      Cannon Water Bomb Balloons 100 Pack
2      Cannon Water Bomb Balloons 100 Pack
3      Cannon Water Bomb Balloons 100 Pack
...
37704      Violinsmart 3/4 size violin bow
37705      Violinsmart 3/4 size violin bow
37706      Violinsmart 3/4 size violin bow
37707      Violinsmart 3/4 size violin bow
Length: 37708, dtype: object
```

¹: Choosing non-`NaN` values between two columns is common enough for `Series` objects to have a `combine_first` method that works just like `combine_product_names` above.

The last step in combining product names is to assign the output above to a new column in `sales_df` and remove its previous product name columns:

```
In [12]: sales_df['Product Name'] = sales_df.apply(combine_product_names, axis='columns')
sales_df = sales_df.drop(['Product Name-Sales', 'Product Name-Products'], axis='columns')
```

If you call `sales_df.info()` now, you'll see the '`Product Name`' column doesn't have missing values anymore:

```
In [13]: sales_df.isna().sum()
```

```
Out [13]: InvoiceNo      0
Channel        0
ProductID      0
Account        0
AccountNo      0
Date           0
Deadline       0
Currency       0
Unit Price     0
Quantity       0
Total          0
Product Name   0
dtype: int64
```

You still need to write `sales_df` back to "`Q1Sales.xlsx`" to make the new product names permanent. However, "`Q1Sales.xlsx`" has three sheets — one with sales in each month of Q1 2020. To keep the three sheets, you'll need to split `sales_df` back into monthly sales and write each subset as a separate sheet. We did something similar in the first project chapter:

```
In [14]: with pd.ExcelWriter('Q1SalesClean.xlsx') as outfile:
    for month_name in sales_df['Date'].dt.month_name().unique():
        sheet_df = sales_df[sales_df['Date'].dt.month_name() == month_name]
        sheet_df.to_excel(outfile, sheet_name=month_name, index=False)
```

Notice that I used a different file name for the output data (i.e., I didn't overwrite "`Q1Sales.xlsx`"). In my experience, it's a good idea to keep your input files unchanged until you're sure the code works. After you check `Q1SalesClean.xlsx` (whether in Excel or with `pandas`) and you're confident the results of your code are what you want them to be, you can overwrite "`Q1Sales.xlsx`" by changing the file name above.

Summary

This quick project chapter showed you how to join two datasets to fill in missing product names in "`Q1Sales.xlsx`". Next, let's see how you can group and pivot your tables with `pandas`.

Groups and pivot tables

This chapter takes a look at group-based operations: splitting a table into several groups of rows and calculating statistics for each group (e.g., the sum of values in a column). This type of table summarization is widespread in any data work, accounting included.

Pivot tables are, in fact, a group-based table operation, which is why they get covered later in this chapter. Pivot tables in Excel generate plenty of mixed feelings — whichever side of that debate you're on, I think you'll find `pandas`'s `pivot_table` an intuitive and powerful improvement.

Let's start this chapter by taking a look at the mechanics of group operations in general. Later, you'll see how to write the code that makes them work in `pandas`.

How group operations work

The table below is a small subset of the sales data you've been working with throughout the book:

ProductID	Product Name	Channel	Unit Price	Quantity	Total
MI/SEN-01085	Sennheiser EW 112P...	Understock.com	18.58	6	111.48
I&S/WIH-08645	Wiha 26598 Nut Dri...	Shoppe.com	16.56	62	1026.72
H&K/KIK-91404	Kikkerland Magneti...	iBay.com	3.64	15	54.60
T&G/YU-76445	Yu-Gi-Oh! - Light-...	Understock.com	4.50	4	18.00
T&G/LAU-88048	Lauri Toddler Tote	iBay.com	14.46	1	14.46
E/AC-63975	AC Adapter/Power S...	Bullseye	28.72	8	229.76
E/CIS-74992	Cisco Systems Giga...	Bullseye	33.39	1	33.39
E/PHI-08100	Philips AJ3116M/37...	Understock.com	4.18	1	4.18
E/POL-61164	NaN	iBay.com	4.78	25	119.50
E/SIR-83381	Sirius Satellite R...	Understock.com	33.16	2	66.32

One of the first questions you might ask of these data is which channel generates the largest sales revenue. Coincidentally, the easiest way to answer that question is by applying a group operation on the table above: first split the table into groups based on values in the `Channel` column, for each group sum the `Total` column, then

combine the results back into another table. This entire process is shown below (I omitted some of the columns above):

Split			Apply (sum)		Combine	
ProductID	Channel	Total		Total		Total
E/AC-63975	Bullseye	229.76	Bullseye	263.15	Channel	
E/CIS-74992	Bullseye	33.39			Bullseye	263.15
ProductID	Channel	Total		Total		Total
I&S/WIH-08645	Shoppe.com	1026.72	Shoppe.com	1026.72	Shoppe.com	1026.72
ProductID	Channel	Total		Total		Total
MI/SEN-01085	Understock.com	111.48	Understock.com	199.98	Understock.com	199.98
T&G/YU-76445	Understock.com	18.00				
E/PHI-08100	Understock.com	4.18				
E/SIR-83381	Understock.com	66.32				
ProductID	Channel	Total		Total		Total
H&K/KIK-91404	iBay.com	54.60	iBay.com	188.56	iBay.com	188.56
T&G/LAU-88048	iBay.com	14.46				
E/POL-61164	iBay.com	119.50				

The output of this group operation is the right-most table above. In a nutshell, that's what group operations are: a procedure that summarizes the values in a table (but, as with joins, we still have a few details to iron out in the rest of this chapter).¹

As you can tell, there's nothing complicated about these group operations. What makes them powerful is that you can go from the original table to the output table in one single line of pandas code — let's see how.

1: In case you come across it, these operations are also sometimes called *split-apply-combine* operations (after the three steps of the process illustrated above).

Group operations in pandas

For the code examples in this section, let's create the same table as above from our sales data. This will help keep output tables concise and easy to check — we will use the entire sales data towards the end of the chapter to apply the same concepts on a larger dataset.

To create a `DataFrame` with the same rows as the table from the previous section, you can use the following code:

```
In [1]: columns = ['ProductID', 'Product Name', 'Channel', 'Unit Price', 'Quantity', 'Total']
sample_df = ledger_df[columns].tail(10)
```

```
In [2]: sample_df
```

Out [2]:

	ProductID	Product Name	Channel	Unit Price	Quantity	Total
14044	MI/SEN-01085	Sennheiser EW 112P...	Understock.com	18.58	6	111.48
14045	I&S/WIH-08645	Wiha 26598 Nut Dri...	Shoppe.com	16.56	62	1026.72
14046	H&K/KIK-91404	Kikkerland Magneti...	iBay.com	3.64	15	54.60
14047	T&G/YU--76445	Yu-Gi-Oh! - Light...	Understock.com	4.50	4	18.00
14048	T&G/LAU-88048	Lauri Toddler Tote	iBay.com	14.46	1	14.46
14049	E/AC-63975	AC Adapter/Power S...	Bullseye	28.72	8	229.76
14050	E/CIS-74992	Cisco Systems Giga...	Bullseye	33.39	1	33.39
14051	E/PHI-08100	Philips AJ3116M/37...	Understock.com	4.18	1	4.18
14052	E/POL-61164	NaN	iBay.com	4.78	25	119.50
14053	E/SIR-83381	Sirius Satellite R...	Understock.com	33.16	2	66.32

The `sample_df` DataFrame you just created contains the bottom ten rows of our sales data (notice the row labels above). As promised, you can run the entire group operation illustrated earlier with one line of pandas code:

In [3]: `sample_df.groupby('Channel').agg({'Total': 'sum'})`

Out [3]:

Channel	Total
Bullseye	263.15
Shoppe.com	1026.72
Understock.com	199.98
iBay.com	188.56

There are two methods used in the example above: `groupby`, which tells pandas what column to use when splitting the input table into groups, and `agg`, which tells pandas what to include in the output table (here, the sum of values in the `'Total'` column for each group).² You can group by multiple columns, and compute one or more aggregates for one or more columns using the two methods above. Before we go over the different ways to use `groupby`, we need to take a quick look inside the pandas machinery for group operations — knowing how `groupby` works will help you develop an intuition for when and how to use it effectively.

2: If you are familiar with SQL, the `groupby` method is similar to SQL's GROUP BY operator, but much more flexible.

The pandas group object

Instead of chaining `groupby` and `agg`, as above, let's take it one step at a time by assigning the output of `groupby` to another variable:

In [4]: `groups = sample_df.groupby('Channel')`

The `groups` variable (and the output of `groupby` in general) is a special kind of pandas object called a `DataFrameGroupBy`³ — you can check its type using:

In [5]: `type(groups)`

3: The `pandas.core.groupby.generic` part is just the directory structure inside the `pandas` library where you can find the definition and code for `DataFrameGroupBy`.

```
In [5]: pandas.core.groupby.generic.DataFrameGroupBy
```

What's a `DataFrameGroupBy` object? It is similar to a Python dictionary, mapping unique values in the grouping column (here, values from `'Channel'` column) to a list of row labels — you can take a look inside this object and see each group of rows by running:

```
In [6]: groups.groups
```

```
Out [6]: {'Bullseye': [14049, 14050],  
          'Shoppe.com': [14045],  
          'Understock.com': [14044, 14047, 14051, 14053],  
          'iBay.com': [14046, 14048, 14052]}
```

In the output above, the numbers associated with each key are the actual row labels from `sample_df`. In short, what `groupby` does is create a mapping between unique column values (in this case, unique values from the `'Channel'` column) and the row labels where those values are in the original `DataFrame`. You can access any group as a separate `DataFrame` using :

```
In [7]: groups.get_group('Bullseye')
```

```
Out [7]:   ProductID      Product Name  Channel  Unit Price  Quantity  Total  
14049    E/AC-63975  AC Adapter/Power Sup...  Bullseye     28.72       8  229.76  
14050    E/CIS-74992  Cisco Systems Gigabi...
```

And you can access one or more columns from each separate group of rows, using the same square bracket notation you use to access columns in `DataFrame` objects:

```
In [8]: groups['Total'].get_group('Bullseye')
```

```
Out [8]: 14049    229.76  
14050    33.39  
Name: Total, dtype: float64
```

```
In [9]: groups[['Quantity', 'Total']].get_group('Bullseye')
```

```
Out [9]:   Quantity  Total  
14049        8  229.76  
14050        1   33.39
```

What's more interesting is that after you select columns you can use any `Series` method with the `groups` object to call that method on each group independently:

```
In [10]: groups['Total'].sum()
```

```
Out [10]: Channel  
Bullseye           263.15  
Shoppe.com         1026.72  
Understock.com    199.98  
iBay.com          188.56  
Name: Total, dtype: float64
```

```
In [11]: groups[['Quantity', 'Total']].sum()
```

	Quantity	Total
Channel		
Bullseye	9	263.15
Shoppe.com	62	1026.72
Understock.com	13	199.98
iBay.com	41	188.56

Depending on how many columns you select, the output is either a `Series` or a `DataFrame` object (i.e., if you select multiple columns, the output is a `DataFrame`). You can put all the earlier steps together and get the same result in one line of `pandas` code with:

```
In [12]: sample_df.groupby('Channel')['Total'].sum()
```

Channel
Bullseye
Shoppe.com
Understock.com
iBay.com
Name: Total, dtype: float64

This code is slightly different from the `agg` example you saw earlier, but it produces the same result. You will come across both styles of applying group operations in code examples you find online: this style is not the easiest to understand, and it takes a while to get used to.

There are two specialized methods⁴ available on the group object you can use to compute group summaries:

- ▶ `agg` — or its alias `aggregate` — which we used earlier and produces a single value from each group of rows (e.g., the sum of a column for each group);
- ▶ `apply` which you can use to manipulate each group independently in several ways, including filtering out groups based on a group value (e.g., filtering out groups that have only one row) or transforming columns based on a group statistic (e.g., dividing each value in a column by the sum of that column, for each group separately). Using `apply` with groups is very similar to using `apply` with `DataFrame` objects.

4: There are two other methods available on the `group` object: `transform` and `filter`. However, you can achieve the same functionality using just the two methods mentioned here.

Let's take a look at `agg` first.

Aggregating group functions

Aggregations are perhaps the most common group operation you will need in your work. The `agg` group method allows you to apply predefined aggregating functions (e.g., `sum`) as well as custom aggregating functions to your data. You have already seen the code below:

```
In [13]: # this works as well
# sample_df.groupby('Channel')['Total'].agg('sum')
sample_df.groupby('Channel').agg({'Total': 'sum'})
```

Out [13]:

Channel	Total
Bullseye	263.15
Shoppe.com	1026.72
Understock.com	199.98
iBay.com	188.56

The argument passed to `agg` is a Python dictionary mapping the name of the column you want to summarize to the name of the function you want to use for summarizing (both as string values). In addition to `'sum'`, there are several out-of-the-box aggregating functions you can use — table 28.1 lists some of them.

Table 28.1: Common pandas aggregating functions for groups. You can find all of them in the pandas documentation at pandas.pydata.org/pandas-docs/stable/user_guide/groupby.html#aggregation.

Function name	Example	Description
<code>'mean'</code>	(sample_df .groupby('Channel') .agg({'Total': 'mean'}))	Computes the mean of the <code>'Total'</code> column for each group.
<code>'median'</code>	(sample_df .groupby('Channel') .agg({'Total': 'median'}))	Computes the median of the <code>'Total'</code> column for each group.
<code>'sum'</code>	(sample_df .groupby('Channel') .agg({'Quantity': 'sum'}))	Computes the sum of the <code>'Quantity'</code> column for each group.
<code>'size'</code>	(sample_df .groupby('Channel') .agg('size'))	Computes the size of each group (i.e., the number of rows in each group). Notice that you don't have to specify a column name because the number of rows is the same in any column.
<code>'count'</code>	(sample_df .groupby('Channel') .agg({'Total': 'count'}))	Computes the number of non-empty entries (i.e., non-NaN values) in the <code>'Total'</code> column for each group.
<code>'first'</code>	(sample_df .groupby('Channel') .agg({'Quantity': 'first'}))	Returns the first value in the <code>'Quantity'</code> column for each group (row order is the same as in the input DataFrame).
<code>'last'</code>	(sample_df .groupby('Channel') .agg({'Total': 'last'}))	Same as above, but returns the last value in the <code>'Total'</code> column for each group.

Table 28.1: Common pandas aggregating functions for groups. You can find all of them in the pandas documentation at pandas.pydata.org/pandas-docs/stable/user_guide/groupby.html#aggregation.

Function name	Example	Description
'min'	(sample_df .groupby('Channel') .agg({'Total': 'min'}))	Returns the minimum value in the 'Total' column for each group.
'max'	(sample_df .groupby('Channel') .agg({'Quantity': 'max'}))	Returns the maximum value in the 'Quantity' column for each group.

If you want to summarize several columns, not just 'Total', you can include multiple column names in the dictionary passed to agg. Even better, you can pick-and-choose aggregating functions for each of the columns you want to summarize:

```
In [14]: (sample_df
      .groupby('Channel').agg({
          'Total': ['sum', 'mean'],
          'Quantity': ['sum', 'max', 'min']
      })
)
```

```
Out [14]:
```

Channel	Total		Quantity		
	sum	mean			
			sum	max	min
Bullseye	263.15	131.575000	9	8	1
Shoppe.com	1026.72	1026.720000	62	62	62
Understock.com	199.98	49.995000	13	6	1
iBay.com	188.56	62.853333	41	25	1

A double-take might be necessary for the code above, but the grouping mechanics are the same as before: pandas splits the table into several groups of rows using values from the 'Channel' column, then for each group computes the specified summaries on their 'Total' and 'Quantity' columns.

The output above is a familiar DataFrame, but with a twist: it has two levels of column names. I briefly mentioned in chapter 25 that rows can have multiple levels of labels — here, you have the same idea, but applied to columns. If you want to select a column from this DataFrame, you need to specify two column labels, not just one:

```
In [15]: # assigns the output above
# to another variable
aggregate_df = (
    sample_df
    .groupby('Channel').agg({
        'Total': ['sum', 'mean'],
        'Quantity': ['sum', 'max', 'min']
    })
)
```

```

    })
)

# selects the max column under Quantity
aggregate_df.loc[:, ('Quantity', 'max')]

```

Out [15]:

Channel	
Bullseye	8
Shoppe.com	62
Understock.com	6
iBay.com	25
Name:	(Quantity, max), dtype: int64

Or you can select an entire group of columns by specifying the top-level column name:⁵

In [16]:

```
aggregate_df.loc[:, 'Quantity']
```

Out [16]:

Channel	sum	max	min
Bullseye	9	8	1
Shoppe.com	62	62	62
Understock.com	13	6	1
iBay.com	41	25	1

5: Remember that with `loc`, you use row labels and column labels to slice a `DataFrame`. The colon used as the first argument for `loc`, in both examples on this page, stands for *all row labels*.

You'll soon notice that group operations tend to produce `DataFrame` objects with multiple levels of row and column labels — in particular when you group by values from multiple columns. There's nothing complicated about these multi-leveled `DataFrame` objects (they're still tables), you just need to use several labels when slicing or selecting columns from them.

Custom aggregating functions

If there's no aggregating function available out-of-the-box for what you need to do, you can always define your own function to use with `agg` — just like you can `apply` any custom function to the rows or columns of a `DataFrame`.

Let's say you want to calculate the difference between the largest sale and the smallest sale in each channel (i.e., a single summary value for each channel). You can use `groupby`, but there's no predefined aggregating function that comes with `pandas` and does what you want. No need to panic, you can define and use your aggregating function with:

In [17]:

```

def total_diff(column):
    return column.max() - column.min()

sample_df.groupby('Channel').agg({'Total': ['min', 'max', total_diff]})

```

Out [17]:

Channel	Total		
	min	max	total_diff
Bullseye	33.39	229.76	196.37
Shoppe.com	1026.72	1026.72	0.00
Understock.com	4.18	111.48	107.30
iBay.com	14.46	119.50	105.04

The `total_diff` function above is designed to accept a `Series` object as its only argument (i.e., the value assigned to the `column` parameter when the function is called should be a `Series` object). Inside the function body, the difference between the maximum value in `column` and its minimum value is calculated and returned. When you use it with `agg` as I did above, pandas calls the function for each group, passing the group's selected column as the function argument (here, the '`Total`' column is passed). Notice also that you can mix predefined functions with custom functions in the same `agg` call.

Inside the custom function, you can manipulate the `Series` object any way you need to, using any `Series` methods (including the string or date methods we looked at in previous chapters, if the column you want to summarize contains strings or dates). However, your custom function must return a single value, otherwise, you will get an error:⁶

```
In [18]: def custom_aggregating_function(column):
    return column
```

```
sample_df.groupby('Channel').agg({'Total': custom_aggregating_function})
```

Out [18]:

```
...
ValueError                                     Traceback (most recent call last)
<ipython-input-21-199163d9ba95> in <module>
      2     return column
      3
----> 4 sample_df.groupby('Channel').agg({'Total': custom_aggregating_function})
...
ValueError: Must produce aggregated value
```

6: A function that takes a sequence of values as an argument (e.g., a `Series` object) and returns a single number computed from that sequence of values (e.g., the sum) is sometimes called a *reducer*.

Overthinking: Other group functions

While aggregations are the most common kind of operation you will be performing with groups, it is worth knowing that there are other kinds as well: group filters and group transformations. These operations can be applied to each group separately by defining a custom function and then using it with the `apply` method (similar to how you use `apply` with `DataFrames`).

Group filters allow you to select groups of rows from your table that meet a certain group-level condition. Say you want to select

rows from those channels that have total sales over 200 dollars — you have several options to get there, but perhaps the shortest path is by defining a custom filtering function and using it with `groupby` and the `apply` method:

```
In [19]: def filter_group(group_df):
    return group_df if group_df['Total'].sum() > 200 else None

sample_df.groupby('Channel').apply(filter_group)
```

```
Out [19]:
```

	ProductID	Product Name	Channel	Unit Price	Quantity	Total
Channel						
Bullseye	14049	E/AC-63975 AC Adapter...	Bullseye	28.72	8	229.76
	14050	E/CIS-74992 Cisco Syst...	Bullseye	33.39	1	33.39
Shoppe.com	14045	I&S/WIH-08645 Wiha 26598...	Shoppe.com	16.56	62	1026.72

This returns the original DataFrame, in the same shape, but removes those rows that belonged to channels that totaled less than 200 dollars in sales (i.e., removes sales from '[iBay.com](#)' and '[Understock.com](#)' channels). When you call `apply` with a custom function, as you did here, pandas passes each group to your function as a `DataFrame` argument, one at a time (i.e., notice you didn't select a column after using `groupby`, but called `apply` directly). Inside the function, you can manipulate the group `DataFrame` (in this case, the parameter called `group_df`) any way you need to.

Row labels in the output above have multiple levels: the first level indicates which group each row is part of (based on the grouping columns you used with `groupby`); the second level has the original row labels from `sample_df`. If you don't need these group labels, you can discard them using `reset_index(drop=True)`:

```
In [20]: sample_df.groupby('Channel').apply(filter_group).reset_index(drop=True)
```

```
Out [20]:
```

	ProductID	Product Name	Channel	Unit Price	Quantity	Total
0	E/AC-63975	AC Adapter/Power Sup...	Bullseye	28.72	8	229.76
1	E/CIS-74992	Cisco Systems Gigabi...	Bullseye	33.39	1	33.39
2	I&S/WIH-08645	Wiha 26598 Nut Drive...	Shoppe.com	16.56	62	1026.72

Another handy use of `apply` is transforming values in a group based on a group statistic. For example, if you want to compute what percentage of the total channel sales each sale in that channel represents, you can use:

```
In [21]: def percent_group_total(group_df):
    group_df['% Group Total'] = group_df['Total'] / group_df['Total'].sum() * 100
    group_df['% Group Total'] = group_df['% Group Total'].round(2)

    return group_df

sample_df.groupby('Channel').apply(percent_group_total)
```

Out [21]:

	ProductID	Product Name	Channel	Unit Price	Quantity	Total	% Group Total	Group Total
14044	MI/SEN-0...	Sennheis...	Understo...	18.58	6	111.48		55.75
14045	I&S/WIH-...	Wiha 265...	Shoppe.com	16.56	62	1026.72		100.00
14046	H&K/KIK-...	Kikkerla...	iBay.com	3.64	15	54.60		28.96
14047	T&G/YU-...	Yu-Gi-Oh...	Understo...	4.50	4	18.00		9.00
14048	T&G/LAU-...	Lauri To...	iBay.com	14.46	1	14.46		7.67
14049	E/AC-63975	AC Adapt...	Bullseye	28.72	8	229.76		87.31
14050	E/CIS-74992	Cisco Sy...	Bullseye	33.39	1	33.39		12.69
14051	E/PHI-08100	Philips ...	Understo...	4.18	1	4.18		2.09
14052	E/POL-61164	NaN	iBay.com	4.78	25	119.50		63.38
14053	E/SIR-83381	Sirius S...	Understo...	33.16	2	66.32		33.16

Notice that values in the new '`% Group Total`' column sum to 100 for each separate channel (e.g., '`% Group Total`' for the one '`Shoppe.com`' row is 100.00).

The `percent_group_total` function above is similar to the filtering function used in the earlier example. In this case, it adds a column to each group `DataFrame` and then returns the group `DataFrame` entirely (without aggregating or removing rows) — pandas takes care of combining the separate group outputs back into a single `DataFrame`.

Hopefully, these examples show how flexible `groupby` and `apply` can be. You can get very creative with both, and you will soon figure out different ways to use these tools that make sense to you and your data. Something to keep in mind is that `apply` can become slow on large datasets with many groups — in those cases, you might want to use pandas's built-in aggregating functions to get the result you're looking for.

Stacking and unstacking

On the road to pivot tables, we need to make a quick stop at stacking and unstacking. Stacking and unstacking are operations that change the shape of a table, but don't modify its data in any way. Stacking refers to placing all values in a table on top of each other, in a single column (i.e., arranging them in a stack of values). The figure below illustrates how stacking works in pandas:

Unstacked table

ProductID	Channel	Total
T&G/CAN-97509	Shoppe.com	281.54
T&G/LEG-37777	Walcart	6.70
T&G/PET-14209	Bullseye	58.35
T&G/TRA-20170	Bullseye	80.76
T&G/TRA-20170	Bullseye	80.76

Stacked table

T&G/CAN-97509	Shoppe.com	281.54
T&G/LEG-37777	Walcart	6.70
T&G/PET-14209	Bullseye	58.35
T&G/TRA-20170	Bullseye	80.76
T&G/TRA-20170	Bullseye	80.76

In pandas, the default stacking behavior is to first stack values in each row, from left to right, then stack all rows on top of each other. If you want to stack columns directly, similar to copying entire columns one under the other in Excel, you need to use the `concat` function instead.

Let's first create an even smaller sample table to show how stacking works in pandas:

```
In [22]: columns = ['ProductID', 'Channel', 'Total']
sample_df = ledger_df[columns].head()

sample_df
```

```
Out [22]:      ProductID    Channel   Total
0  T&G/CAN-97509  Shoppe.com  281.54
1  T&G/LEG-37777       Walcart    6.70
2  T&G/PET-14209     Bullseye   58.35
3  T&G/TRA-20170     Bullseye  80.76
4  T&G/TRA-20170     Bullseye  80.76
```

To stack this table (which has the same data as the illustration above), you use the `stack` method:

```
In [23]: sample_df.stack()
```

```
Out [23]: 0  ProductID      T&G/CAN-97509
           Channel          Shoppe.com
           Total            281.54
 1  ProductID      T&G/LEG-37777
           Channel          Walcart
           Total             6.7
 2  ProductID      T&G/PET-14209
           Channel        Bullseye
           Total            58.35
 3  ProductID      T&G/TRA-20170
           Channel        Bullseye
           Total            80.76
 4  ProductID      T&G/TRA-20170
           Channel        Bullseye
           Total            80.76
dtype: object
```

The output is a `Series` object — however, it doesn't look exactly like the illustration above because it includes a bunch of row labels as well. When pandas stacks the values in each row, it uses the original column names as row labels for the row's stacked version. For every row in the input table, it does the same thing and puts stacked rows on top of each other, as you can see above. The result is a `Series` object with two levels of row labels: the first level is the same as the input table, whereas the second level contains column names from the input table (the actual row values in the table remain unchanged).

Unstacking is the inverse operation: it transforms the second level of row labels (or, more generally, the outer-most level of row labels) and their associated values into table columns. You can use it whenever you have a `Series` or `DataFrame` with multiple levels of row labels. To rotate the stacked version of `sample_df` back to its original shape, you can use:

```
In [24]: stacked_sample = sample_df.stack()

stacked_sample.unstack()
```

```
Out [24]:      ProductID    Channel   Total
0  T&G/CAN-97509  Shoppe.com  281.54
1  T&G/LEG-37777     Walcart     6.7
2  T&G/PET-14209    Bullseye   58.35
3  T&G/TRA-20170    Bullseye   80.76
4  T&G/TRA-20170    Bullseye   80.76
```

All this may seem riveting on its own, but unstacking is most useful when you want to reshape a table output by `groupby`. As it turns out, grouping, aggregating and then unstacking values are the three steps in creating a pivot table.

Pivot tables

There are probably more pivot table tutorials available on the web than there are about all other Excel features combined. The reason for all the tutorials is that pivot tables are just as powerful as they are confusing. Much of the confusion comes from the fact that their internal machinery is hidden (i.e., you get to see just the input and the output) and involves several distinct steps, so you can't easily develop intuition about *how* they work.

This section walks you through pivoting a `DataFrame` using the `groupby`, `agg` and `unstack` methods first, before showing you how to get the same result with pandas's `pivot_table` function. Hopefully, going through each step will help you see how pivot tables work, not just what they do.

First, let's go back to `ledger_df` and create a new column that tells us by what quarter a product is considered “*aged*” (remember that the '`Deadline`' column holds the date after which a product is considered “*aged*” for inventory tracking purposes) — you can do that using:

```
In [25]: ledger_df = pd.read_excel('Q1Sales.xlsx')

ledger_df['Deadline'] = pd.to_datetime(ledger_df['Deadline'])
ledger_df['Deadline Quarter'] = ledger_df['Deadline'].dt.to_period(freq='Q-DEC')

ledger_df
```

```
Out [25]:
```

	InvoiceNo	Channel	Product Name	...	Quantity	Total	Deadline	Quarter
0	1532	Shoppe.com	Cannon Water...	...	14	281.54		2019Q4
1	1533	Walcart	LEGO Ninja T...	...	1	6.70		2020Q2
2	1534	Bullseye		NaN	5	58.35		2020Q2
3	1535	Bullseye	Transformers...	...	6	80.76		2019Q4
4	1535	Bullseye	Transformers...	...	6	80.76		2019Q4
...
14049	15581	Bullseye	AC Adapter/P...	...	8	229.76		2020Q1
14050	15582	Bullseye	Cisco System...	...	1	33.39		2020Q1
14051	15583	Understock.com	Philips AJ31...	...	1	4.18		2020Q1
14052	15584	iBay.com		NaN	25	119.50		2020Q2
14053	15585	Understock.com	Sirius Satel...	...	2	66.32		2020Q1

[14054 rows x 13 columns]

The code above uses only pandas functions or methods you've seen before. It reads the sales dataset again (on line 1), converts the 'Deadline' column to the datetime64 data type, then extracts quarter labels from the 'Deadline' column (using `to_period`) and assigns them to a new column called 'Deadline Quarter'.

The reason for the new 'Deadline Quarter' column is to help us check how many items continue to sell after their aging quarter across the different channels. To get that metric, you can group the sales data by both 'Channel' and 'Deadline Quarter' values, and sum the 'Quantity' column for each group:

```
In [26]: ledger_df.groupby(['Channel', 'Deadline Quarter']).agg({'Quantity': 'sum'})
```

```
Out [26]:
```

Channel	Deadline Quarter	Quantity
		Bullseye
	2020Q1	3413
	2020Q2	3034
	2020Q3	377
Shoppe.com	2019Q4	11424
	2020Q1	9969
	2020Q2	11212
	2020Q3	1757
Understock.com	2019Q4	18518
	2020Q1	21614
	2020Q2	23222
	2020Q3	3484
Walcart	2019Q4	3816
	2020Q1	5174
	2020Q2	3921
	2020Q3	411
iBay.com	2019Q4	9711
	2020Q1	11525
	2020Q2	11995
	2020Q3	1317

In this case, row grouping is done using values from two input columns (i.e., not just 'Channel', as in the previous examples). How-

ever, the same table-splitting mechanism is used: each combination of channel and deadline quarter values determines what group an input table row belongs to. Then, for each separate group of rows, values in the '`Quantity`' column are summed and combined in the `DataFrame` you see above. In general, you can `groupby` as many columns as you want to — however, multiple column names need to be passed to `groupby` in a Python list. Their order in the list determines row label levels in the output `DataFrame`.

The result above is a `DataFrame` with multi-level row labels: the first level has unique values from the '`Channel`' column; the second level has unique values from the '`Deadline Quarter`' column. It's the result we wanted, but in long instead of wide format, making it hard to compare values from the same quarter across channels. To fix that, you can `unstack` the outer-most level of row labels, turning it into a set of columns:

```
In [27]: ledger_df.groupby(['Channel', 'Deadline Quarter']).agg({'Quantity': 'sum'}).unstack()
```

```
Out [27]:
```

		Quantity				
		Deadline Quarter	2019Q4	2020Q1	2020Q2	2020Q3
Channel						
Bullseye		2212	3413	3034	377	
Shoppe.com		11424	9969	11212	1757	
Understock.com		18518	21614	23222	3484	
Walcart		3816	5174	3921	411	
iBay.com		9711	11525	11995	1317	

Now the output is a `DataFrame` with one row for every channel in the sales data and a column for each deadline quarter — the values in the table are the sum of the '`Quantity`' column for different combinations of '`Channel`' and '`Deadline Quarter`'. You may have noticed that, even though we used a combination of `groupby`, `agg` and `unstack` to get here, the result above is a pivot table based on `ledger_df`'s data.

This is how pivot tables work in `pandas`: first, they group rows using values in two (or more) columns, then compute an aggregate value for each group of rows, and finally `unstack` (i.e., “rotate”) values from one of the grouping columns into a header for the output.

As I mentioned earlier, you can get the same result with less code by using `pandas`'s `pivot_table` function:

```
In [28]: pd.pivot_table(ledger_df,
                      index='Channel',
                      columns='Deadline Quarter',
                      values='Quantity',
                      aggfunc='sum')
```

```
1  
2  
3  
4  
5
```

```
Out [28]:
```

	Deadline	Quarter	2019Q4	2020Q1	2020Q2	2020Q3
Channel						
Bullseye			2212	3413	3034	377
Shoppe.com			11424	9969	11212	1757
Understock.com			18518	21614	23222	3484
Walcart			3816	5174	3921	411
iBay.com			9711	11525	11995	1317

The code above is perhaps more similar to how you construct pivot tables in Excel, but it is just a shortcut for the same `groupby-agg-unstack` operation you saw earlier. The first argument passed to `pivot_table` is the `DataFrame` you want to summarize, followed by several keyword arguments (which are all required):

- ▶ `index` and `columns` tell pandas what columns to use for grouping rows. Unique values from these two columns will become the row labels and column header in the output `DataFrame`, respectively;
- ▶ `values` and `aggfunc` tell pandas what values from the input table to aggregate and what function to use when aggregating them (in this case, the `sum` function, but you can use any of the aggregating functions mentioned in table 28.1 or your own functions).

Something you can easily do with `pivot_table` — and less so with the `groupby-agg-unstack` approach — is add both row-wise and column-wise totals to the output table by passing the `margins` and `margins_name` keyword arguments:

```
In [29]: pd.pivot_table(ledger_df,
                     index='Channel',
                     columns='Deadline Quarter',
                     values='Quantity',
                     aggfunc='sum',
                     margins=True,
                     margins_name='TOTAL')
```

```
Out [29]:
```

	Deadline	Quarter	2019Q4	2020Q1	2020Q2	2020Q3	TOTAL
Channel							
Bullseye			2212	3413	3034	377	9036
Shoppe.com			11424	9969	11212	1757	34362
Understock.com			18518	21614	23222	3484	66838
Walcart			3816	5174	3921	411	13322
iBay.com			9711	11525	11995	1317	34548
TOTAL			45681	51695	53384	7346	158106

As with `groupby`, instead of single column names, you can pass lists of column names for the keyword arguments available with `pivot_table` (or a list of aggregating functions to `aggfunc`). The result would be a more complex pivot table, with more columns or rows, but still based on the same grouping, aggregating and unstacking machinery you saw earlier — give it a try.

Summary

This chapter covered group operations in `pandas`. You'll often need to split data into groups and compute summaries on each group independently. In `pandas`, you do that by calling `groupby` and specifying what group aggregation or filtering methods you want to use. The last part of this chapter also showed you how to pivot tables using `pandas`'s `pivot_table` function.

If you're used to Excel, the way `DataFrame` variables are displayed in Jupyter notebooks might be frustrating at times. However, you can use `pandas` functions to customize the display of tables in your notebooks — the following chapter shows you how.

Overthinking: Changing how DataFrames are displayed

29

The previous chapters have taken you through a detailed tour of pandas's table handling features. The following part of the book guides you through Python's data visualization landscape. However, before we leave pandas territory, there's one more feature I want to cover: changing how DataFrame objects look like in your notebooks (e.g., how tables get truncated, how decimal values are formatted, how to change table text size, etc.).

As in the previous chapters, launch JupyterLab and `import pandas` in your notebook. We'll be looking at table display rather than table data, so a small DataFrame is enough for this chapter — you can load the top 100 rows in "Q1Sales.xlsx" by running:

```
In [1]: sales_df = pd.read_excel('Q1Sales.xlsx', nrows=100)
```

Now that you have a DataFrame let's change how it's displayed in your notebook. With pandas you can configure both table *display* options and table *styling* options — the difference between these options will become apparent as we work through some examples. Let's look at table display options first.

Setting display options

One of the most noticeable differences between Excel and Jupyter notebooks is that you don't see all your table data in notebooks. When you inspect a DataFrame in a Jupyter notebook, you only see its top and bottom rows (and only its outer columns if your table has many columns). However, you can configure how many rows or columns from your table get displayed. For instance, you can set the maximum number of columns that get shown when you inspect a DataFrame by running:

```
In [2]: pd.options.display.max_columns = 6
```

After running the code above, if you inspect `sales_df` in a separate cell, you'll see only six of its columns (i.e., its three outermost columns on the left and right sides of the table).

Similarly, you can set the minimum and maximum number of rows that get shown when you view a DataFrame:

```
In [3]: pd.options.display.min_rows = 20
pd.options.display.max_rows = 40
```

If you inspect `sales_df` now, you'll see ten of its top and bottom rows (i.e., `min_rows` altogether). The `max_rows` option sets the maximum number of rows that get displayed — if a `DataFrame` has fewer rows than the value of `max_rows`, all of its rows get displayed. Once the number of rows in a `DataFrame` exceeds `max_rows`, the `min_rows` value determines how many rows are shown.

To display *all* the data in your `DataFrame` (i.e., to show all its rows and columns) you can set the display options above to `None`:

```
In [4]: pd.options.display.max_columns = None
pd.options.display.max_rows = None
```

Now, if you inspect `sales_df`, you'll see all of its rows and columns. This might seem like an improvement over the truncated tables you're used to by now. However, if you work with large datasets, setting the display options above to `None` will make your notebooks challenging to work with (you'll have to scroll a lot to find your code cells, and your notebooks will likely become slower to use).

Setting display options this way affects the display of *all* tables in your notebook. You can set several other display options, including `max_colwidth`, which sets the width of columns, and `precision`, which sets how many decimal numbers get shown in your tables — table 29.1 lists some of the other options.

Table 29.1: Display options available with `pandas`. These (and more) options are documented in the official `pandas` documentation at pandas.pydata.org/pandas-docs/stable/user_guide/options.html#available-options.

Option name	Default value	Description
<code>chop_threshold</code>	<code>None</code>	If set to a float value, all decimal numbers smaller than the given threshold will be displayed as exactly 0 in the table.
<code>colheader_justify</code>	<code>'right'</code>	Controls the justification of column headers. Valid options are <code>'left'</code> or <code>'right'</code> .
<code>date_dayfirst</code>	<code>False</code>	When <code>True</code> , prints and parses dates with the day first (e.g., 20/01/2005).
<code>date_yearfirst</code>	<code>False</code>	When <code>True</code> , prints and parses dates with the year first, (e.g., 2005/01/20).
<code>float_format</code>	<code>None</code>	The value assigned to this option should be a function that accepts a floating point number as its only argument and returns a string with the desired format of the number.
<code>max_columns</code>	20	Sets the number of columns displayed. Setting this option to <code>None</code> means all columns get displayed.
<code>max_colwidth</code>	50	The maximum width in characters of a column. When the column has more characters than <code>max_colwidth</code> , a placeholder (i.e., an ellipsis) is added to the column value. Setting this option to <code>None</code> means all characters get displayed.

Table 29.1: Display options available with pandas. These (and more) options are documented in the official pandas documentation at pandas.pydata.org/pandas-docs/stable/user_guide/options.html#available-options.

Option name	Default value	Description
max_rows	60	Sets the maximum number of rows displayed. Setting this option to <code>None</code> means all rows get displayed.
min_rows	10	Sets the number of rows to show in a truncated table (when the number of rows in the table exceeds <code>max_rows</code>). Ignored when <code>max_rows</code> is set to <code>None</code> . When set to <code>None</code> , follows the value of <code>max_rows</code> .
precision	6	The number of places after the decimal used when displaying decimal numbers in a table.
show_dimensions	'truncate'	Whether to print out dimensions at the end of a displayed DataFrame. If this option is set to ' <code>'truncate'</code> ', only print out the dimensions if the table is truncated (i.e., not all rows or columns in the table are displayed); <code>True</code> or <code>False</code> are valid options.

A display option that you might be particularly interested in is `float_format`. You can use this display option to format the display of decimal numbers in your tables. However, instead of a value, you need to assign a custom function to this option: a function that takes a single number as its only argument and returns a string value. For instance, to set a comma as the thousands separator for decimal numbers in your tables, you can run:

```
In [5]: def float_format_function(value):
    return f'{value:,}'
```

Now if you inspect a Series or DataFrame with decimal numbers, you'll see the numbers formatted with a comma as the thousands separator:

```
In [6]: pd.Series([1000.0, 2000.0, 5000.0])
```

```
Out [6]: 0    1,000.0
1    2,000.0
2    5,000.0
Length: 3, dtype: float64
```

You can use any custom function to format decimal values; for instance, you can even add a message to your formatting function:

```
In [7]: def float_format_function(value):
    return f'My value is {value:,}'

pd.options.display.float_format = float_format_function

pd.Series([1000.0, 2000.0, 5000.0])
```

```
Out [7]: 0  My value is £1,000.0
          1  My value is £2,000.0
          2  My value is £5,000.0
Length: 3, dtype: float64
```

If something goes wrong, and you want to revert all display options to their default values, you can call the `pandas` function below. Display options also get reset when you stop (or restart) your notebooks.

```
In [8]: pd.reset_option("display")
```

Keep in mind that the options above change the way tables look in your notebooks, not their data. If you save a `DataFrame` to an Excel file, you'll see the same data when you open the file with Excel regardless of your notebook display options.

Styling tables

The `pandas` options above change the display of all tables in your notebook. However, you can use `pandas` to apply conditional styling to single `DataFrame` variables as well (e.g., set text or background color, change font size, add borders, etc.). Even better, you can also save your styled `DataFrame` variable (together with its custom colors or fonts) as an Excel spreadsheet.

To set visual styles on a `DataFrame`, you need to use its `style` attribute and call one of the methods that format and style the `DataFrame`. As with the display options above, these methods don't modify your table's data, just its look. For instance, to display `sales_df` with blue text and a pink background, you can run:

```
In [9]: sales_df.style.set_properties(**{'color':'blue', 'background-color': 'pink'})
```

If you run the code above, you should see a pink table with blue text. The code is somewhat strange: `set_properties` is a method that accepts an arbitrary number of arguments. The `**` operator above turns a Python dictionary into a sequence of arguments that `set_properties` can work with. Regardless of its details, the code above is what you need to run to change the visual style of an entire `DataFrame`.

Keep in mind that running the code above doesn't return another `DataFrame` but a `Styler` object so you can't chain other `DataFrame` methods to the output of `set_properties` above:

```
In [10]: table = sales_df.style.set_properties(**{'color':'blue', 'background-color': 'pink'})

type(table)
```

Out [10]: `pandas.io.formats.style.Styler`

As such, styling your tables should be the last step in your data wrangling process.

Besides '`color`' and '`background-color`', there are several other visual properties you can set for your tables. These properties are "borrowed" from CSS (i.e., *Cascading Style Sheets* is one of the computer languages that websites are built with) — table 29.2 lists some of the properties you can set together with their valid options.

Table 29.2: Properties available with pandas's styling feature. These (and more) properties are documented in the official pandas documentation at pandas.pydata.org/pandas-docs/stable/user_guide/style.html.

Style name	Valid options	Description
<code>'background-color'</code>	Any valid CSS color name (the next chapter lists CSS colors).	Sets the table or cell background color.
<code>'border-style'</code>	One of <code>'dotted'</code> , <code>'dashed'</code> , <code>'solid'</code> , or <code>'double'</code> .	Sets the table border line style. You can control border lines separately by changing the property name to one of <code>'border-left-style'</code> , <code>'border-top-style'</code> , <code>'border-right-style'</code> , or <code>'border-bottom-style'</code> .
<code>'border-width'</code>	One of <code>'thin'</code> , <code>'medium'</code> , <code>'thick'</code> , or a specific value such as <code>'4pt'</code> . You must set a <code>'border-style'</code> for this to work.	Sets the table border line width. You can control border lines separately by changing the property name to one of <code>'border-left-width'</code> , <code>'border-top-width'</code> , <code>'border-right-width'</code> , or <code>'border-bottom-width'</code> .
<code>'border-color'</code>	Any valid CSS color name (the next chapter lists CSS colors).	Sets the table border line color. You can control border lines separately by changing the property name to one of <code>'border-left-color'</code> , <code>'border-top-color'</code> , <code>'border-right-color'</code> , or <code>'border-bottom-color'</code> .
<code>'color'</code>	Any valid CSS color name (the next chapter lists CSS colors).	Sets text color.
<code>'font-weight'</code>	Any value between 100 and 900.	Sets how thick or thin characters in table text are displayed.
<code>'text-align'</code>	One of <code>'left'</code> , <code>'right'</code> , <code>'center'</code> , or <code>'justify'</code> .	Sets the horizontal alignment of text in table cells.
<code>'number-format'</code>	One of Excel's custom formats such as <code>'#,##0'</code> or <code>'#,##0 £_);(#,##0 £)'</code> .	Sets the number format using Excel formatting style.

As I mentioned earlier, besides styling an entire `DataFrame`, you can also style a subset of columns or values in your tables. The mechanism to do that is similar to applying a custom function to the columns (or rows) of a `DataFrame`: you must first define a styling function and then `apply` it to your `DataFrame`.

Let's make values in the '`Quantity`' column of `sales_df` green if they're above the average value in that column and red otherwise. To do that, you need to define a custom function that accepts a

`Series` as its only argument and returns a list of style properties (one for each value in the `Series`):

```
In [11]: def color_quantity_column(values):
    colors = []

    for quantity in values:
        if quantity > values.mean():
            colors.append('color: green')
        else:
            colors.append('color: red')

    return colors
```

The function above goes through a `Series` (i.e., the `values` argument) and constructs a list of strings (i.e., either `'color: red'` or `'color: green'`) with the same number of items as `values`. You can apply the function and style the `'Quantity'` column by running:

```
In [12]: sales_df.style.apply(color_quantity_column, subset=['Quantity'])
```

In your notebook you should now see values in the `'Quantity'` column in green or red text (depending on whether they're above or below the average quantity). You can set multiple styles at the same time, not just text color. For instance, you can extend `color_quantity_column` with:

```
In [13]: def color_quantity_column(values):
    colors = []

    for quantity in values:
        if quantity > values.mean():
            colors.append('color: green; font-weight: 900; font-size: 12pt')
        else:
            colors.append('color: red; text-align: left')

    return colors
```

Similarly, you can set cell background color in the `'Total'` column with another function:

```
In [14]: def color_total_column(values):

    green = 'background-color: green'
    red = 'background-color: red'

    return [green if v > 100 else red for v in values]
```

The function above is similar to `color_quantity_column`, but sets the `'background-color'` styling property and uses a list comprehension to construct the list of colors instead of a `for` loop. You can now apply both styling functions on `sales_df` by running:

```
In [15]: (sales_df
          .style
          .apply(color_quantity_column, subset=['Quantity'])
          .apply(color_total_column, subset=['Total'])
      )
```

The output should look similar to figure 29.1 below.

	InvoiceNo	Channel	Product Name	ProductID	Account	AccountNo	Date	Deadline	Currency	Unit Price	Quantity	Total
0	1532	Shoppe.com	Cannon Water Bomb Balloons 100 Pack	T&G/CAN-97509	Sales	5004	2020-01-01 00:00:00	11/23/19	USD	20.110000	14	281.540000
1	1533	Walcart	LEGO Ninja Turtles Stealth Shell in Pursuit 79102	T&G/LEG-37777	Sales	5004	2020-01-01 00:00:00	06/15/20	USD	6.700000	1	6.700000
2	1534	Bullseye	nan	T&G/PET-14209	Sales	5004	2020-01-01 00:00:00	05/07/20	USD	11.670000	5	58.350000
3	1535	Bullseye	Transformers Age of Extinction Generations Deluxe Class Autobot Drift Figure	T&G/TRA-20170	Sales	5004	2020-01-01 00:00:00	12/22/19	USD	13.460000	6	80.760000
4	1535	Bullseye	Transformers Age of Extinction Generations Deluxe Class Autobot Drift Figure	T&G/TRA-20170	Sales	5004	2020-01-01 00:00:00	12/22/19	USD	13.460000	6	80.760000

Figure 29.1: Styled sales data.

Like DataFrame variables, the Styler object you've been working with has a `to_excel` method you can use to save your styled table to an Excel spreadsheet. You can run the code below and open “Q1SalesStyled.xlsx” in Excel to see the same styles as above:

```
In [16]: (sales_df
          .style
          .apply(color_quantity_column, subset=['Quantity'])
          .apply(color_total_column, subset=['Total'])
          .to_excel('Q1SalesStyled.xlsx', index=False)
      )
```

This brief walkthrough showed you the basics of styling your tables, but there are other styling properties and methods you can use. If you need color in your spreadsheets, head over to pandas's website¹ to learn more about its styling options.

1: At pandas.pydata.org/pandas-docs/stable/user_guide/style.html.

Summary

This chapter showed you how to change the display and style of your DataFrame variables. This chapter also marks the end of our pandas tour. The following part of the book shows you how to turn a DataFrame into a plot using some of Python's most popular data visualization libraries.

VISUALIZING DATA

PART THREE

This part of the book looks at turning data into plots.¹ Plots often carry the weight of arguments in your documents or presentations — knowing how to make them attractive and convincing is surprisingly important in business.

Both Excel and Python² allow you to turn tables into plots. While Excel offers sensible styles and options for making plots with one click, Python allows you to craft and customize your plots down to the smallest details.

The Python ecosystem has many different libraries for data visualization, and it can sometimes be frustrating to find the right one for your needs.³ In the following chapters, I'll introduce you to Python's most versatile and popular visualization library: `matplotlib` — and a few others that extend it. `Matplotlib` is popular because it works well with other libraries from the Python ecosystem, including `pandas`, as you will soon see.

Entire books have been written about making plots with Python, so the following chapters can only briefly introduce a vast topic.⁴ Even so, they will help you understand how to make plots with code and set you up to explore Python's data visualization universe on your own.

Setup

In your Python for Accounting workspace, you'll find a folder for the third part of the book. In this folder, there are several Jupyter notebooks for each of the chapters ahead, as well as a CSV file you'll need for the plotting examples we'll go through.

The “`Q1DailySales.csv`” file contains the same sales data we've been working with so far, but grouped by channel and date. If you open the file, you'll see it has 91 rows (one for each day in the first quarter of 2020) and one column for each of the sales channels:

1: Charts, graphs, or plots are terms that loosely refer to the same concept: a graphic representation of data. While Excel uses *chart*, I use *plot* because it's more common in Python's data world.

2: Through its many data visualization libraries.

3: There are so many different visualization libraries in Python that there's a website dedicated to helping you choose which one covers your use-case: pyviz.org.

4: There's also a yearly `matplotlib` plotting competition called the *John Hunter Excellence in Plotting Contest*. Plotting can get competitive.

	Date	Understock.com	Shoppe.com	iBay.com	Walcart	Bullseye
0	2020-01-01	20707.62	6911.72	5637.54	13593.17	9179.39
1	2020-01-02	18280.59	17351.46	5959.61	12040.16	5652.32
2	2020-01-03	17191.15	10578.60	8346.60	9876.21	6127.92
3	2020-01-04	17034.69	6052.03	10168.41	12811.26	10370.95
4	2020-01-05	17074.18	11866.74	12462.30	8318.34	4641.02
..
86	2020-03-27	20183.41	5111.76	7703.85	3026.12	503.20
87	2020-03-28	8190.09	1392.89	4456.91	2776.78	1772.25
88	2020-03-29	7267.96	3966.57	6717.35	1195.16	1142.15
89	2020-03-30	14712.07	3826.95	11044.34	1702.39	676.08
90	2020-03-31	12343.37	10372.53	13687.49	157.49	848.51

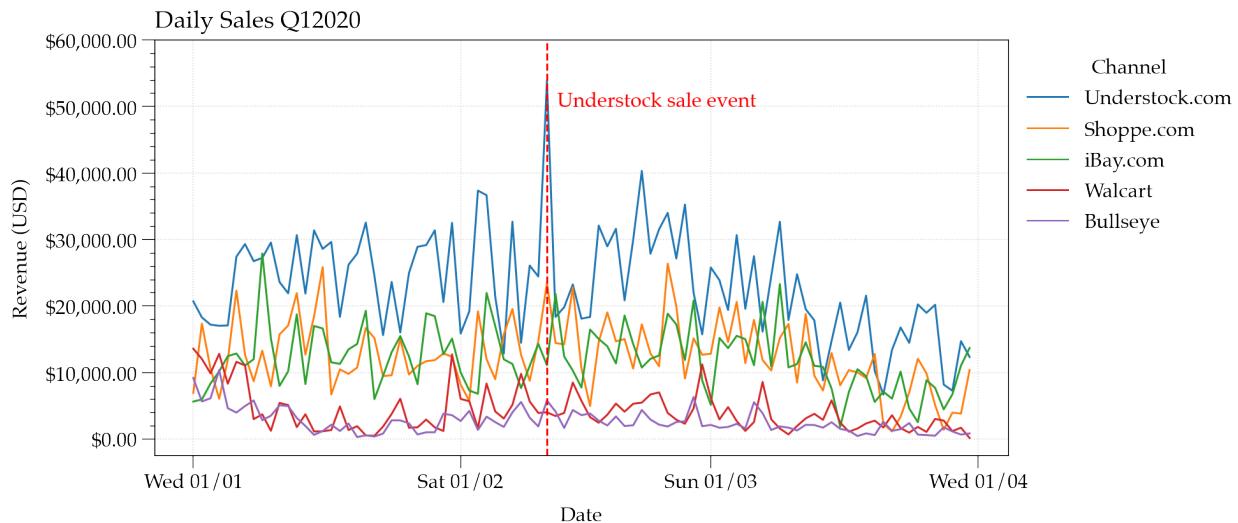
The values in this dataset represent the total daily revenue per channel for our wholesale supplier. We'll use these data to create several plots over the next chapters using Python's data visualization libraries.

When you're ready, launch JupyterLab, and let's see how you turn the revenue table above into a line plot using `matplotlib`.

Plotting with matplotlib

The `matplotlib` library¹ is central to Python's data visualization universe: there's no way around learning how to use it if you want to make plots with Python. In addition, several other visualization libraries extend `matplotlib` to provide their specialized plotting tools, so knowing how `matplotlib` works makes using those other libraries easier.

In this chapter, you'll use `matplotlib` to turn the daily sales data I introduced earlier into the line plot shown below:



Before we look at code, let's first review some of the concepts and vocabulary that `matplotlib` uses. This will help clarify what different `matplotlib` functions do later on.

Elements of a matplotlib plot

Figure 30.1 highlights the main elements of a `matplotlib` plot: on the left, you have a plot of two lines and some scatter points, and on the right, you have the same plot but with its `matplotlib` components highlighted and labeled.²

I made both plots using `matplotlib` (including the highlighted version on the right); besides introducing plot components, they serve as good examples of how detailed `matplotlib` plots can be.

Most of the highlighted plot components on the right of figure 30.1 are straightforward (i.e., a line is called a line), but there are some

1: It comes with Anaconda, so you already have it installed if you followed the setup guide in chapter 2.

2: The data used in the plots were generated for this example only and don't represent anything.

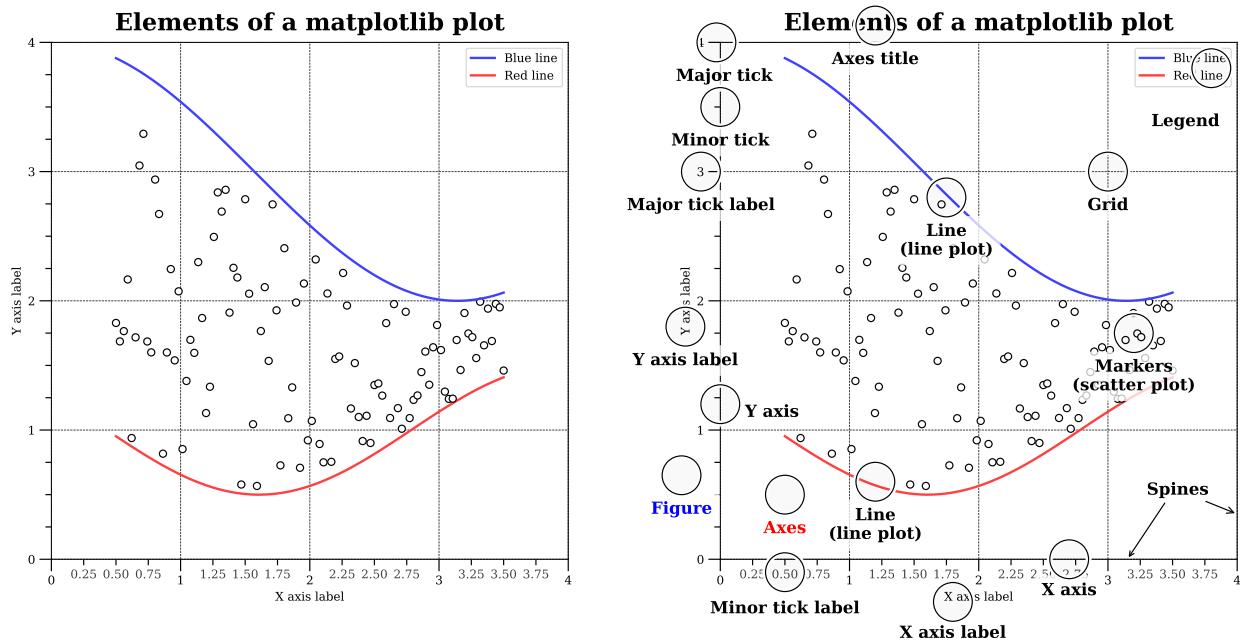


Figure 30.1: On the left, a simple plot with two lines and scattered points; on the right, the same plot with its main components highlighted and named. The plots are adapted from an example available on the official `matplotlib` website, accessible at matplotlib.org/gallery/showcase/anatomy.html.

matplotlib specifics we need to review before you start working on your plots:

- ▶ The **figure** (highlighted with a blue label) is a placeholder for everything displayed in a plot. By itself, a figure doesn't contain anything, but it can hold one or more **axes**.
- ▶ The **axes** (highlighted with a red label) represents the actual plotting area. It includes lines, points, ticks and tick labels, axes title, etc. Any data you want to visualize will be drawn within the bounds of an **axes** plot component.
- ▶ The **x-axis** and **y-axis** are the horizontal and vertical dimensions of the plot. They are part of the **axes** component, and each has associated labels, minor and major ticks, and tick labels.
- ▶ The main plot elements (in this case, the two lines and the scatter points) are collections of x and y coordinates (i.e., data points) represented visually within the bounds of the **axes**. You can represent the same collection of data points in different ways, depending on what type of plot you need (line plot, scatter plot, bar plot, etc.).

If you already use Excel to create charts, many of these plot elements are familiar to you. The methods you'll use in the following sections are named after the plot components they modify (e.g., `set_xticklabels`) and clarifying what these plot components are (and what their `matplotlib` name is) will help you understand what those methods do.

Plotting basics

To start working on the daily sales plot shown earlier, you first need to `import` the two libraries we'll be using:

```
In [1]: import pandas as pd          1
import matplotlib.pyplot as plt     2
```

You're already familiar with `pandas` — let's use it to load the daily sales dataset into a `DataFrame`:

```
In [2]: daily_sales_df = pd.read_csv('Q1DailySales.csv')
daily_sales_df['Date'] = pd.to_datetime(daily_sales_df['Date'])

daily_sales_df.head()
```

```
Out [2]:      Date  Understock.com  Shoppe.com  iBay.com  Walcart  Bullseye
0  2020-01-01        20707.62     6911.72    5637.54   13593.17    9179.39
1  2020-01-02        18280.59    17351.46    5959.61   12040.16    5652.32
2  2020-01-03        17191.15    10578.60    8346.60    9876.21    6127.92
3  2020-01-04        17034.69    6052.03   10168.41   12811.26   10370.95
4  2020-01-05        17074.18    11866.74   12462.30    8318.34    4641.02
```

The second `import` statement loads the `pyplot` submodule from `matplotlib` and assigns it the `plt` alias. The `matplotlib` library has several submodules, each with its specifics, but `pyplot` is the only one you'll need for most plots (and the only one we'll use in this chapter).

Figure and Axes

Just like `pandas`'s workhorse objects are the `DataFrame` and the `Series`, `matplotlib`'s core objects are its `Figure` and `Axes`. Every plot you make with `matplotlib` is a `Figure` object containing one or more `Axes` objects. The `Figure` object is a placeholder for everything displayed on the screen (which can include multiple axes, legends, annotations, or anything you can think of to put in a plot). You can create a `Figure` object using:

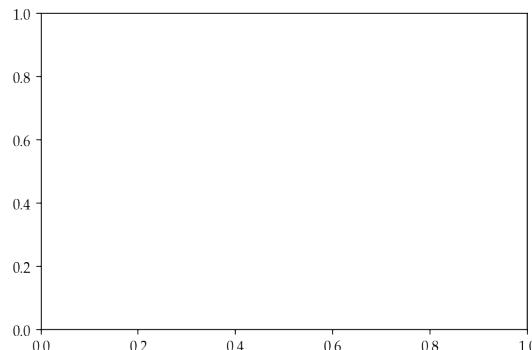
```
In [3]: fig = plt.figure()

<Figure size 432x288 with 0 Axes>
```

The `fig` variable you just created is an empty `Figure` object. By default, JupyterLab will display any `matplotlib` `Figure` object right under the code cell you create it in — here, because the `Figure` is empty, there's nothing to show yet, so JupyterLab just prints out a description of the `fig` variable.

To build up the sales plot, you need to gradually add elements to this `fig` variable. The first thing you need to add is one or more `Axes`. You can do that with:

```
In [4]: fig = plt.figure()
ax = fig.add_subplot()
```

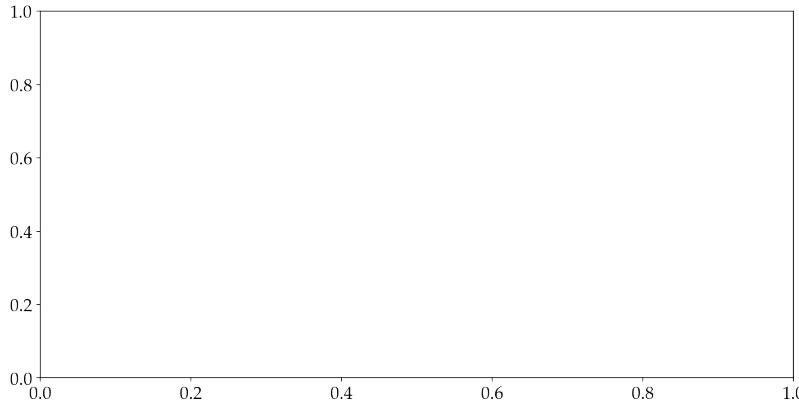


The code above creates an `Axes` object, adds it to your `Figure`, and assigns it to the `ax` variable. You can add multiple `Axes` to the same `Figure` to create side-by-side plots or complicated plot grids — we'll come back to this, for now let's continue with one plot.

The `ax` variable above is your `matplotlib` workhorse: you'll call methods on `ax` to draw points, lines, or bars from your data or to modify your plot in any other way.

By default, the entire figure will be 6.4 inches wide by 4.8 inches tall.³ If you want to change the size of your figure, you can pass the `figsize` argument when calling `plt.figure`:

```
In [5]: fig = plt.figure(figsize=(12, 6))
ax = fig.add_subplot()
```



3: And 100 dots per inch (`dpi`), in case you need to print your plots.

The first value in the tuple passed to `figsize` is the figure width, followed by its height (both in inches). Besides `figsize`, you can use a few other keyword arguments when creating a figure to customize its look — table 32.1 list some of them.

Table 30.1: Optional keyword arguments available when creating a `Figure`. Read more about the `Figure` object in `matplotlib`'s documentation at matplotlib.org/api/_as_gen/matplotlib.figure.Figure.html.

Argument	Example	Description
<code>figsize</code>	<code>plt.figure(figsize=(12, 6))</code>	Sets the figure width and height (in inches). If not provided, defaults to (6.4, 4.8).

Table 30.1: Optional keyword arguments available when creating a `Figure`. Read more about the `Figure` object in `matplotlib`'s documentation at matplotlib.org/api/_as_gen/matplotlib.figure.Figure.html.

Argument	Example	Description
<code>dpi</code>	<code>plt.figure(figsize=(12, 6), dpi=200)</code>	Sets the resolution of the <code>Figure</code> . If not provided, defaults 100. Higher resolution images use up more computer memory.
<code>frameon</code>	<code>plt.figure(frameon=False)</code>	Enables or disables <code>Figure</code> frame. By default, the frame is visible.

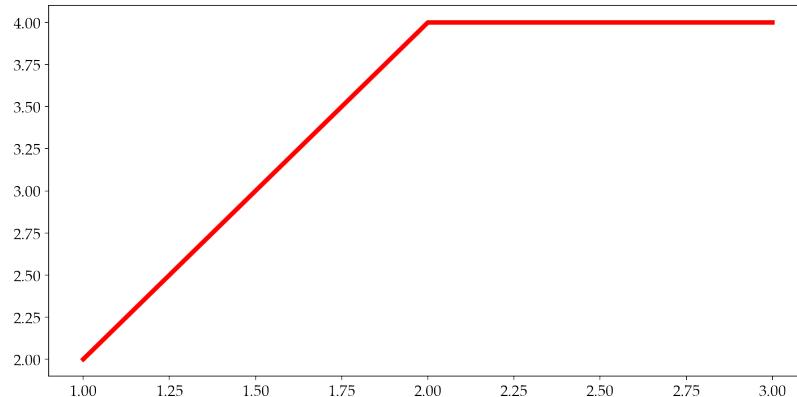
Plotting data

Now that you have a `Figure` with `Axes` set up, you need to plot some data. Depending on what type of plot you want, there are several methods you can call on the `ax` variable you created earlier. As a first example, let's create a line plot using the `plot` method:

```
In [6]: fig = plt.figure(figsize=(12, 6))
ax = fig.add_subplot()

ax.plot([1, 2, 3], [2, 4, 4], color='red', linewidth=4)
```

```
Out [6]: [<matplotlib.lines.Line2D at 0x1370055d0>]
```



The `plot` method is available only on `Axes` objects. Its first two arguments above are lists of numbers: the first sequence is a list of coordinates for the x-axis, the second a list of coordinates for the y-axis; `matplotlib` goes through each pair of coordinates, one by one, and draws a straight line from one to the next (i.e., this is what the `plot` method does). The other two arguments used above set the line color to red and line width to 4 points (by default, the line width is 1 point) — we'll go over plot colors and styles later.

The `plot` method connects all data points passed as input with a straight line. You can use several other methods to draw different kinds of plot elements from your data (besides straight lines). All these methods require a list of x-coordinates and a list of y-coordinates as their first arguments. Table 30.2 lists some of the other `Axes` methods, with examples of the plots they produce.

Table 30.2: Some of the `Axes` methods you can use to create various types of plots. Read more about all available plotting methods in the official matplotlib.org/api/axes_api.html#plotting.

Method	Example	Output	Description
<code>plot</code>	<pre>fig = plt.figure() ax = fig.add_subplot() ax.plot([1, 2, 3], [2, 4, 4], color='red')</pre>		Connects input data points with a straight line.
<code>scatter</code>	<pre>fig = plt.figure() ax = fig.add_subplot() ax.scatter([1, 2, 3], [2, 4, 4], color='purple')</pre>		Draws each input data point as a single point.
<code>bar</code>	<pre>fig = plt.figure() ax = fig.add_subplot() ax.bar(['a', 'b', 'c'], [1, 2, 3], color='maroon')</pre>		Draws each input data point as a separate bar (the top of the bar is placed at the provided y-coordinates).
<code>barh</code>	<pre>fig = plt.figure() ax = fig.add_subplot() ax.barh(['a', 'b', 'c'], [3, 2, 1], color='olive')</pre>		Same as <code>bar</code> , but makes horizontal bars instead.
<code>step</code>	<pre>fig = plt.figure() ax = fig.add_subplot() ax.step([1, 2, 3], [1, 0, 1], color='teal', linewidth=4)</pre>		Connects data points using vertical or horizontal lines.
<code>stackplot</code>	<pre>fig = plt.figure() ax = fig.add_subplot() ax.stackplot([1, 2, 3], [2, 4, 4], color='coral')</pre>		Same as <code>plot</code> , but fills the area under the line with the specified color.

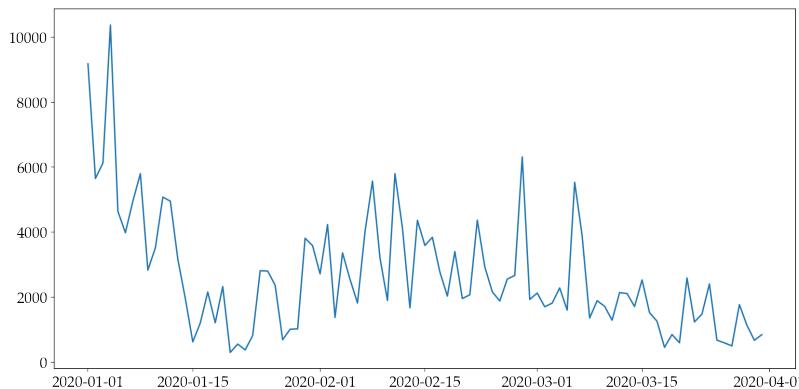
We'll continue using the `plot` method in this chapter because we want to make a line plot from the daily sales data. However, keep in mind that all the methods you'll use in the following sections (e.g., to customize plot elements, change colors or styles, or save a plot to a file) work for all plots, regardless of which `Axes` method you use to draw them.

Plotting sales data

To plot the daily sales data, you use the same `plot` method by passing a list of x and y-coordinates as arguments. To draw just one of the columns in `daily_sales_df` as a line on the plot:

```
In [7]: fig = plt.figure(figsize=(12, 6))  
ax = fig.add_subplot()  
  
ax.plot(daily_sales_df['Date'], daily_sales_df['Bullseye'])  
1  
2  
3  
4
```

```
Out [7]: [<matplotlib.lines.Line2D at 0x12eabb350>]  
5
```



You used the same kind of data with `plot` as in the previous section: a sequence of x-coordinates as the first argument (i.e., values from the `'Date'` column), and a sequence of y-coordinates for the y-axis as the second argument (i.e., values from the `'Bullseye'` column). If you don't specify a color for the line, `matplotlib` cycles through a predefined set of colors and uses a different color every time you call `plot` on the same `Axes` object.

To draw another line on the same plot, you can just use the `plot` method again:

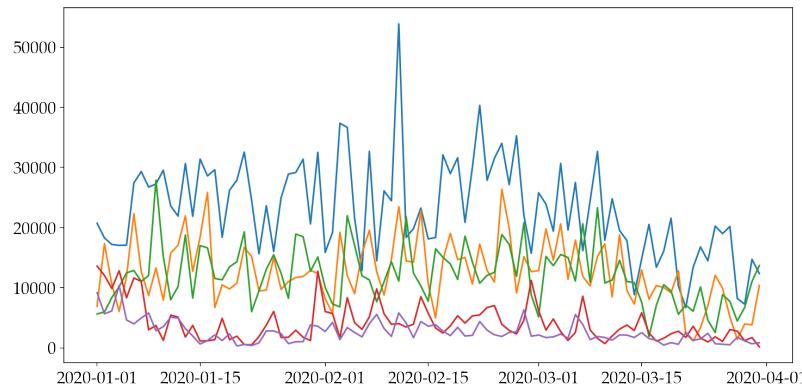
```
In [8]: fig = plt.figure(figsize=(12, 6))  
ax = fig.add_subplot()  
  
ax.plot(daily_sales_df['Date'], daily_sales_df['Bullseye'])  
1  
2  
3  
4  
ax.plot(daily_sales_df['Date'], daily_sales_df['Walcart'])  
5
```

And if you want to plot all of `daily_sales_df`, you can use the `plot` method multiple times, for each one of its columns:

```
In [9]: fig = plt.figure(figsize=(12, 6))
ax = fig.add_subplot()

ax.plot(daily_sales_df['Date'], daily_sales_df['Understock.com'])
ax.plot(daily_sales_df['Date'], daily_sales_df['Shoppe.com'])
ax.plot(daily_sales_df['Date'], daily_sales_df['iBay.com'])
ax.plot(daily_sales_df['Date'], daily_sales_df['Walcart'])
ax.plot(daily_sales_df['Date'], daily_sales_df['Bullseye'])
```

Out [9]: [<matplotlib.lines.Line2D at 0x12b513b90>]

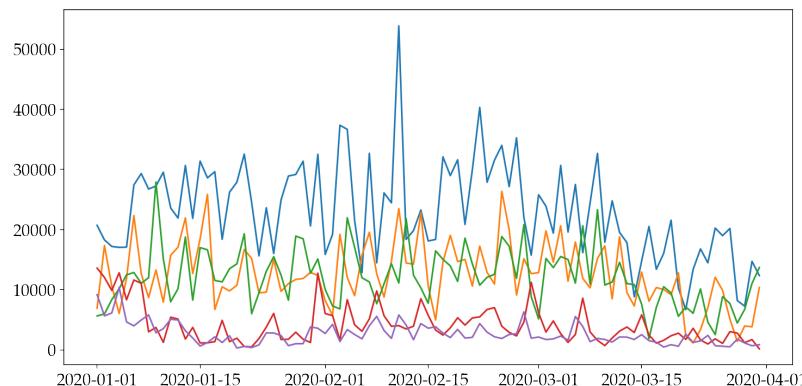


It's starting to look like a useful plot, but we still need to fix a few things: add a legend, so you know which line corresponds to which channel, add a title, and some other details. However, before we fix any of those, let's first improve the code above slightly by making it less repetitive. Because pandas works well with matplotlib, you can get the same plot as above in fewer lines of code by running:

```
In [10]: fig = plt.figure(figsize=(12, 6))
ax = fig.add_subplot()

ax.plot(daily_sales_df.set_index('Date'))
```

Out [10]: [<matplotlib.lines.Line2D at 0x120bcf6d0>,
<matplotlib.lines.Line2D at 0x135b1a610>,
<matplotlib.lines.Line2D at 0x135b1ad10>,
<matplotlib.lines.Line2D at 0x135b1ae50>,
<matplotlib.lines.Line2D at 0x135b1a9d0>]



When you pass an entire `DataFrame` to the `plot` method (as I did above), `matplotlib` plots each column in the `DataFrame` as a separate line — here, a separate line because I used the `plot` method; the same is true when you use `scatter`, `bar`, or any of the other plotting methods. `Matplotlib` uses the `DataFrame` row labels as x-coordinates and values from each column in the `DataFrame` as y-coordinates.

In this example, I wanted `matplotlib` to use values from the '`Date`' column as x-coordinates, so I set the '`Date`' column as the `DataFrame` index when calling `plot`. We'll be using dates as x-coordinates for the sales plot so you can set the '`Date`' column as `daily_sales_df`'s index for good:

```
In [11]: daily_sales_df = daily_sales_df.set_index('Date')
```

1

You probably noticed that the plotting code above produces some text output as well as an actual plot (something like `matplotlib.lines.Line2D at 0x12b513b904`). This output is just a text description of whatever the last line in a code cell *returns* — above, it is a list of `Line2D` objects, one for each line in the plot. Most often, this output isn't useful — you can suppress it by adding a semicolon (`;`) at the end of the last line in a plotting cell (which is why I'll add it at the end of code examples in the rest of this chapter).

4: This label describes a `Line` object added to the plot. While the details of how this `Line` object is created and stored are not particularly useful, you might want to know that the identifier at the end of this label is called a *memory address* and helps uniquely identify `Line2D` objects in your computer's memory.

Adjusting plot details

The sales plot you have now needs a few more things to make it easier to read: a title, x and y-axis labels, a legend. There is an `Axes` method you can use to add each of these elements to your plot — this section takes a closer look at some of these methods.

Depending on your audience and your interest in crafting plots, you may only need a small subset of the methods mentioned here. On the other hand, if you like making pixel-perfect plots, the methods discussed in this section are just a small subset of what's available, you'll have to discover the others on your own. Regardless, fine-grained control over plots is where `matplotlib` shines.

Before we start, keep in mind that most of the methods you'll be using accept keyword arguments to change the way they work (e.g., with the `set_title` `Axes` method below, you can change a plot's title placement by passing either '`left`', '`right`' or '`center`' as the `loc` keyword argument). I won't mention all available keyword arguments for each method because you can always add a `? after any method name and run that code in a separate cell to read more about it (including what arguments it accepts).`

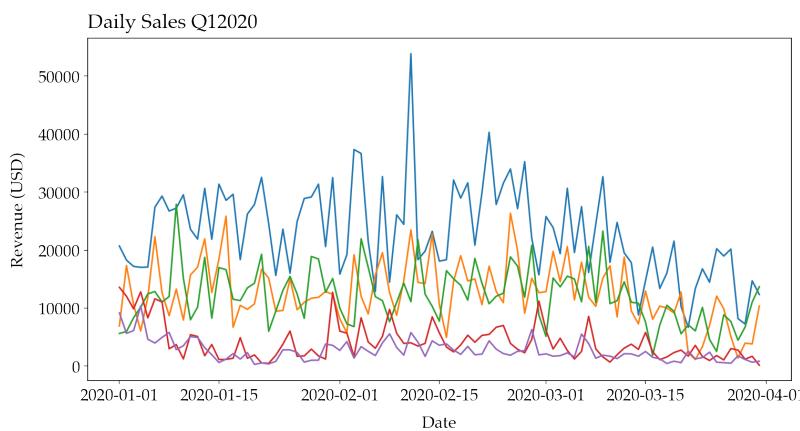
Title and axis labels

To set a title for a plot, you can use the `set_title` method. Similarly, you can set axis labels to help readers understand what the axes on your plot mean with `set_xlabel` and `set_ylabel`:

```
In [12]: fig = plt.figure(figsize=(12, 6))
ax = fig.add_subplot()

ax.plot(daily_sales_df)

ax.set_title('Daily Sales Q12020', loc='left', pad=10)
ax.set_xlabel('Date')
ax.set_ylabel('Revenue (USD)');
```



We'll take a look at how to change font style (i.e., font size and font family) for the plot title and labels later. For now, let's see how to set ticks and their labels.

Ticks and tick labels

On most plots, both the x-axis and the y-axis have several short, evenly spaced dashes that help readers understand the range of data that is being plotted. These highlighted dashes are called *ticks*, and their associated labels are called *tick labels* (Excel calls them *tick marks*).

There are two types of ticks on a plot: *major ticks*, which indicate larger (equally spaced) intervals on an axis and are more prominent, and *minor ticks*, which indicate smaller intervals between major ticks and are typically not labeled (figure 30.2 in the margin shows the different types of ticks on a `matplotlib` plot).

Right now, the x-axis tick labels on our sales plot are not particularly easy to read. By default, `matplotlib` will try and place ticks and their labels as evenly spaced and nicely formatted as possible. However, if you want to change the location of ticks on the x-axis, you can use the `Axes set_xticks` method and pass a list of values

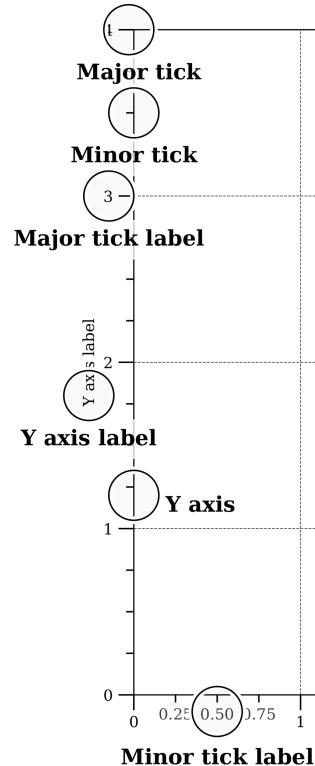


Figure 30.2: Fragment from *Elements of a matplotlib plot* highlighting major and minor ticks and their labels.

at which to place the ticks (in our case, a list of dates). You can define a list of dates any way you want to — you can even manually type `Timestamp` values for each tick if you want to be very specific about your tick placement:

```
In [13]: # an easier way to get this list of dates
# is by using pandas's date_range function
# pd.date_range('01 January 2020',
#               '1 April 2020', freq='MS')

xtick_values = [
    pd.Timestamp('2020-01-01'), pd.Timestamp('2020-02-01'),
    pd.Timestamp('2020-03-01'), pd.Timestamp('2020-04-01')
]
```

The `xtick_values` variable is a list of dates representing the first of each month between January and April 2020 (I used pandas's top-level `date_range` function⁵ to create this list and copied the values).

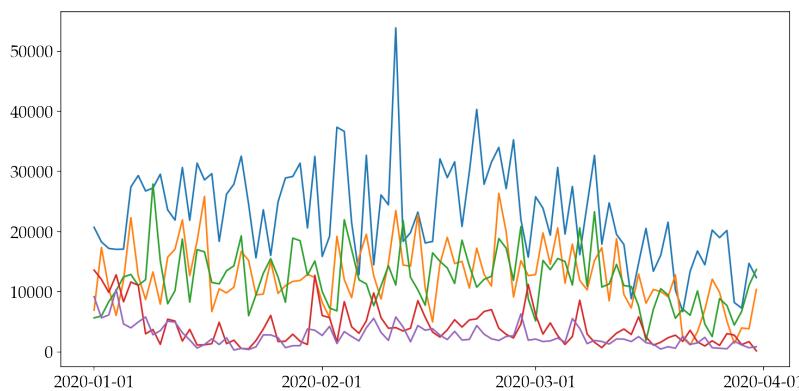
```
In [14]: fig = plt.figure(figsize=(12, 6))
ax = fig.add_subplot()

ax.plot(daily_sales_df)

xtick_values = [
    pd.Timestamp('2020-01-01'), pd.Timestamp('2020-02-01'),
    pd.Timestamp('2020-03-01'), pd.Timestamp('2020-04-01')
]

ax.set_xticks(xtick_values)
```

5: You can read more about `date_range` at pandas.pydata.org/pandas-docs/stable/reference/api/pandas.date_range.html.



To change the labels associated with each tick, you need to create another list with the labels you want to use and pass it to the `set_xticklabels` method. The easiest way to make a list of labels is by using the date values in `xtick_values` and a Python list comprehension:

```
In [15]: xtick_labels = [date.strftime('%a %d/%m') for date in xtick_values]

xtick_labels
```

```
Out [15]: ['Wed 01/01', 'Sat 01/02', 'Sun 01/03', 'Wed 01/04']
```

The output is just a list of strings, but if the code above seems strange, head back to chapter 6 for a refresher on Python list comprehensions. Inside the list comprehension, the `strftime` method is called on each date value in `xtick_values` to convert it to a string. The format specifiers passed to `strftime` (i.e., '`%a %d/%m`') are the same specifiers mentioned in chapter 22 (I never remember what these format specifiers do, so if you're like me, you can see all of them explained at strftime.org).

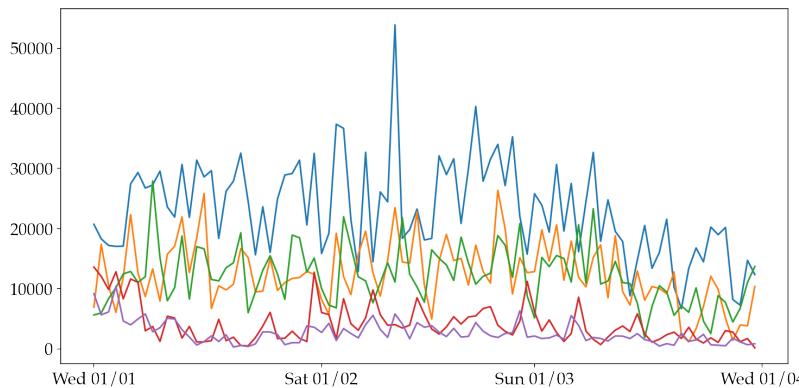
Putting all the steps together, you can set the ticks and tick labels on the sales plot using the following code:

```
In [16]: fig = plt.figure(figsize=(12, 6))
ax = fig.add_subplot()

ax.plot(daily_sales_df)

xtick_values = pd.date_range('01 January 2020', '1 April 2020', freq='MS')
xtick_labels = [date.strftime('%a %d/%m') for date in xtick_values]

ax.set_xticks(xtick_values)
ax.set_xticklabels(xtick_labels);
```

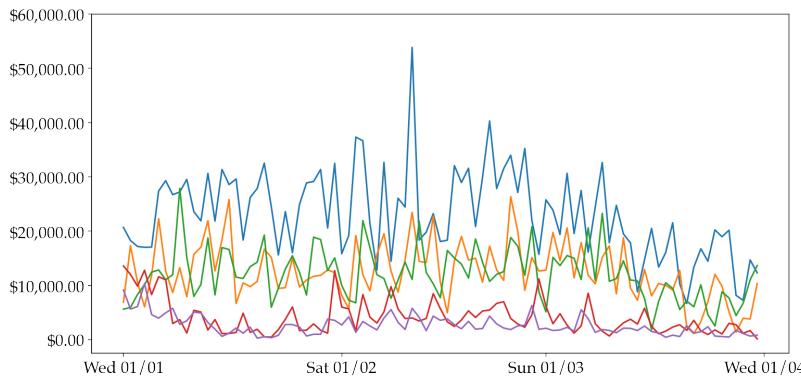


Note that axis limits will change depending on the ticks you set (i.e., if you set ticks all the way to December, `matplotlib` will widen the x-axis range to show all of them).

You can use the same approach to format and position the y-axis ticks. In this case, because we are plotting numerical values on the y-axis (i.e., not dates), you need to use a list of numbers with the `set_yticks` method. The code below extends the previous example (i.e., you need to type it in the same cell as the code above):

```
ytick_values = range(0, 60001, 10000)
ytick_labels = [f'${value:.2f}' for value in ytick_values]

ax.set_yticks(ytick_values)
ax.set_yticklabels(ytick_labels);
```



This code follows the same logic as with the x-axis ticks: you create a list of numbers using the Python `range` function (which takes as arguments a start value, an end value, and a step) and a list of labels created using the tick values. You then set the y-axis ticks and their labels with the `set_yticks` and `set_yticklabels` methods.

To set *minor* ticks (or their labels) on either axis, you can pass the `minor=True` keyword argument to the same tick methods. Extending the previous examples further, you can add minor ticks on the y-axis of the sales plot using:

```
ytick_minor_values = range(0, 60001, 2000)
ax.set_yticks(ytick_minor_values, minor=True);
```

16

17

You can customize ticks further, by using the `ax.tick_params` method. For example, to help readers easily differentiate between minor and major ticks, you can set tick length using:

```
ax.tick_params(axis='both', which='major', length=10)
ax.tick_params(axis='y', which='minor', length=4)
```

18

19

There are many different ways to customize your plot ticks and their associated label (including color, rotation, padding, and others) — check `ax.tick_params?` in a separate code cell in your notebook.

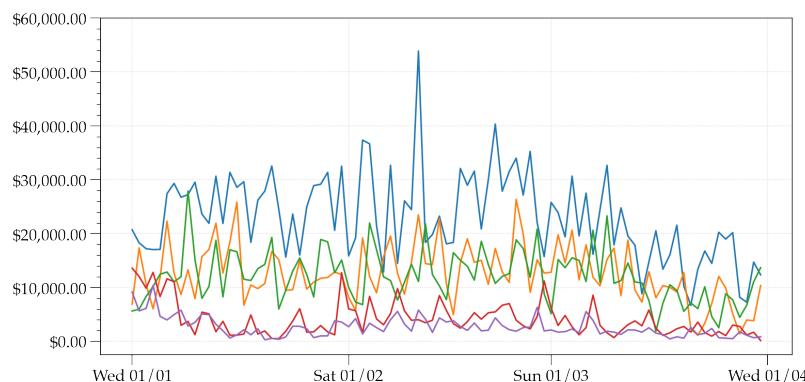
Finally, you can extend the axis ticks into your plot's main area by adding a grid. Plot grids are useful in guiding readers when reading values on your plots. To add a grid, you can use the `grid` Axes method:

```
ax.grid(axis='both', which='major', linestyle='dotted', alpha=0.5);
```

20

In this example, you specify which axis ticks to extend into a grid (here, major ticks from both the x and the y axis; '`x`' or '`y`' are also valid options for the `axis` keyword argument), and set the line style to `'dotted'` (also valid are `'solid'` or `'dashed'`). The `alpha` keyword argument sets the transparency of the grid lines (with 0 being fully transparent and 1 being fully opaque) — I often make grid lines transparent in my plots so they don't distract readers from the main visual elements.

The result after customizing ticks, their labels, and adding a plot grid is shown below:

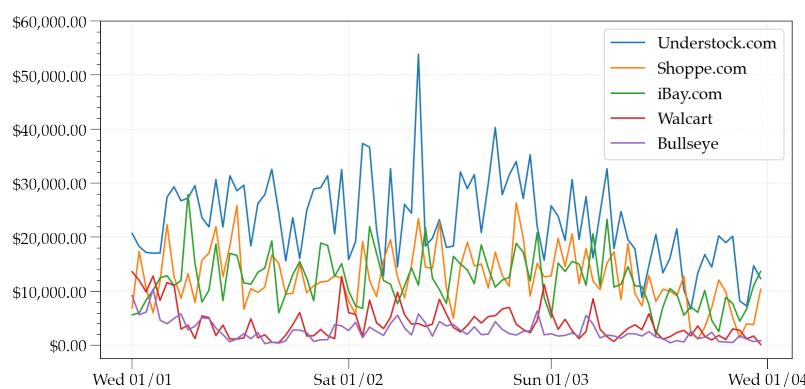


Legend

The sales plot right now doesn't have a legend that explains which line corresponds to which sales channel. To add one, you can use the `legend` `Axes` method and pass a list of labels (i.e., string values) that you want to associate with each line on the plot. Because we used `daily_sales_df` to plot all the lines at once, you can use `daily_sales_df.columns` as the legend labels. Extending the previous examples, you can add a legend with:

```
ax.legend(daily_sales_df.columns);
```

21



By default, `matplotlib` will try to place the legend in the plot area that has the most empty space. You can change this location by passing the `loc` keyword argument, and specifying one of the predefined locations listed in the margin on the right as a value:

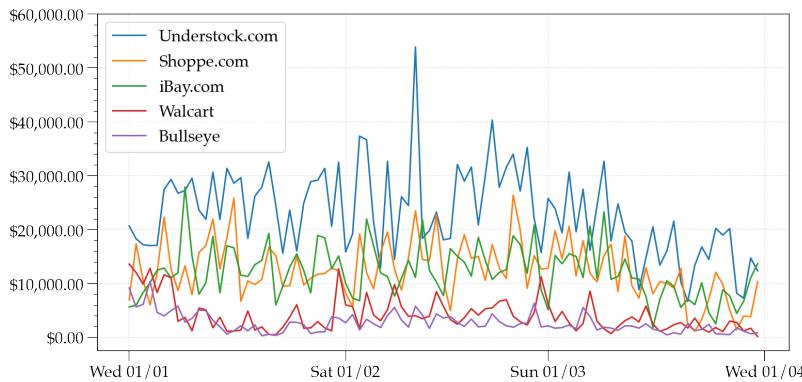
```
ax.legend(daily_sales_df.columns, loc='upper left');
```

21

Predefined legend locations you can use with the `loc` keyword argument:

- 'best'
- 'upper right'
- 'upper left'
- 'lower left'
- 'lower right'
- 'right'
- 'center left'
- 'center right'
- 'lower center'
- 'upper center'
- 'center'

By default, `matplotlib` places the legend at the '`best`' location (i.e., the area of the plot with most empty space).

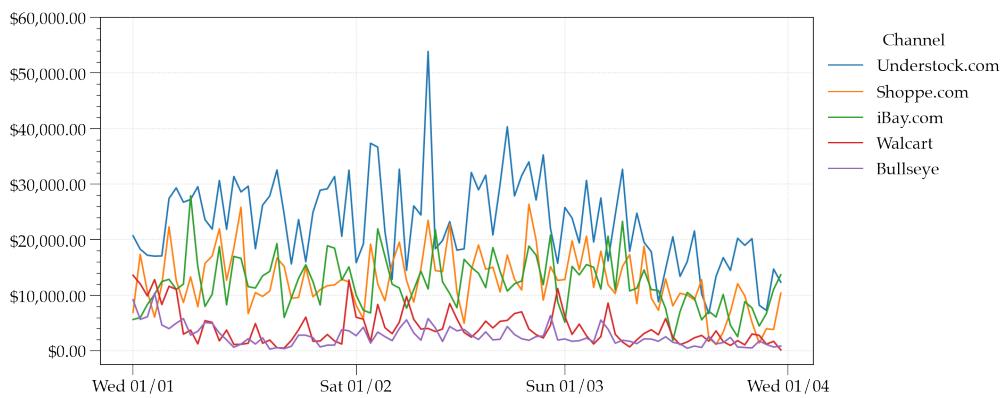


If you want to move the legend outside of the main plot area, you can use the `bbox_to_anchor` keyword argument to specify a point on the plot to use as an anchor for the legend, together with the `loc` keyword argument. When using `bbox_to_anchor`, the `loc` keyword argument specifies which corner of the legend to place at the anchor point. For example, to move the legend outside of the axes, the following code places the upper left corner of the legend at the upper right corner of the plots:

```

ax.legend(
    daily_sales_df.columns,
    title='Channel', frameon=False,
    loc='upper left', bbox_to_anchor=(1, 1)
);

```



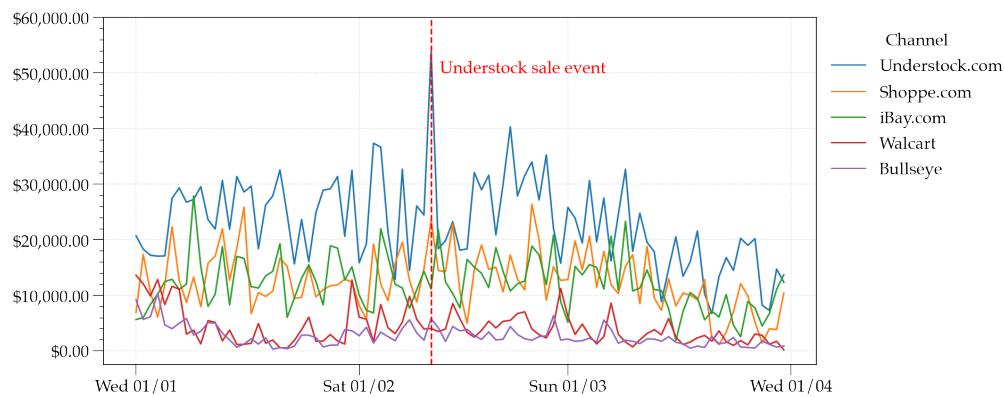
The coordinates for the anchor point passed to `bbox_to_anchor` are relative to each axis, meaning that `(0, 0)` corresponds to the lower left corner of the main plot area (i.e., the origin of the axes), and `(1, 1)` corresponds to the upper right corner.

In addition to positioning the legend outside the axes, in this example, you also set the legend title (using the `title` keyword argument). You also removed the frame around the legend (using the `frameon=False` argument) — run `ax.legend?` in a separate cell to find more legend configuration options.

Annotations

The last element we need to add to the sales plot is a vertical line marking a sales event on the 11th of February, as well as some text describing what this vertical line is. To add a vertical line to the plot, you can use the `ax.axvline` method, and to add a text annotation to the plot, right next to the line, you can use the `ax.annotate` method:

```
event_date = pd.Timestamp(2020, 2, 11) 26
27
ax.axvline(event_date, linestyle='dashed', color='red')
ax.annotate(' Understock sale event', xy=(event_date, 50000), color='red'); 28
29
```



The `ax.axvline` method draws a vertical line that spans the plot's entire height at a certain x-coordinate (similarly, you can use `ax.axhline` to draw a horizontal line). Because we are using dates on the x-axis, on line 6 you create a `pd.Timestamp` variable for the sales event date, and then use that variable as the x-coordinate for the vertical line drawn with `ax.axvline`.

To add a text label at an arbitrary position on the plot, you can use `ax.annotate` and specify what text you want to add to the plot, as well as the x and y-coordinates for the label.

As with any other plot element, there are several keyword arguments you can use with both `ax.axvline` and `ax.annotate` (in the example above, you set the `color` of both elements to `'red'`) — as always, run `ax.axvline?` or `ax.annotate?` in a separate cell to find out more.

The complete sales plot

The entire code needed to create the sales plot shown at the beginning of the chapter is listed below:

```
In [25]: fig = plt.figure(figsize=(12, 6))
ax = fig.add_subplot()

# Plot data
ax.plot(daily_sales_df)

# Plot title
ax.set_title('Daily Sales Q12020', loc='left', pad=10)

# Axis labels
ax.set_xlabel('Date')
ax.set_ylabel('Revenue (USD)')

# Add legend
ax.legend(
    daily_sales_df.columns,
    title='Channel', frameon=False,
    loc='upper left', bbox_to_anchor=(1, 1)
)

# Add x-ticks and x-tick labels
xtick_values = pd.date_range('1 January 2020', '1 April 2020', freq='MS')
xtick_labels = [date.strftime('%a %d/%m') for date in xtick_values]

ax.set_xticks(xtick_values)
ax.set_xticklabels(xtick_labels)

# Add y-ticks and y-tick labels
ytick_values = range(0, 60001, 10000)
ytick_labels = [f'${value:,.2f}' for value in ytick_values]

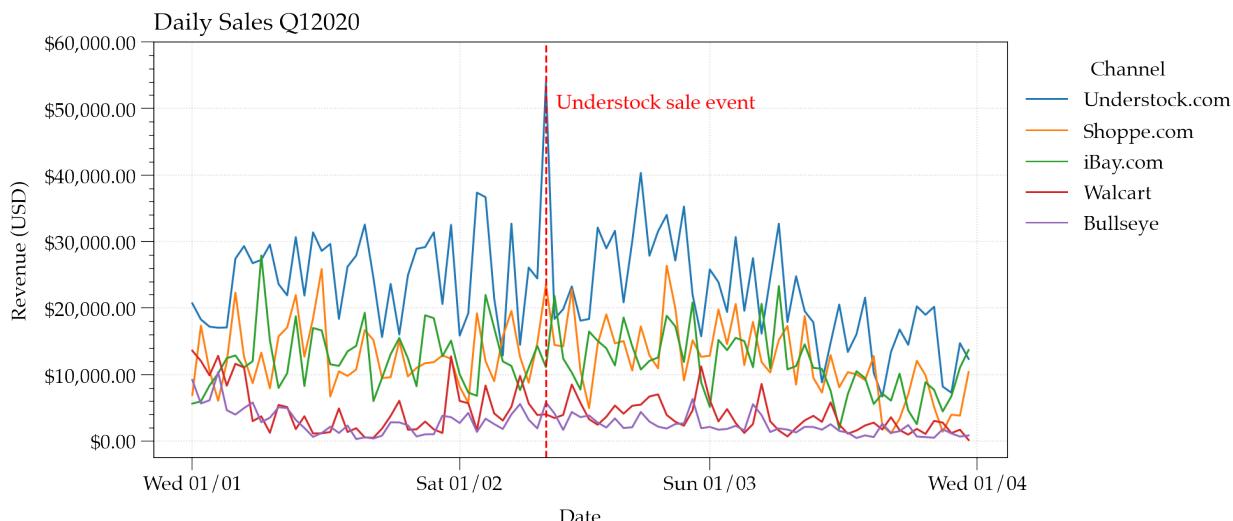
ax.set_yticks(ytick_values)
ax.set_yticklabels(ytick_labels)

yticks_minor_values = range(0, 60001, 2000)
ax.set_yticks(yticks_minor_values, minor=True)

# Configure tick length
ax.tick_params(axis='both', which='major', length=10)
ax.tick_params(axis='y', which='minor', length=4)

# Add grid
ax.grid(axis='both', which='major', linestyle='dotted', alpha=0.5);

# Add sales event annotation
event_date = pd.Timestamp(2020, 2, 11)
ax.axvline(event_date, linestyle='dashed', color='red')
ax.annotate('Understock sale event', xy=(event_date, 50000), color='red');
```



It may seem like a lot of code for a relatively simple plot. But after you write this code once, you can reuse it later in any similar plot you create — and of course, you don’t have to use all the plot customization options we went through if you don’t need them.

Overthinking: Saving a plot

To save a `matplotlib` plot, you can right-click it in JupyterLab while holding the `Shift` key. This will bring up your browser’s contextual menu, where you can copy or save the plot as an image file — which is useful when you need to copy plots into PowerPoint presentations or other documents quickly.

Another option for saving your plots is to use the `savefig` method on the `fig` variable (**not** the `ax` variable). For instance, you can save the plot above by calling:

```
In [26]: fig.savefig('Q1DailySales.png')
```

The first argument to `savefig` is a file name (by default, the image file is saved in the same folder as your Jupyter notebook, but you can specify a full path instead if you want to). The `savefig` method is handy when you need to create multiple plots in a `for` loop and save them to your drive (without having to right-click and save each one manually).

Overthinking: Styles, colors, and fonts

The sales plot example shows you how to add content to your `matplotlib` plots, but what if you want to change their style: different background color, larger fonts, more padding around labels? This overthinking section walks you through some of `matplotlib`’s styling options.

There are several predefined plot styles available with `matplotlib` — the default style is the one you saw used in the sales plot: white background, framed plot, thin lines, etc. These predefined styles try to make sensible plot design choices for you (they are somewhat similar to chart styles in Excel). You can list all the available `matplotlib` style names by running:

```
In [27]: import matplotlib.pyplot as plt
plt.style.available
```

```
Out [27]: ['default',
'classic',
'fivethirtyeight',
...
'ggplot',
'grayscale',
'seaborn']
```

Figure 30.3 shows a few examples of plots using the `default`, `ggplot` and `seaborn` styles (the style names are set as the y-label in each row of plots in the figure) — you can see more examples of plots using different styles in `matplotlib`'s figure gallery.⁶ The differences between plot styles can be subtle but typically involve different values for many of the same plot options you set in the sales plot code (e.g., tick frequency, line colors, grid style, etc.).

6: At matplotlib.org/gallery/style_sheets/style_sheets_reference.

If you want to use a different plot style, you should select it right

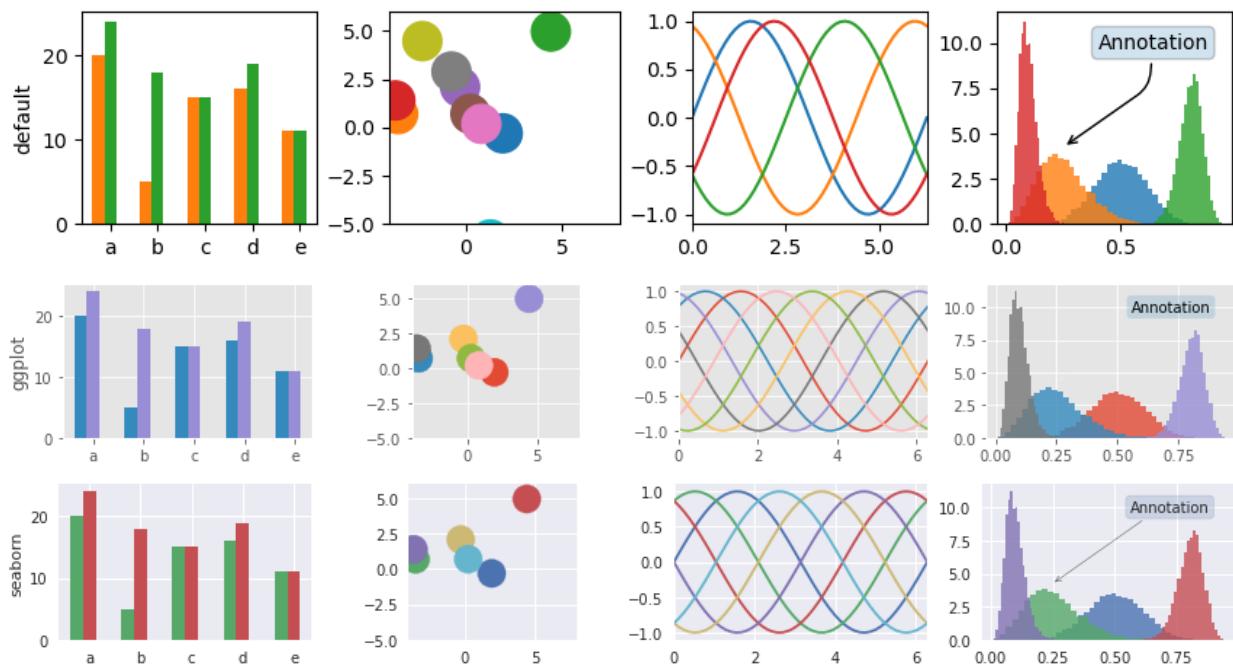


Figure 30.3: Several plot examples using the `default`, `ggplot` and `seaborn` `matplotlib` styles. Each row of plots uses a different style. You can see more examples using different `matplotlib` styles in the figure gallery available at matplotlib.org/gallery/style_sheets/style_sheets_reference.

after importing `matplotlib` so that all the plots in your notebook have the same look. You can choose a different plot style with:

```
In [28]: import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
```

Figure 30.4 below shows what the sales plot you created earlier would look like after selecting the '`fivethirtyeight`' style.⁷

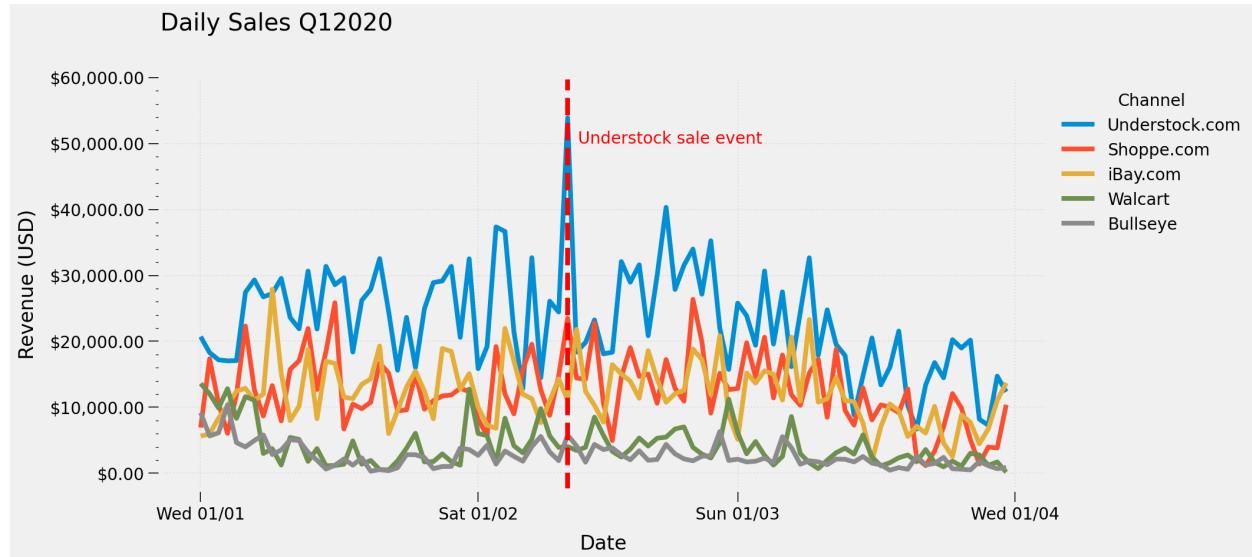


Figure 30.4: Sales plot using the '`fivethirtyeight`' `matplotlib` plot style.

Modifying styles

If you find yourself manually setting the same options every time you create a plot (e.g., setting major tick length to 10), you may want to update the plot style you're using instead of calling the same `Axes` methods every time you create a new plot.

Each `matplotlib` style is a collection of parameters that define how various parts of your plot looks like (i.e., a dictionary of key-value pairs, where keys are plot design options, like tick length or background color, and their associated values determine what your plots look like). When you select a different plot style, `matplotlib` uses another collection of styling parameters. However, you can manually change the value associated with any of these parameters as well.

You can list *all* available plot parameters for the current style, and their values, by running the code below (the output will be long, as there are over 300 different plot parameters you can customize; I shortened the output here to make it fit).⁸

```
In [29]: plt.rcParams
```

7: *Fivethirtyeight* is a website that publishes data analysis articles about politics, economics, and sports. This `matplotlib` style tries to reproduce the look of plots shown on that website.

8: You can read about each parameter (and what values you can assign to it) in `matplotlib`'s documentation at matplotlib.org/tutorials/introductory/customizing.

```
Out [29]: RcParams({'axes.axisbelow': 'line',
                     'axes.edgecolor': 'black',
                     'axes.facecolor': 'white',
                     'axes.grid': False,
                     'axes.grid.axis': 'both',
                     'axes.grid.which': 'major',
                     'axes.prop_cycle': cycler('color', ['#1f77b4',
... , '#17becf']),
                     ...
                     'ytick.major.left': True,
                     'ytick.major.pad': 3.5,
                     'ytick.minor.left': True,
                     'ytick.minor.pad': 3.4,
                     'ytick.minor.visible': False,
                     'ytick.minor.width': 0.6})
```

The output above is a `matplotlib`-specific dictionary containing key-value pairs for all plot parameters and their associated values for your current plot style. You can use it like a Python dictionary to access different parameter values:

```
In [30]: plt.rcParams['axes.labelcolor']
```

```
Out [30]: 'black'
```

Or to set parameter values:

```
In [31]: plt.rcParams['legend.labelspacing'] = 2
```

In the example above, you set the spacing between items in a plot legend to 2 points. After running this code, all `matplotlib` plots you create in your notebook will have 2-point legend item spacing.

Because there are so many parameters, it can be tricky to find the ones you want to modify. But `plt.rcParams` works like a Python dictionary: you can filter its keys based on a certain term to find the parameters that you need. For example, to find all parameter names that include the word '`title`', you can run:

```
In [32]: [(key, value) for key, value in plt.rcParams.items() if 'title' in key]
```

```
1
```

```
Out [32]: [('axes.titlecolor', 'auto'),
            ('axes.titlelocation', 'center'),
            ('axes.titlepad', 6.0),
            ('axes.titlesize', 'x-large'),
            ('axes.titleweight', 'normal'),
            ('figure.titlesize', 'large'),
            ('figure.titleweight', 'normal'),
            ('legend.title_fontsize', None)]
```

It can also be tricky to figure out the values you can set for different parameters. However, if you set an invalid value for a plot parameter, you'll get an error, and most error messages include a

list of valid options for the parameter you tried to set. For instance, if you try to set `'axes.titlesize'` to an invalid option:

```
In [33]: plt.rcParams['axes.titlesize'] = 'not too small, not too big'
```

`ValueError`

Traceback (most recent call last)

...

`ValueError: Key axes.titlesize: not too small, not too big is not a valid font size. Valid font sizes are xx-small, x-small, small, medium, large, x-large, xx-large, smaller, larger.`

You can use the same approach to set any of the styling parameters listed when running `plt.rcParams`⁹ — and you can set as many as you need to. Remember that styling options set using this approach will be applied to *all* plots in your notebook (so it's a good idea to configure styling options right after importing `matplotlib` if you want to make your plots share a common look).

If you need to reset all styling options to their default values, you can either run the method below, or remove all styling code from your notebook, and then restart your notebook kernel.

```
In [34]: plt.rcParams()
```

In some cases, it may be easier to set plot styling options by modifying the `rcParams` dictionary rather than using the `Axes` methods from the previous section. However, many `matplotlib` code examples you will come across online use `ax` methods, and knowing what the `ax` methods are called and what they do will help you navigate `matplotlib` examples. Whichever styling approach you end up using, plot contents (e.g., title, axis labels, legend entries) can only be set using the `ax` methods you used earlier.

Colors

Many of the `ax` methods allow you to specify a `color` keyword argument to change the color of a plot element (e.g., `ax.plot([1, 2, 3], [1, 2, 3], color='red')`). Some of the colors you can choose from are shown in the figure on the right — you can follow the link in the caption to see all available colors.

When you don't specify a value for the `color` keyword argument, `matplotlib` cycles through a predefined list of colors and uses a different color for each new plot element.

The list of default colors can be explicitly defined using `rcParams`, like the other styling parameters mentioned earlier. However, the code to do that is slightly odd — you need to `import` a `cycler` submodule; to set a different list of default colors for your current plot style, you can use:

9: The `rcParams` name comes from *run configuration parameters*. The `rc` letters appearing at the beginning or the end of a file name or variable name typically mean that the file or variable store some configuration options.



Figure 30.5: Color names you can use with `matplotlib`. A complete list of colors is available in `matplotlib`'s documentation at matplotlib.org/gallery/color/named_colors.

```
In [35]: from cycler import cycler
        1
        2
plt.rcParams['axes.prop_cycle'] = cycler(
        3
    'color',
        4
    ['red', 'blue', 'gray']
        5
)
        6
```

After you run this code, `matplotlib` will cycle through the list of '`red`', '`blue`', and '`gray`' whenever it adds a new element to a plot (i.e., whenever you call an `Axes` method to add a new line, bar, or point to a plot, without explicitly specifying a `color`). When it runs out of colors, it starts over from the beginning.

Fonts

In addition to plot style parameters, you can use `rcParams` to set different font options. For the sales plot examples in the previous section, I used the following font settings:

```
In [36]: import matplotlib.pyplot as plt
        1
        2
plt.rcParams['font.family'] = 'serif'
        3
plt.rcParams['font.serif'] = 'Palatino'
        4
plt.rcParams['font.size'] = 18
        5
plt.rcParams['font.weight'] = 'normal'
        6
plt.rcParams['font.style'] = 'normal'
        7
```

If you run this code, you set all plots in your notebook to use a serif font called *Palatino*, with a base size of 18 points, normal weight, and style. All text elements in your plots are scaled using the base font size set here. You can list all font-related plot parameters with:

```
In [37]: [(key, value) for key, value in plt.rcParams.items() if 'font' in key]
```

To get a list of fonts available on your computer, you can use the following code (which outputs a long list):

```
In [38]: import matplotlib
        1
        2
matplotlib.font_manager.fontManager.ttflist
        3
Out [38]: [<Font 'DejaVu Sans' (DejaVuSans-Bold.ttf) normal normal 700 normal>,
        1
    <Font 'cmmi10' (cmmi10.ttf) normal normal 400 normal>,
        2
    <Font 'STIXNonUnicode' (STIXNonUniBolIta.ttf) italic normal 700 normal>,
        3
    ...
        4
    <Font 'Arial Black' (Arial Black.ttf) normal normal 900 normal>,
        5
    <Font 'STIXSizeTwoSym' (STIXSizTwoSymBol.otf) normal normal 700 normal>,
        6
    <Font '.SF NS Display Condensed' (SFNSDisplayCondensed.otf) normal normal 400 condensed>]
```

Summary

This chapter showed you how to turn a `DataFrame` into a `matplotlib` plot. We covered some of `matplotlib`'s most useful features, as well as some of its more hidden styling options.

There are many functions and arguments you can use with `matplotlib`, and they can often be frustrating to figure out. However, just knowing what `matplotlib`'s `Figure` and `Axes` objects are and how to use a few of their methods is enough to get you started with Python-based data visualization. The following project chapter shows you how to use `matplotlib`'s main features to turn a cash-flow statement into a waterfall plot.

Project: Making a waterfall plot from a cash flow statement

31

In the previous chapter, you saw that `matplotlib` is a data visualization toolkit more than a gallery of plot templates you can re-use. However, you can easily use it to build up any plot from scratch.

This chapter shows you how to use `matplotlib` and `pandas` to turn a cash flow statement into a waterfall plot. Waterfall plots¹ are useful for showing how an initial value is affected by a series of interventions. The data you'll be using is the “*Cash Flow Statement.xlsx*” file in the “*P3 - Visualizing data*” folder of your *Python for Accounting* workspace. When you open this file with Excel, you should see something similar to figure 31.1 below:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA
1																										
2																										
CASH FLOW Statement																										
3																										
4																										
5	Beginning Cash Balance	0	\$23,482		\$39,210		(\$1,455)		(\$17,015)		(\$10,782)		(\$61,919)		(\$36,516)		(\$24,988)		\$41,925		\$30,167		\$110,895			
6	Cash Inflows (Income):																									
7	Cash Collections	\$16,858	\$19,123		\$5,557		\$45,318		\$207		\$2,300		\$14,719		\$3,794		\$18,737		\$4,583		\$40,231		\$27,565		\$198,992	
8	Credit Collections	\$31,157	\$8,484		\$8,314		\$1,828		\$8,253		\$4,570		\$12,637		\$44,026		\$35,364		\$10,836		\$12,001		\$1,255		\$178,725	
9	Investment Income	\$10,449	\$24,453		\$3,860		\$3,263		\$23,606		\$3,850		\$44,656		\$21,898		\$32,910		\$5,323		\$46,416		\$33,766		\$254,450	
10	Other:	\$20,788	\$23,450		\$3,412		\$2,009		\$24,076		\$1,143		\$5,967		\$10,566		\$34,263		\$39,473		\$29,935		\$13,826		\$208,908	
11																										0
12																										
13	Total Cash Inflows	\$79,252	\$75,510		\$21,143		\$52,418		\$56,142		\$11,863		\$77,979		\$80,284		\$121,274		\$60,215		\$128,583		\$76,412		\$841,075	
14	Available Cash Balance	\$79,252	\$98,992		\$60,353		\$50,963		\$39,127		\$1,081		\$16,060		\$43,768		\$98,286		\$102,140		\$158,750		\$187,307			
15	Cash Outflows (Expenses):																									
16	Advertising	(\$708)	(\$367)		(\$566)		(\$710)		(\$980)		(\$664)		(\$907)		(\$256)		(\$447)		(\$377)		(\$972)		(\$1,162)		(\$8,116)	
17	Bank Service Charges	(\$2,100)	(\$577)		(\$1,429)		(\$1,593)		(\$989)		(\$999)		(\$1,697)		(\$2,041)		(\$2,872)		(\$609)		(\$1,289)		(\$1,462)		(\$17,657)	
18	Insurance	(\$1,890)	(\$2,077)		(\$1,251)		(\$1,543)		(\$1,265)		(\$803)		(\$2,360)		(\$1,426)		(\$2,160)		(\$1,326)		(\$1,237)		(\$2,553)		(\$19,891)	
19	Interest	(\$1,464)	(\$2,400)		(\$952)		(\$1,323)		(\$1,689)		(\$1,955)		(\$578)		(\$774)		(\$1,417)		(\$2,861)		(\$2,108)		(\$2,145)		(\$19,666)	
20	Inventory Purchases	(\$286)	(\$2,781)		(\$1,419)		(\$3,923)		(\$1,075)		(\$463)		(\$3,336)		(\$451)		(\$2,107)		(\$2,731)		(\$2,682)		(\$23,361)			

Figure 31.1: Screenshot of “Cash Flow Statement.xlsx” opened with Excel.

You'll use `pandas` to read, extract, and reshape these data and `matplotlib` to turn them into the waterfall plot below:

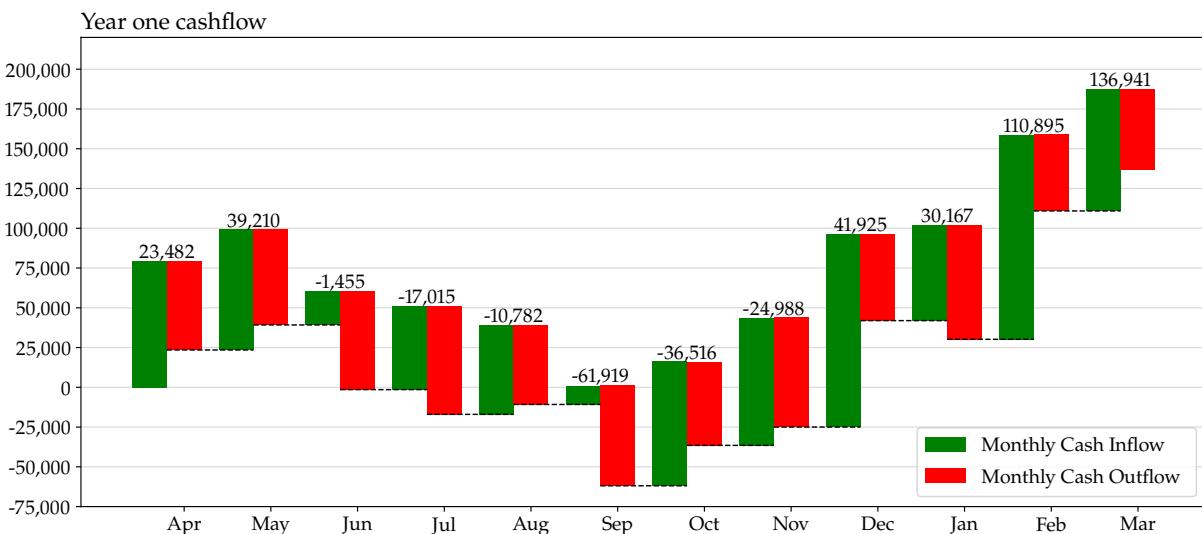


Figure 31.2: Waterfall plot made with `matplotlib` using data from “Cash Flow Statement.xlsx”.

1: Waterfall plots — also known as cascade charts — are mainstream visualization tools in accounting.

This project has three parts: loading and reshaping the data, creating the main plot elements, and styling the plot by adding tick labels, legend entries, a title, and a few other visual elements.

To get started, launch JupyterLab and open this chapter's notebook. You'll need both `pandas` and `matplotlib` for this project, so go ahead and `import` them at the top of your notebook with:

```
In [1]: import pandas as pd
import matplotlib.pyplot as plt
```

Now let's load the cash flow data into a `DataFrame` and reshape it into something `matplotlib` can work with.

Loading the cash flow data

Reading data from Excel files with `pandas` is no mystery by now. You can load data from “*Cash Flow Statement.xlsx*” into a `pandas DataFrame` with:

```
In [2]: df = pd.read_excel('Cash Flow Statement.xlsx')

df
```

```
Out[2]:      CASH FLOW Statement          Unnamed: 1  ...  Unnamed: 25  Unnamed: 26
0             Year 1                  NaN  ...        NaN        NaN
1                 NaN                  NaN  ...        NaN        NaN
2                 NaN                  NaN  ...        NaN        NaN
3  Beginning Cash Bal...                  NaN  ...        NaN        NaN
4  Cash Inflows (Inco...                  NaN  ...        NaN        NaN
..                   ...
41                 NaN                  NaN  ...        NaN        0
42                 NaN                  Subtotal  ...        NaN     -91459
43                 NaN      Total Cash Outf...  ...        NaN     -668410
44  Ending Cash Balance                  NaN  ...        NaN        NaN
45                 NaN                  NaN  ...        NaN        NaN
```

[46 rows x 27 columns]

However, it looks like the `DataFrame` above needs some scrubbing before you can use it with `matplotlib`.

For the waterfall plot, you'll need the beginning and ending cash balances for each month, as well as the monthly in and out cash flows. You can easily spot these values in Excel, but figuring out where they are in the `DataFrame` above needs some trial-and-error. For instance, month labels are in the third row of `df` (among many `NaN` values):

```
In [3]: df.loc[2]
```

```
Out[3]: CASH FLOW Statement      NaN
         Unnamed: 1      NaN
         Unnamed: 2      Apr
         Unnamed: 3      NaN
         Unnamed: 4      May
         ...
         Unnamed: 22     Feb
         Unnamed: 23     NaN
         Unnamed: 24     Mar
         Unnamed: 25     NaN
         Unnamed: 26     TOTALS
Name: 2, Length: 27, dtype: object
```

Similarly, beginning cash balances are in the fourth row of `df`:

```
In [4]: df.loc[3]
```

```
Out[4]: CASH FLOW Statement      Beginning Cash Bal...
         Unnamed: 1      NaN
         Unnamed: 2      0
         Unnamed: 3      NaN
         Unnamed: 4      23482
         ...
         Unnamed: 22     30167
         Unnamed: 23     NaN
         Unnamed: 24     110895
         Unnamed: 25     NaN
         Unnamed: 26     NaN
Name: 3, Length: 27, dtype: object
```

Once you figure out which rows you need to keep, you can discard all the other ones by reassigning your selection to `df`:

```
In [5]: df = df.loc[[2, 3, 11, 43, 44]]
```

```
df
```

```
Out[5]:   CASH FLOW Statement      Unnamed: 1 ... Unnamed: 25  Unnamed: 26
         2             NaN      NaN ...      NaN      TOTALS
         3  Beginning Cash Bal...      NaN      ...      NaN      NaN
         11            NaN      Total Cash Inflows ...      NaN      841075
         43            NaN      Total Cash Outf... ...      NaN      -668410
         44  Ending Cash Balance      NaN      ...      NaN      NaN
[5 rows x 27 columns]
```

Even with fewer rows, it's still hard to tell what these rows are: there are lots of missing values, missing headers or headers mixed with values. You need to apply several reshaping and cleaning steps to make these data useful:

```
In [6]: df = df.transpose().dropna().reset_index(drop=True)
```

```
df
```

```
Out[6]:    2      3      11     43     44
0  Apr       0   79252 -55770  23482
1  May   23482  75510 -59782  39210
2  Jun   39210  21143 -61808 -1455
3  Jul    -1455  52418 -67978 -17015
4  Aug   -17015  56142 -49909 -10782
5  Sep   -10782  11863 -63000 -61919
6  Oct   -61919  77979 -52576 -36516
7  Nov   -36516  80284 -68756 -24988
8  Dec   -24988 121274 -54361  41925
9  Jan    41925  60215 -71973  30167
10 Feb   30167 128583 -47855 110895
11 Mar  110895  76412 -50366 136941
```

The code above rotates the table from wide format to tall format (i.e., a tall table has more rows than columns); it then discards all rows that contain (any) missing values and resets the df's row labels to be consecutive numbers. The table is easier to understand now, but adding more descriptive column names would be even better; you can do that by running:

```
In [7]: df.columns = [
    'Month', 'Beginning Cash Balance', 'Total Cash Inflow',
    'Total Cash Outflow', 'Ending Cash Balance'
]
```

```
df
```

```
Out[7]:   Month Beginning Cash Balance Total Cash Inflow Total Cash Outflow Ending Cash Balance
0   Apr                  0           79252        -55770        23482
1   May                23482          75510        -59782        39210
2   Jun                39210          21143        -61808       -1455
3   Jul                 -1455          52418        -67978       -17015
4   Aug                -17015          56142        -49909       -10782
5   Sep                -10782          11863        -63000       -61919
6   Oct                -61919          77979        -52576       -36516
7   Nov                -36516          80284        -68756       -24988
8   Dec                -24988         121274        -54361        41925
9   Jan                 41925          60215        -71973       30167
10  Feb                30167         128583        -47855      110895
11  Mar              110895          76412        -50366      136941
```

Tall tables² are always easier to manipulate and visualize in Python's data universe — if your Excel data has more columns than rows, consider reshaping it as you did here.

There's one last data preparation step you need to do. All cash outflows are negative values, but it will be easier to plot these values later if they're positive. You can multiply the 'Total Cash Outflow' column by -1 to turn them into positive values:

```
In [8]: df['Total Cash Outflow'] = df['Total Cash Outflow'] * (-1)
```

2: In statistical data analysis, *tidying data* is a well-defined process for making data ready for analysis. One of the steps in this process is reformatting wide tables into tall ones. Tidy datasets are often easier to manipulate and visualize, regardless of the tools you use to analyze them. You can read more about data tidying at vita.had.co.nz/papers/tidy-data.pdf.

To check that nothing went wrong in the cleanup, you can verify that the ending cash balance for each month is equal to the starting balance plus inflow minus outflow:

```
In [9]: df['Ending Cash Balance'] == (
    df['Beginning Cash Balance']
    + df['Total Cash Inflow']
    - df['Total Cash Outflow']
)
```

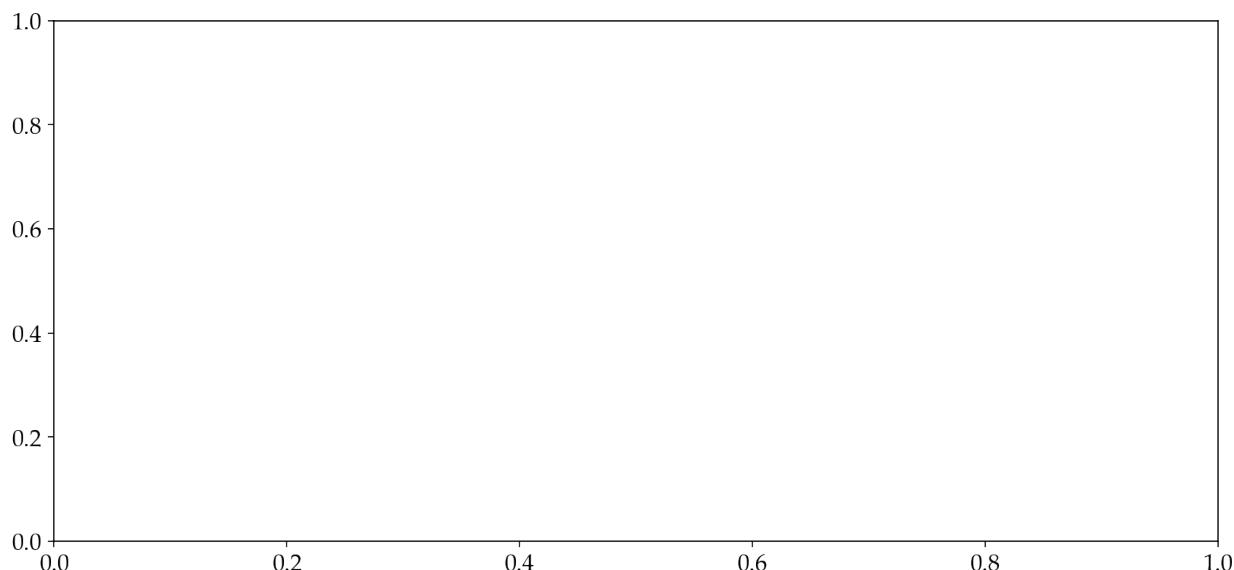
```
Out [9]: 0    True
1    True
2    True
3    True
4    True
5    True
6    True
7    True
8    True
9    True
10   True
11   True
dtype: bool
```

It looks like you have a cash flow dataset that is clean and ready for plotting with `matplotlib`.

Adding the main plot elements

You saw in the previous chapter that the first step in making a `matplotlib` plot is to set up the `Figure` and `Axes` objects:

```
In [10]: fig = plt.figure(figsize=(12, 6))
ax = fig.add_subplot()
```

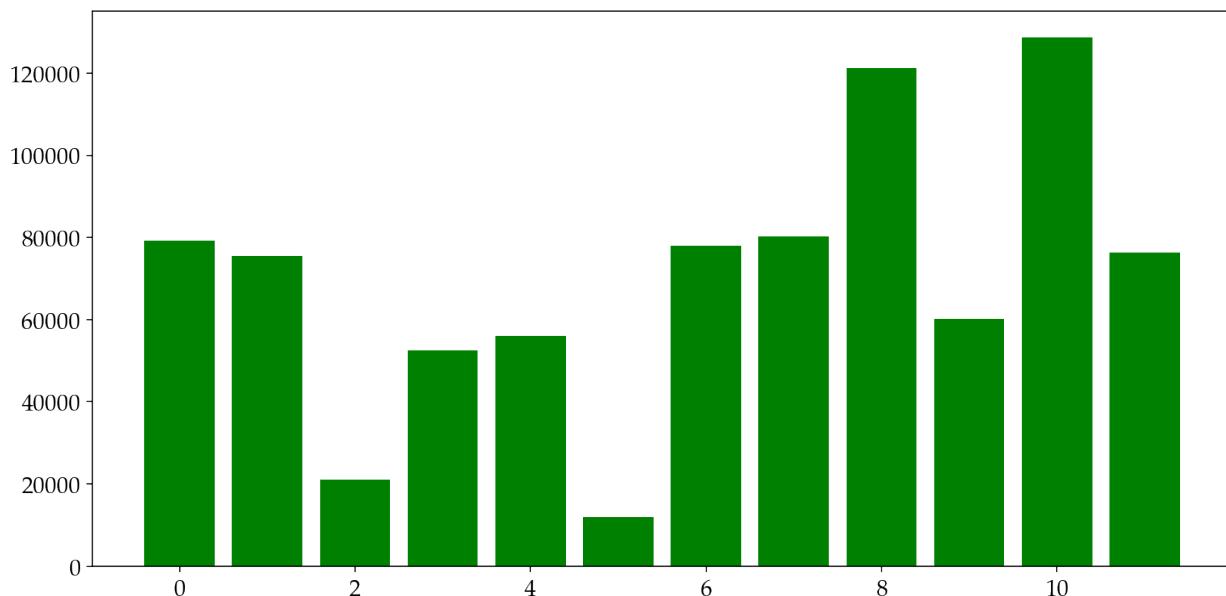


I made the figure extra wide (i.e., by passing the `figsize` keyword argument to `figure`) so all the waterfall bars fit, and the plot doesn't look too crowded. The first value in the tuple passed to `figsize` is the plot width, followed by its height.

A waterfall plot is a type of bar plot. To draw bars on an `Axes` variable, you need to call its `bar` method, passing a list of x and y-coordinates for the bars you want to draw as its first two arguments. In this case, you can use `df`'s row labels as the x-coordinates (because they're a list of consecutive numbers), and `df`'s '`Total Cash Inflow`' as y-coordinates:

```
In [11]: # you get the same Figure and Axes variables as before
# with the code below, but in one line of code
fig, ax = plt.subplots(1, 1, figsize=(12, 6))

ax.bar(df.index,
       df['Total Cash Inflow'],
       color='green')
```

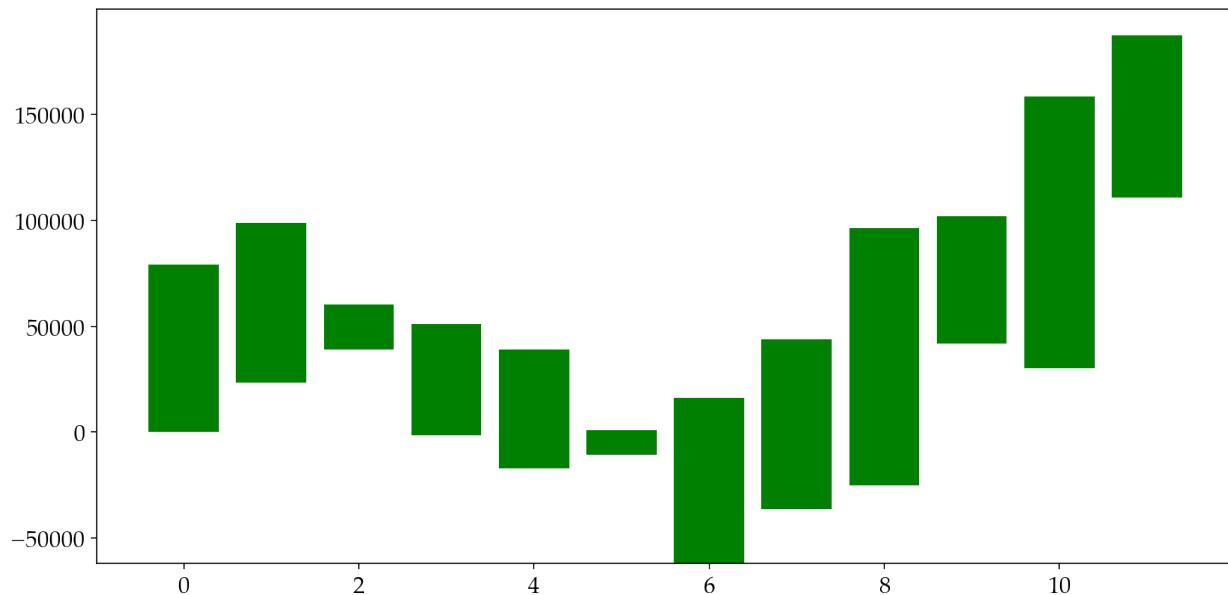


You now have some green bars representing monthly cash inflows. Because `df` has 12 rows of values, the code above draws 12 green bars. However, all the bars in the plot above start at 0. To turn this into a waterfall plot, you need to place the bottom of each green bar at the beginning cash balance for its month. You can do that passing a sequence of "bottom" y-coordinates (in this case, the beginning cash balance column) with the `bottom` keyword argument:

```
In [12]: fig, ax = plt.subplots(1, 1, figsize=(12, 6))

ax.bar(df.index,
       df['Total Cash Inflow'],
       bottom=df['Beginning Cash Balance'],
```

```
color='green')
```

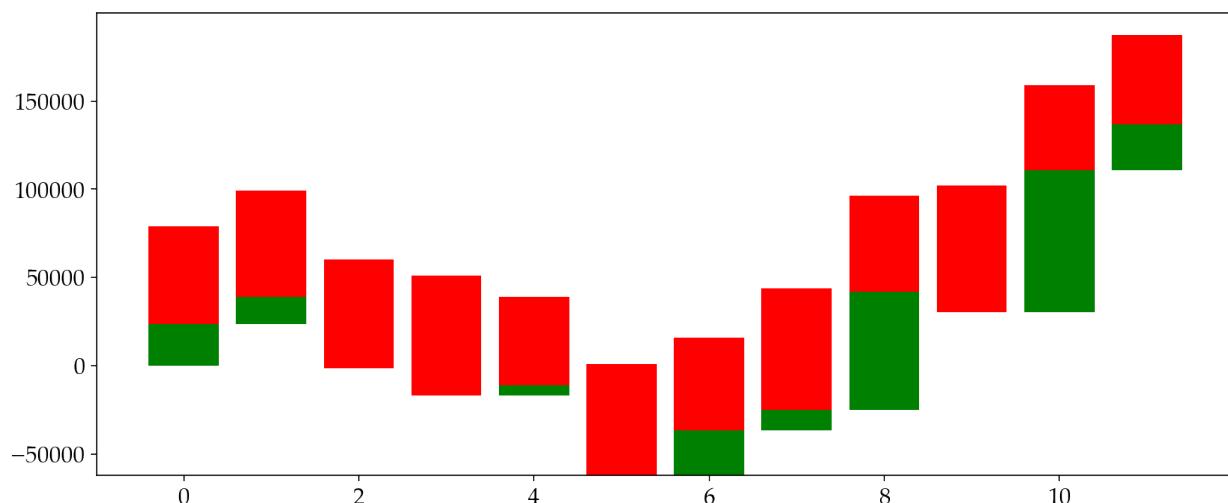


The plot is starting to look like the waterfall we wanted. The next step is adding the red bars — you can use the same `bar` method as with the green bars (but the bottom of red bars needs to be at the monthly ending cash balance, not the starting cash balance):

```
In [13]: fig, ax = plt.subplots(1, 1, figsize=(12, 6))

ax.bar(df.index,
       df['Total Cash Inflow'],
       bottom=df['Beginning Cash Balance'],
       color='green')

ax.bar(df.index,
       df['Total Cash Outflow'],
       bottom=df['Ending Cash Balance'],
       color='red')
```



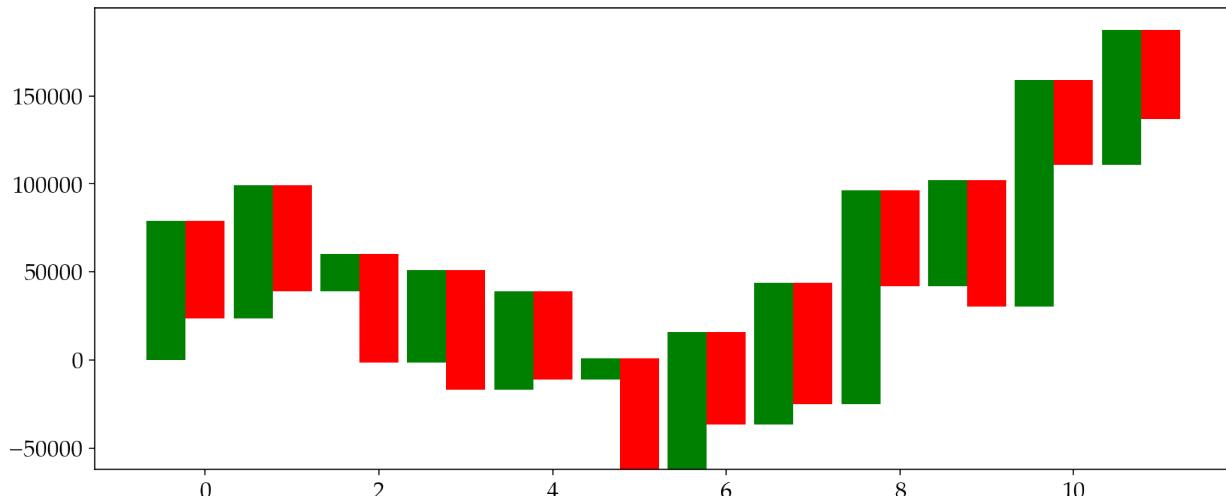
The code above adds red bars, but they overlap with the green bars because both sets of bars use the same list of x-coordinates (i.e., `df`'s row labels). You need to pass a different set of x-coordinate values to either the green bars or the red ones if you want them not to overlap. Luckily, you can modify the x-coordinates of the green bars by subtracting a constant value from the `df`'s index:

```
In [14]: fig, ax = plt.subplots(1, 1, figsize=(12, 6))

bar_width = 0.4

ax.bar(df.index - bar_width,
       df['Total Cash Inflow'],
       bottom=df['Beginning Cash Balance'],
       color='green',
       width=bar_width,
       label='Monthly Cash Inflow')

ax.bar(df.index,
       df['Total Cash Outflow'],
       bottom=df['Ending Cash Balance'],
       color='red',
       width=bar_width,
       label='Monthly Cash Outflow')
```



The code above sets an explicit `bar_width` (by passing the `width` keyword argument to `bar`) and shifts all green bars to the left by one `bar_width`. Once you understand what `matplotlib`'s functions and methods do, using them is simple geometry. You can place different elements on your plots by specifying their x and y-coordinates relative to the plot origin (which is the bottom left corner of the plot with coordinates (0, 0)).

I also added a `label` keyword argument to each `bar` function call above. When you add a legend in the following section, `matplotlib` will use these labels as the plot legend entries.

The figure above looks like the waterfall plot we wanted, but it's half-way there. Let's make it easier to understand by adding a title, a legend, and month names as x-axis tick labels next.

Styling the plot

Crafting the details on your plots is where all the `matplotlib` fun is. The waterfall bars we have now are in the right place, but the plot is crowded (especially on its vertical axis) — let's enlarge the y-axis limits to add some space above and below the bars. You can do that by adding the following code to the previous code cell:

```
In [15]: ax.set_ylim([-75000, 220000])
```

Another confusing part of the plot is its current x-axis tick labels. Instead of numbers, you can place the values in `df`'s 'Month' column as the x-axis tick labels with:

```
In [16]: xticks = df.index
xticklabels = df['Month']

ax.set_xticks(xticks)
ax.set_xticklabels(xticklabels);
```

Similarly, you can increase the frequency and improve the formatting of y-axis tick labels with:

```
In [17]: yticks = range(-75000, 225000, 25000)
yticklabels = [f'{ytick:,}' for ytick in yticks]

ax.set_yticks(yticks)
ax.set_yticklabels(yticklabels)
```

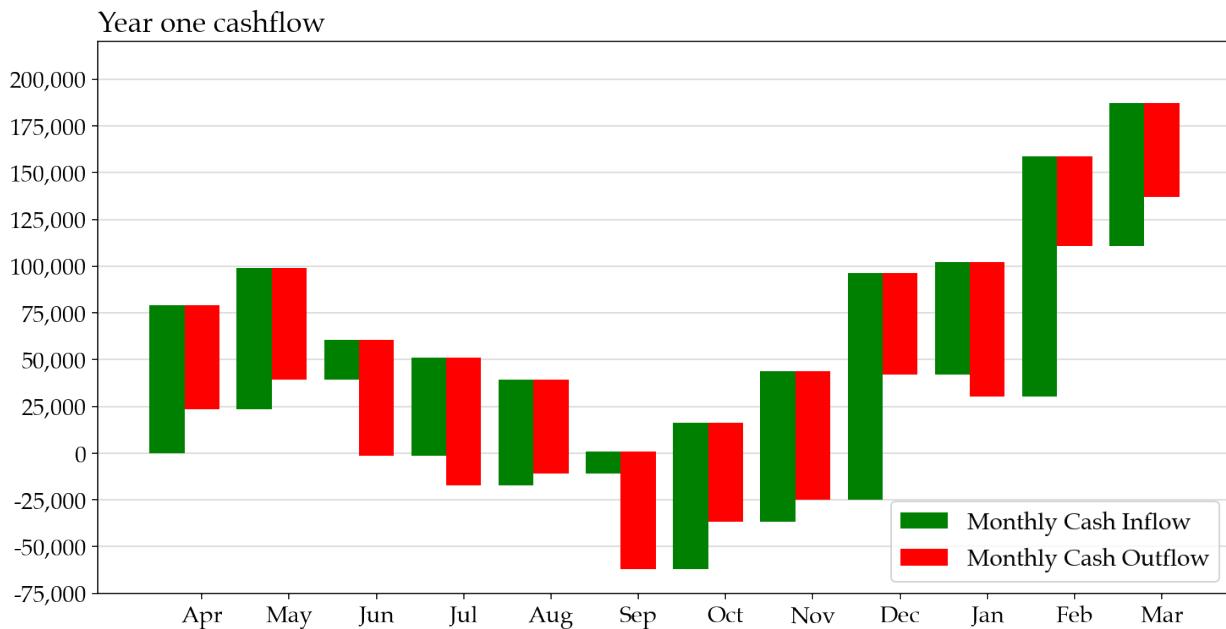
And you can also extend the y-axis ticks into the main plot area by adding a grid:

```
In [18]: ax.grid(axis='y', alpha=0.5)
ax.set_axisbelow(True)
```

The `set_axisbelow` method makes sure the y-axis grid lines are displayed beneath the bars and not on top. Finally, you can add a title and a legend to the plot using:

```
In [19]: ax.set_title('Year one cashflow', loc='left');
ax.legend(loc='lower right');
```

The waterfall plot is almost there, but it's still missing text annotations above the bars listing the ending cash balance for each month. For these text annotations, you'll have to use a `for` loop — the code is longer but no less explicit. If you add the code below to the same plotting cell as before, you'll get the extra annotations:



```
In [20]: for index, row in df.iterrows():
    beginning_balance = row['Beginning Cash Balance']
    ending_balance = row['Ending Cash Balance']
    inflow = row['Total Cash Inflow']

    annotation_x = index - bar_width / 2
    annotation_y = beginning_balance + inflow + 2000

    ax.annotate(f"{ending_balance:,.0f}",
                xy=(annotation_x, annotation_y),
                horizontalalignment='center')

    if index < 11:
        ax.hlines(ending_balance,
                  index - bar_width / 2,
                  index + 2 * bar_width,
                  color='black', linewidth=1, linestyle='dashed')
```

The `for` loop goes through each row in `df` and, for each row, calls the `ax.annotate` method. Different `xy` coordinates for the text label are computed and set as the `xy` keyword argument at each iteration of the loop.

The entire code needed to create the waterfall plot shown at the beginning of this chapter is listed below:

```
In [21]: fig, ax = plt.subplots(1, 1, figsize=(12, 5.5))

# Adding the main plot elements
bar_width = 0.4
ax.bar(df.index - bar_width,
       df['Total Cash Inflow'],
       bottom=df['Beginning Cash Balance'],
```

```

color='green',
width=bar_width,
label='Monthly Cash Inflow') 8
8
9
10
11
12
12
13
13
14
14
15
15
16
16
17
17
18
18
19
19
20
20
21
21
22
22
23
23
24
24
25
25
26
26
27
27
28
28
29
29
30
30
31
31
32
32
33
33
34
34
35
35
36
36
37
37
38
38
39
39
40
40
41
41
42
42
43
43
44
44
45
45
46
46
47
47
48
48
49
49
50
50
51
51
52
52
53
53
54
54
55
55

```

color='red',
width=bar_width,
label='Monthly Cash Outflow')

Styling the plot

ax.set_xlim([-75000, 220000])

xticks = df.index
xticklabels = df['Month'].unique()

ax.set_xticks(xticks)
ax.set_xticklabels(xticklabels)

yticks = range(-75000, 225000, 25000)
yticklabels = [f'{ytick:,}' for ytick in yticks]

ax.set_yticks(yticks)
ax.set_yticklabels(yticklabels)

ax.grid(axis='y', alpha=0.5)
ax.set_axisbelow(True)

ax.set_title('Year one cashflow', loc='left')
ax.legend(loc='lower right')

Adding text annotation and lines

for index, row in df.iterrows():

 beginning_balance = row['Beginning Cash Balance']
 ending_balance = row['Ending Cash Balance']
 inflow = row['Total Cash Inflow']

 ax.annotate(f'{ending_balance:,}',
 xy=(index - bar_width / 2, beginning_balance + inflow + 2000),
 horizontalalignment='center')

 if index <= 10:
 ax.hlines(ending_balance,
 index - bar_width / 2,
 index + 2 * bar_width,
 color='black', linewidth=1, linestyle='dashed')

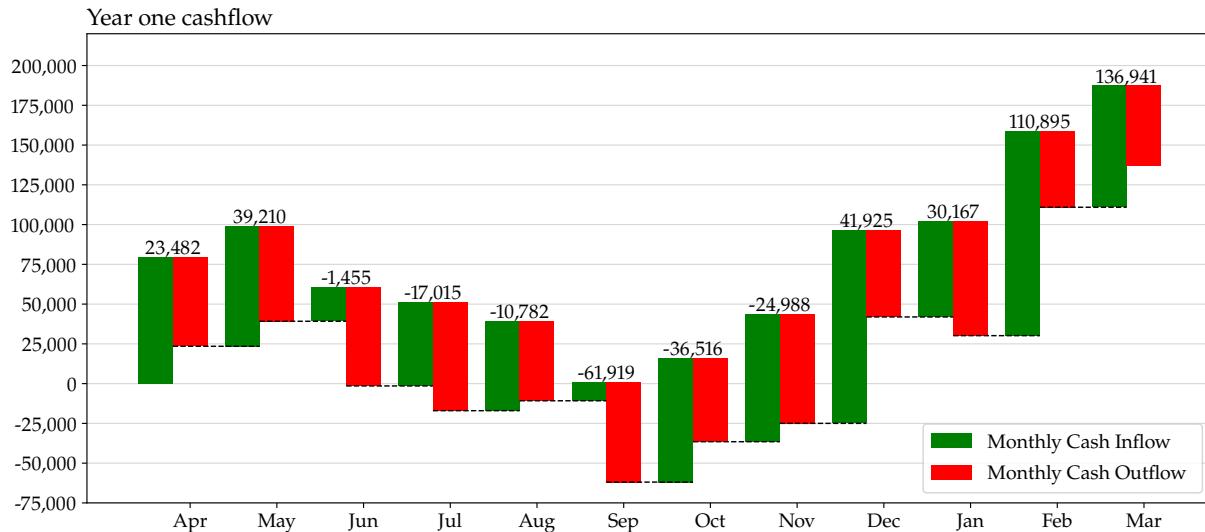


Figure 31.3: Waterfall plot made with `matplotlib` using “*Cash Flow Statement.xlsx*” data.

That's a lot of code for a relatively simple plot. If you need a quick waterfall chart, you're likely better off with Excel. However, if you need to make a waterfall plot every other month using a different cash flow statement, you can re-use the code above with minimum effort. The real value of using Python and `matplotlib` is that you can easily scale and customize your plot making.

Summary

This chapter showed you how to turn a cash flow statement into a waterfall plot using Python, `pandas`, and `matplotlib`. As you saw above, working with `matplotlib` can sometimes require a lot of coding. However, there are faster ways to create plots with Python, and the following chapter introduces some of them.

Other plotting libraries

In the previous chapter, you created a `matplotlib` plot from scratch, setting up `Figure` and `Axes` objects, adding plot elements one-by-one, adjusting labels, ticks, legend entries, and everything in-between. The flexibility you get with `matplotlib` is a benefit if you need to customize your plots down to the smallest details, but it can also get annoying if you just need to visualize your data quickly. This chapter shows you how to use the `pandas`, `seaborn`, and `hvplot` libraries to turn data into plots with far less code than you need with `matplotlib`.

You'll be using the same Q1 daily sales data you used in the `matplotlib` chapter — you can load it in your chapter notebook by running the following code:

```
In [1]: import pandas as pd  
  
daily_sales_df = pd.read_csv('Q1DailySales.csv', parse_dates=['Date'])  
daily_sales_df = daily_sales_df.set_index('Date')  
  
daily_sales_df
```

```
Out[1]:
```

	Understock.com	Shoppe.com	iBay.com	Walcart	Bullseye
Date					
2020-01-01	20707.62	6911.72	5637.54	13593.17	9179.39
2020-01-02	18280.59	17351.46	5959.61	12040.16	5652.32
2020-01-03	17191.15	10578.60	8346.60	9876.21	6127.92
2020-01-04	17034.69	6052.03	10168.41	12811.26	10370.95
2020-01-05	17074.18	11866.74	12462.30	8318.34	4641.02
...
2020-03-27	20183.41	5111.76	7703.85	3026.12	503.20
2020-03-28	8190.09	1392.89	4456.91	2776.78	1772.25
2020-03-29	7267.96	3966.57	6717.35	1195.16	1142.15
2020-03-30	14712.07	3826.95	11044.34	1702.39	676.08
2020-03-31	12343.37	10372.53	13687.49	157.49	848.51

[91 rows x 5 columns]

Plotting with pandas

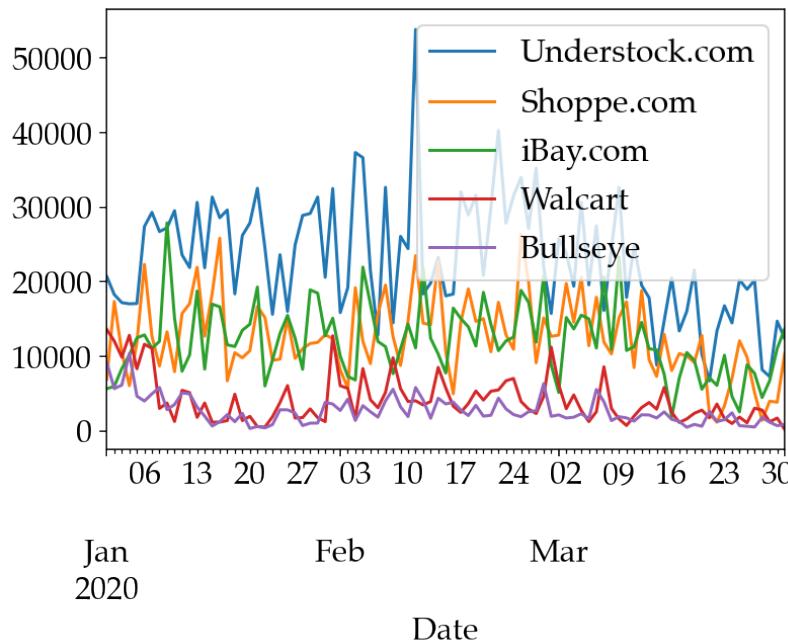
In addition to its data slicing tools, `pandas` lets you quickly turn `Series` or `DataFrame` objects into plots by calling their `plot` method. When you call the `plot` method, `pandas` actually creates a `matplotlib` plot for you: it deals with a lot of the boilerplate (e.g.,

creates `Figure` and `Axes` objects) and makes sensible styling choices for you so that you can visualize data with less code.

To turn the daily sales data into a plot, instead of setting up `Figure` and `Axes` objects, all you need to run is:

```
In [2]: daily_sales_df.plot()
```

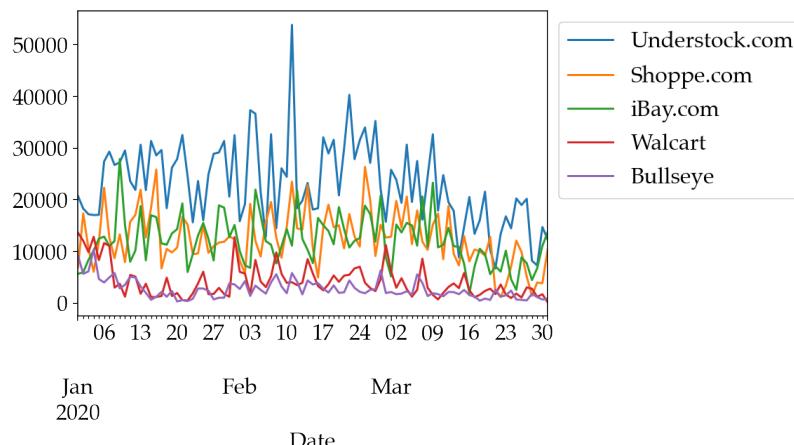
```
Out [2]: <AxesSubplot:xlabel='Date'>
```



This is still a `matplotlib` plot, but it's created by `pandas`, which uses `matplotlib` internally and makes a few styling choices on your behalf: adds a legend to the plot, increases the base font size, and nicely formats the x-axis tick labels. The index of `daily_sales_df` (i.e., its row labels) is used as x-coordinates, and values from each of its columns are used as y-coordinates for the separate lines.

The `pandas` `plot` method returns a `matplotlib` `Axes` object (the same `matplotlib` `Axes` object you used throughout the previous two chapters). You can assign this object to a variable and customize your plot further, using the same `matplotlib` styling methods we explored earlier. For example, if you want to move the legend outside of the main plotting area, you can use:

```
In [3]: ax = daily_sales_df.plot() 1
2
ax.legend(bbox_to_anchor=(1, 1), loc='upper left'); 3
```



In addition to using `matplotlib` styling methods, if you choose a different `matplotlib` style or set different `plt.rcParams` settings, any plots created with `pandas` will reflect those settings as well.

Why use `matplotlib` at all when `pandas` makes it easier to create the same plots in less code? There is no good reason: plotting with `pandas` often leads to code that is faster to write and easier to understand, so you should use it regularly. Even more, `pandas` makes certain styling choices for you that are more sensible than the default `matplotlib` ones. You will often find that `pandas`'s styling choices are good enough, and you don't need to customize your plots further. However, you will at times need to customize parts of your `pandas` plots. In those cases, knowing what `matplotlib` methods to use and how is the only way you'll be able to go beyond what is provided out-of-the-box by `pandas`.

The `pandas` `plot` method accepts several keyword arguments you can use to customize your plots (with similar names to their `matplotlib` equivalents). Depending on whether you are calling `plot` on a `Series` (e.g., a single column) or a `DataFrame`, different keyword arguments are available — table 32.1 lists some of them.

Table 32.1: Optional keyword arguments available when calling `plot` on a `Series` or `DataFrame` object. Read more about plotting with `pandas` in the official documentation available at pandas.pydata.org/pandas-docs/stable/user_guide/visualization.html.

Argument	Example	Description
<code>kind</code>	<code>daily_sales_df.plot(kind='bar')</code>	Sets the kind of plot to create. Valid options are <code>'line'</code> , <code>'bar'</code> , <code>'barh'</code> , <code>'hist'</code> , <code>'box'</code> , <code>'area'</code> , <code>'density'</code> , <code>'pie'</code> , <code>'scatter'</code> and <code>'hexbin'</code> . By default, a <code>'line'</code> plot is created. We will cover different kinds of plots in the following section.
<code>figsize</code>	<code>daily_sales_df.plot(figsize=(12, 6))</code>	Sets the figure width and height, in inches (same as <code>matplotlib</code> 's <code>plt.figure</code>).

Table 32.1: Optional keyword arguments available when calling `plot` on a `Series` or `DataFrame` object. Read more about plotting with pandas in the official documentation available at pandas.pydata.org/pandas-docs/stable/user_guide/visualization.html.

Argument	Example	Description
y	<code>daily_sales_df.plot(figsize=(12, 6), y=['iBay.com', 'Shoppe.com'])</code>	Selects which columns from the <code>daily_sales_df</code> DataFrame to plot (i.e., to use as y-coordinates for the separate plot items). By default, all columns are drawn on the plot, using the DataFrame index for the x-coordinates. Requires a list of column names.
x	<code>daily_sales_df.plot(figsize=(12, 6), x='iBay.com', y=['Shoppe.com', 'Understock.com'])</code>	Selects which column from <code>daily_sales_df</code> to use for x-coordinates. By default, the DataFrame index is used for the x-coordinates. Only one column name can be specified for the x argument. The x and y keyword arguments are available only for DataFrame objects.
title	<code>daily_sales_df.plot(title='Daily sales in February')</code>	Sets the plot's title, as in the <code>matplotlib</code> example from the previous section.
grid	<code>daily_sales_df.plot(grid=True)</code>	Enables the plot grid. By default, the plot grid is not visible.
legend	<code>daily_sales_df.plot(legend=False)</code>	Disables the plot legend. By default, the legend is visible. Another value you can assign to <code>legend</code> is ' <code>reverse</code> ', which reverses the legend item order.
xticks yticks	<code>daily_sales_df.plot(xticks=pd.date_range('31 January 2020', '01 March 2020', freq='W-MON'), yticks=[0, 20000, 40000, 60000, 80000, 100000])</code>	Sets the values to use for the x-axis or y-axis ticks. Both arguments require a sequence of values (e.g., a Python list).
xlim ylim	<code>daily_sales_df.plot(xlim=[pd.Timestamp('31 Jan 2020'), pd.Timestamp('15 Feb 2020')], ylim=[0, 40000])</code>	Sets the lower and upper plot limits on either axis. If there is data beyond these limits, it will not be shown on the plot. Both arguments require a sequence of two values (e.g., a Python list or tuple containing two values, representing the lower and upper bounds of the axes).
rot	<code>daily_sales_df.plot(rot=30)</code>	Sets tick label rotation in degrees (x-ticks for vertical, y-ticks for horizontal plots).

Different kinds of pandas plots

When you call the `plot` method on a `DataFrame` without any arguments, pandas will try to make a line plot from its data (i.e., draw one line for each column, using DataFrame row labels as x-coordinates and column values as y-coordinates).

Suppose your data isn't suited for a line plot. In that case, you can tell pandas to draw a different kind of plot by passing the `kind` keyword argument to `plot` — valid options for the `kind` keyword argument are: `'line'`, `'bar'`, `'barh'`, `'hist'`, `'box'`, `'area'`, `'density'`, `'pie'`, `'scatter'` and `'hexbin'`. Most of these plot types have equivalents in Excel, so they should be familiar, even though they might not share the same name. Table 32.2 below shows an example of each created using the daily sales data (for some of the examples, I summed `daily_sales_df` column-wise before creating the plot).

Table 32.2: Different plot types you can create using the `pandas` `plot` method. Read more about plotting with pandas in the official documentation available at pandas.pydata.org/pandas-docs/stable/user_guide/visualization.

Plot type	Example	Output
<code>'line'</code>	<code>daily_sales_df.plot(kind='line')</code>	
<code>'area'</code>	<code>daily_sales_df.plot(kind='area', legend='reverse')</code>	
<code>'pie'</code>	<code>daily_sales_df.sum().plot(kind='pie', explode=[0, 0, 0.1, 0, 0])</code>	
<code>'bar'</code>	<code>daily_sales_df.sum().plot(kind='bar', rot=20)</code>	

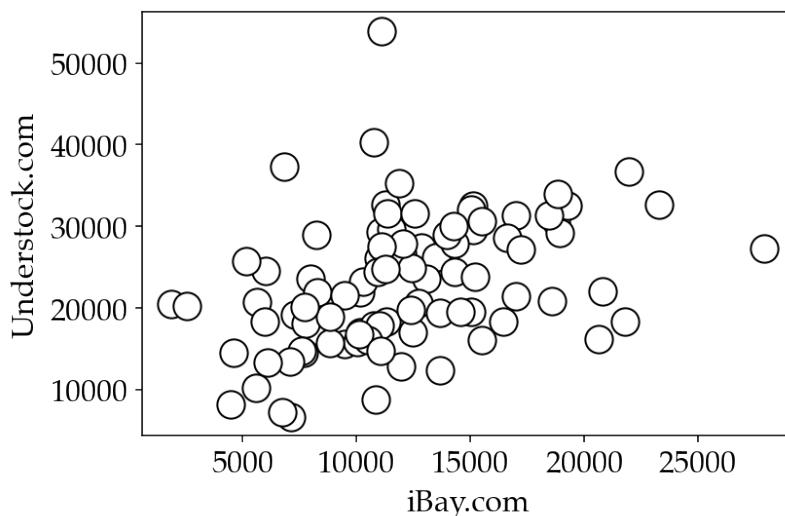
Table 32.2: Different plot types you can create using the `pandas` `plot` method. Read more about plotting with `pandas` in the official documentation available at pandas.pydata.org/pandas-docs/stable/user_guide/visualization.html.

Plot type	Example	Output
'barh'	<code>daily_sales_df.sum().plot(kind='barh')</code>	
'hist'	<code>daily_sales_df.plot(kind='hist', bins=20)</code>	
'density'	<code>daily_sales_df.plot(kind='density')</code>	
'box'	<code>daily_sales_df.plot(kind='box', rot=30)</code>	
'scatter'	<code>daily_sales_df.plot(kind='scatter', x='iBay.com', y='Understock.com')</code>	
'hexbin'	<code>daily_sales_df.plot(kind='hexbin', x='iBay.com', y='Understock.com', gridsize=10, sharex=False)</code>	

You may have noticed that, depending on the kind of plot you

need, different keyword arguments are available with the `pandas` `plot` method. For instance, the `explode` keyword argument is available for `'pie'` plots and allows you to move pie wedges away from the center of the plot, but isn't available for `'line'` plots. All these different keyword arguments are difficult to discover or remember. Even more, if you run `daily_sales_df.plot?` in a separate cell, you will only see the more generic plotting arguments listed and explained. To make them more discoverable, `pandas` offers an alternative to the `kind` keyword argument: you can use `plot` as an accessor (like you use `str` or `dt` to access string or date methods on a `DataFrame`), but in this case, to access more specific plotting methods. For example, you can create a scatter plot with the following code:

```
In [4]: (daily_sales_df
      .plot
      .scatter(
          x='iBay.com', y='Understock.com',
          marker='o', s=200,
          color='white', edgecolor='black'
      )
)
```



Notice that instead of calling `plot(kind='scatter', ...)` you can use `plot.scatter(...)`. While both options produce the same plot, the main benefit of using the latter version is that you can also run `daily_sales_df.plot.scatter?` in a separate code cell to read documentation specific to scatter plots, and not generic plotting documentation (that's how I found out you can use the `s` keyword argument to control the size of scatter points on a plot).

Similarly, you can use `plot` as an accessor for all the different kinds of plots mentioned above (e.g., `daily_sales_df.sum().plot.pie()` to draw a pie plot from the sales data). Whichever notation you end up using, you will likely come across both styles in other examples

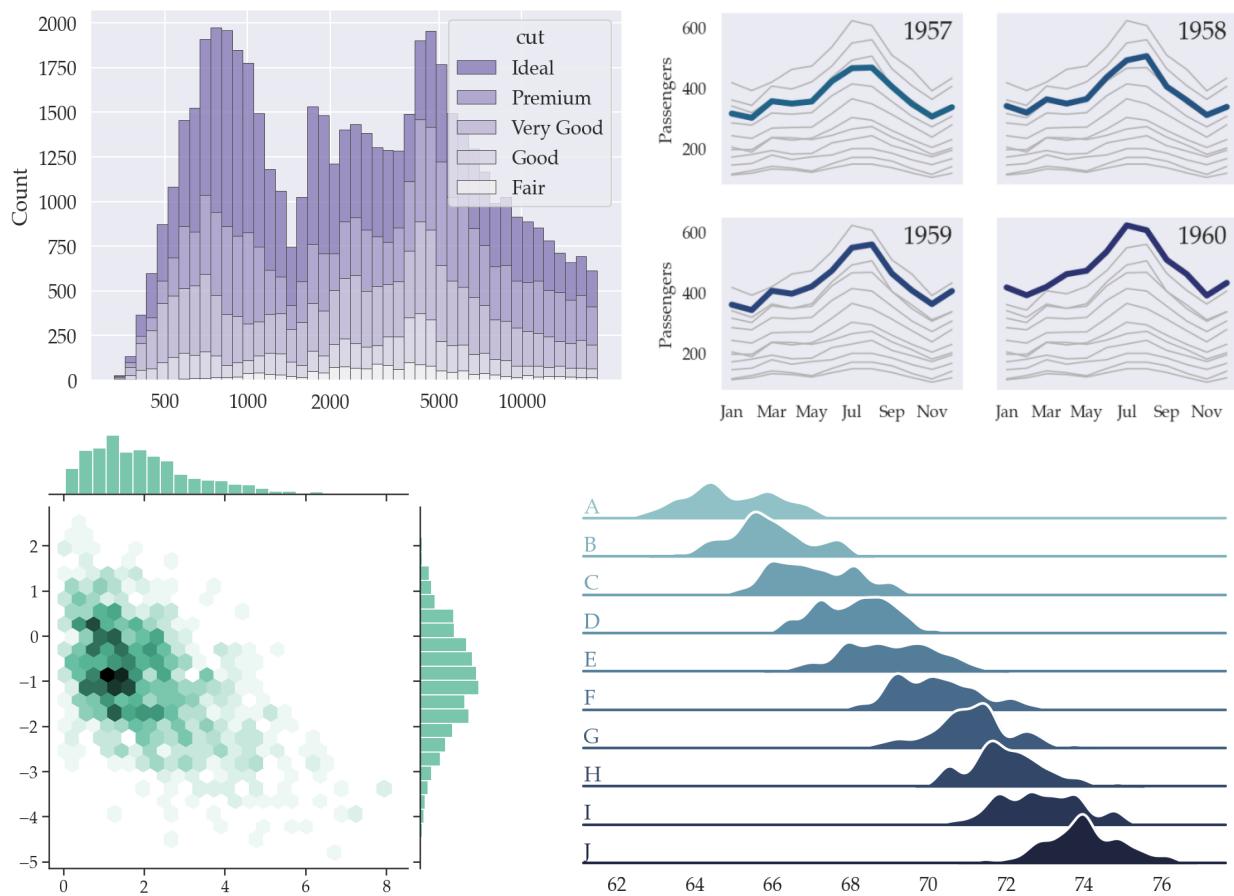


Figure 32.1: A few plots from `seaborn`'s example gallery (from top to bottom, these are called *stacked histogram*, *line plot*, *hexbin plot with marginal distributions* and *ridge plot*). You can view these plots (and the code for making them) and many more at seaborn.pydata.org/examples/index.html#example-gallery. The data used for these plots is either randomly generated, or sourced from publicly available datasets.

(in the `pandas` documentation or elsewhere online); reading about both styles here will prevent some confusion later on.

You can customize any plot you create with `pandas` using the same `matplotlib` methods we covered earlier. Together, `pandas` and `matplotlib` likely cover most simple data visualization needs — but if you want to make a complex plot with `matplotlib`, you'll need to roll up your sleeves. Or you can use `seaborn` instead.

Plotting with seaborn

`Seaborn` is another Python data visualization library. It is designed for drawing clear yet complex graphics, such as plot grids or plots that illustrate relationships in your data — you can see a few example plots created with `seaborn` in figure 32.1.

Just like `pandas`, `seaborn` is based on `matplotlib`. It creates a `matplotlib` plot for you, handling all the boilerplate of setting up `Figure` and `Axes` objects — and because it's based on `matplotlib`,

you can still use the same `Axes` methods to customize your `seaborn` plots further, if you need to.

Let's go through a quick example that illustrates how simple `seaborn` plotting code is. We'll use `daily_sales_df` again, but this time to look at how correlated daily sales are in the online sales channels in our data.

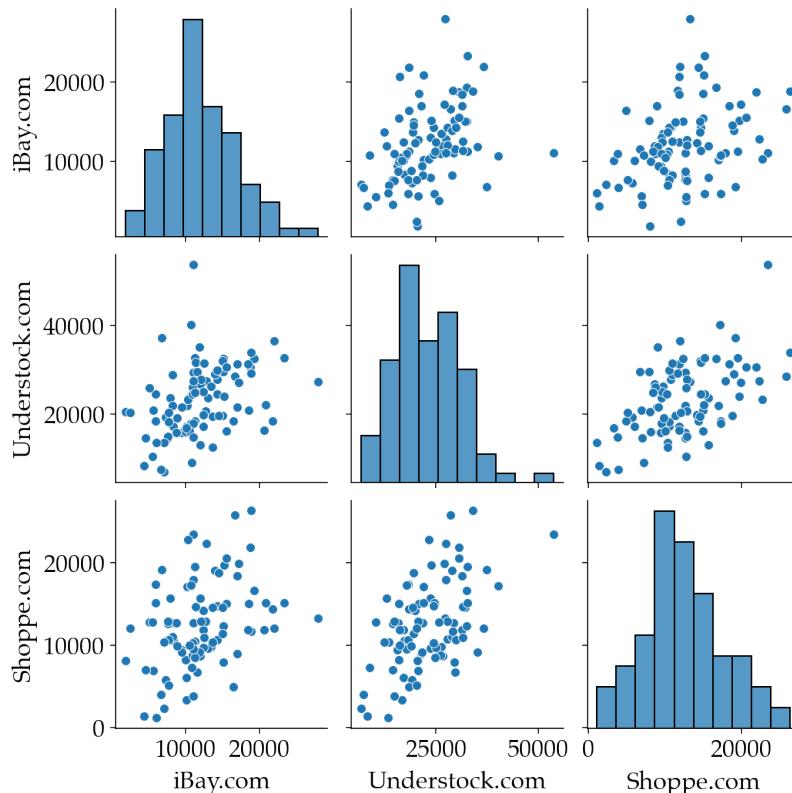
Like `matplotlib`, `seaborn` comes with Anaconda, so you already have it installed on your computer if you followed the setup guide in chapter 2. Before creating any plots, you first need to `import` `seaborn` into your notebook and assign it the `sns` alias:¹

```
In [5]: import seaborn as sns
```

To create a plot describing the relationship between total daily sales (in the three online sales channels in our data) you can use `seaborn`'s `pairplot` function. The following code creates a grid of plots showing the pairwise relationship between the columns of a `DataFrame` you pass as an argument to `pairplot`:

```
In [6]: sns.pairplot(daily_sales_df[['iBay.com', 'Understock.com', 'Shoppe.com']])
```

```
Out [6]: <seaborn.axisgrid.PairGrid at 0x1a22a2c450>
```



¹: You can use any alias you want to (or none at all), but the `sns` alias is commonly used.

The output above contains several `matplotlib` plots organized in a grid. The diagonal plots (i.e., the histograms) show how each column's values are distributed. In contrast, off-diagonal plots show the pairwise relationship between every combination of

columns (each point on the scatter plots is a different row in the `DataFrame` passed as input). The x and y-axis labels tell you which two channels are shown in each pairwise plot.

The `pairplot` function returns a `PairGrid` object, which is a `seaborn` specific plot “*container*” — the interesting part about this object is that it, well, contains the individual `matplotlib` plots that make up the grid you see above. To access these `matplotlib` plots, you can assign the output of `pairplot` to a separate variable and use its `axes` attribute:

```
In [7]: grid = sns.pairplot(daily_sales_df[['iBay.com', 'Understock.com', 'Shoppe.com']])           1
      grid.axes                                         2
      grid.axes                                         3
```

```
Out [7]: array([
    [<AxesSubplot:ylabel='iBay.com'>,
     <AxesSubplot:>,
     <AxesSubplot:>],
    [<AxesSubplot:ylabel='Understock.com'>,
     <AxesSubplot:>,
     <AxesSubplot:>],
    [<AxesSubplot:xlabel='iBay.com', ylabel='Shoppe.com'>,
     <AxesSubplot:xlabel='Understock.com'>,
     <AxesSubplot:xlabel='Shoppe.com'>]
], dtype=object)
```

Because we have a 3-by-3 grid of plots, `grid.axes` is a list² of lists, each list representing one row of plots on the grid and containing three `matplotlib` `Axes` objects. You can use any of the `matplotlib` methods we explored in the previous chapter with these `Axes` — for example, if you want to add titles to each of the plots on the top row of the grid, you can use the `Axes` `set_title` method:

```
In [8]: grid = sns.pairplot(daily_sales_df[['iBay.com', 'Understock.com', 'Shoppe.com']])           1
      first_row = grid.axes[0]                                         2
      first_row[0].set_title('First')                                     3
      first_row[1].set_title('Second')                                    4
      first_row[2].set_title('Third');                                     5
      first_row[2].set_title('Third');                                     6
```

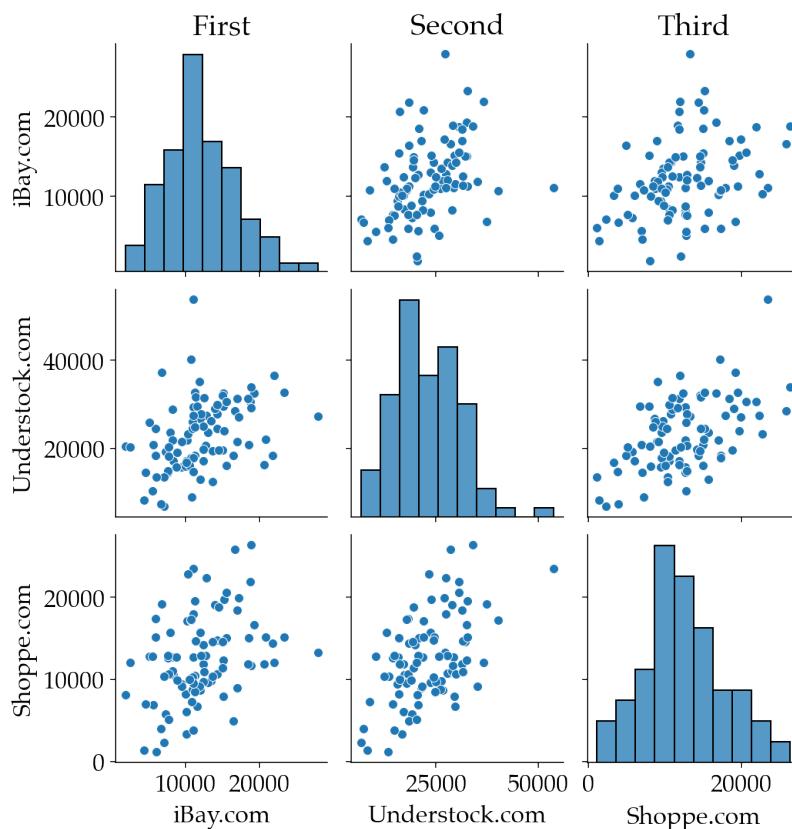
The output is shown on the following page.

Besides its `pairplot` function, `seaborn` comes with several other visualization functions — including a set of tools for handling plot styles and colors that is much easier to use than `matplotlib`'s styling tools. Going through each of its functions wouldn't be of much benefit because after you learn what `seaborn` does and how it works, you can always check its examples gallery³ to find a plot that looks like the one you're trying to make and adapt it to your use case — which is how I use it most often.

Overall, the `seaborn` library makes it easy to create intricate plots in fewer lines of code and fills in some of the missing plotting

2: It is actually an `array`, but you can use it like a Python list.

3: At seaborn.pydata.org/examples/index.html#example-gallery.



functionality in `pandas` and `matplotlib`. Even though it is designed for visualizations tailored to statistical data analysis, you might find its plots useful in your work, which is why I introduced it here. In the last project chapter, we'll use `seaborn` again to make a heatmap plot from the same sales dataset.

Using `matplotlib`, `pandas` and `seaborn` together

Before we leave `matplotlib` territory, one last feature I want to cover is creating subplots and putting `matplotlib`, `pandas` or `seaborn` plots in the same figure.

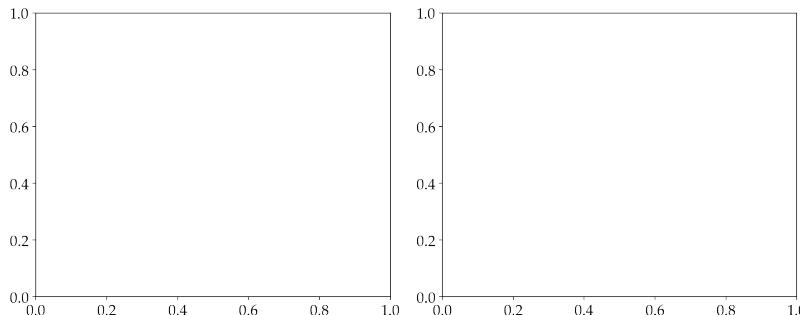
In the previous chapters, I mentioned that a `matplotlib` figure can have any number of `Axes` objects, not just one. You can create a `Figure` object and add multiple `Axes` to it with:

```
In [9]: import matplotlib.pyplot as plt

fig = plt.figure(figsize=(12, 6))
ax_left = fig.add_subplot(1, 2, 1) # rows, columns, position
ax_right = fig.add_subplot(1, 2, 2)
```

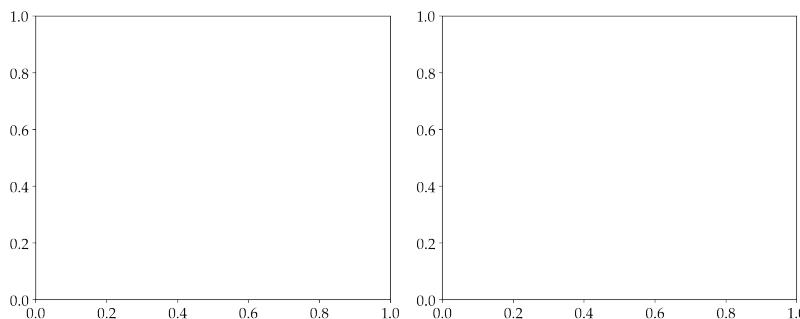
Here, you use `add_subplot` twice to add two `Axes` to the same figure — both of which are assigned to separate variables.⁴ The arguments passed to `add_subplot` above tell `matplotlib` to place

4: You can use any names for the `Axes` variables, `ax_left` and `ax_right` fit this example.



each of the `Axes` on a grid of subplots, at different positions. The `1` and `2` used with `add_subplot` above tell `matplotlib` how many rows and columns the plot grid will have. The last argument passed to `add_subplot` tells `matplotlib` the order of `Axes` in the grid. Below is a shorter alternative that does the same thing, but in one line instead of three:

```
In [10]: fig, (ax_left, ax_right) = plt.subplots(1, 2, figsize=(12, 6))
```



The `subplots` function returns a `Figure` object and a sequence of `Axes` objects (that is shaped according to the plot layout you want to create). In this example, you create one row and two columns of `Axes` (the first two arguments passed to `plt.subplots`) and unpack⁵ them into two variables called `ax_left` and `ax_right`. However, when you want to create more complicated layouts, it's easier to store the sequence of `Axes` as a separate variable and access individual `Axes` using indices. For instance, to create a 3-by-3 plot grid, you can use the following code:

```
In [11]: fig, grid = plt.subplots(3, 3, figsize=(12, 6))
```

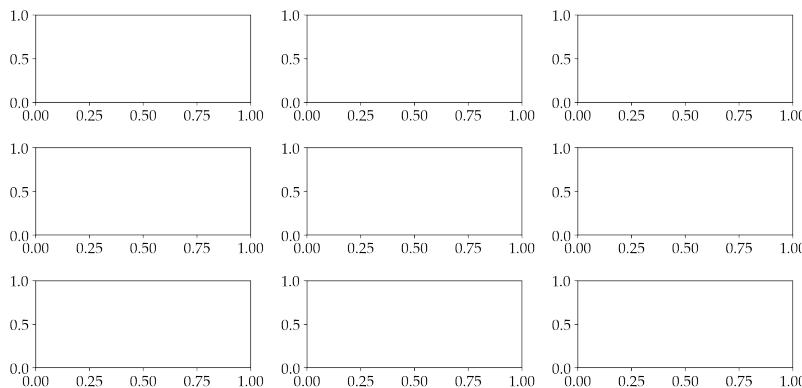
The `grid` variable above is similar to our earlier `seaborn` plot grid. If you inspect it in a separate cell, you'll see it is a list of lists⁶ each list representing one row of plots on the grid and containing three `matplotlib` `Axes` objects:

```
In [12]: grid
```

```
Out [12]: array([[[<AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>],
   [<AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>],
   [<AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>]], dtype=object)
```

5: Unpacking allows you to expand a Python list into separately named variables. Because `plt.subplots` returns a list with two `Axes` objects, you can unpack them as separate variables as I did here.

6: It is a `numpy array`, but you can use like a Python list.

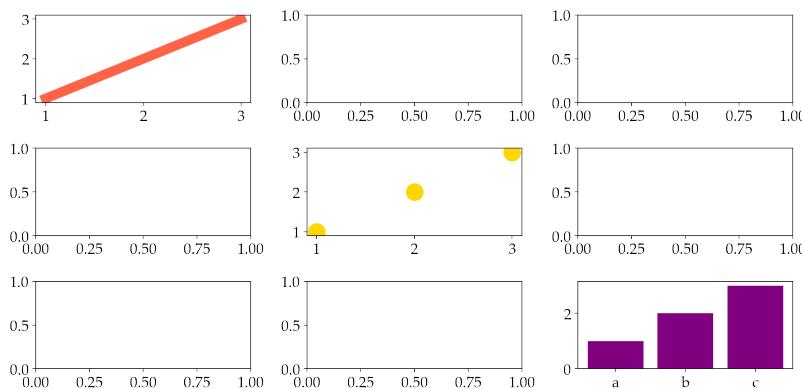


As with the `seaborn` example before, to access individual `Axes` objects in this grid, you can index the `grid` variable:

```
In [13]: fig, grid = plt.subplots(3, 3, figsize=(12, 6))

ax_top_left = grid[0][0]
ax_mid_center = grid[1][1]
ax_bottom_right = grid[2][2]

ax_top_left.plot([1, 2, 3], [1, 2, 3], linewidth=10, color='tomato')
ax_mid_center.scatter([1, 2, 3], [1, 2, 3], s=300, color='gold')
ax_bottom_right.bar(['a', 'b', 'c'], [1, 2, 3], color='purple')
```



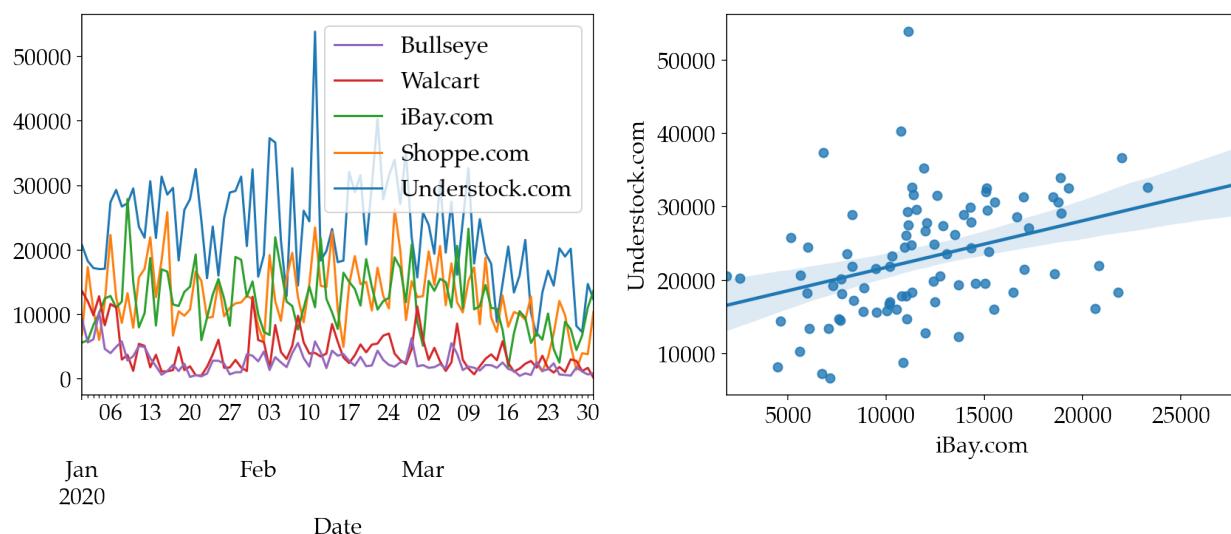
In the code example above, when indexing `grid`, the first index value is used to determine a row in the grid of plots, and the second value is used to determine a column. Together, they access an `Axes` object, on which you can use any of the (now familiar) `matplotlib` plotting methods — in this case, the `plot`, `scatter` and `bar` methods.

So far, this is just a rehashing of `matplotlib`'s features. The interesting part is that, because both `pandas` and `seaborn` extend `matplotlib`, you can setup a plot grid using `matplotlib` (as you did above) and then use `pandas` or `seaborn` to draw plots in the grid. For instance, you can set up a side-by-side plot layout using `matplotlib`, and then use `pandas` for the left plot, and `seaborn` for the right one:

```
In [14]: fig, (ax_left, ax_right) = plt.subplots(1, 2, figsize=(12, 6))

# pandas plot on the left
daily_sales_df.plot(
    ax=ax_left,
    legend='reverse'
)

# seaborn plot on the right
sns.regplot(
    ax=ax_right,
    data=daily_sales_df,
    x='iBay.com',
    y='Understock.com'
)
```



You've already seen the `pandas` `plot` method before — `seaborn`'s `regplot` function draws a regression line⁷ and its associated error from the values you pass as the `x` and `y` arguments.

Notice the `ax` keyword argument used with both plotting functions above: most `seaborn` plotting functions and the `DataFrame` `plot` method accept an `ax` keyword argument. If you already have `Axes` objects setup with `matplotlib` (like we do here), you can use the `ax` keyword argument to tell either `pandas` or `seaborn` which `Axes` to use for drawing.

One area where `matplotlib` is limited⁸ is creating interactive plots that you can zoom, pan or hover over to change the display of information. Let's quickly take a look at a Python visualization library that lets you do all of those things.

7: For more in-depth regression analysis, you can use Python's `scikit-learn` or `statsmodels` libraries, both of which you've already installed through Anaconda.

8: You can use `matplotlib` to create interactive plots, but I think the alternative mentioned below is easier to use and more powerful. You can read about interactive `matplotlib` plots at matplotlib.org/users/interactive.

Overthinking: Interactive plots

The world of Python visualization tools can be confusing at times. Besides the libraries we already covered, there are many other libraries that either make it easy to integrate plots into web-pages, simplify the code needed to create plots at the cost of customization options, and sometimes both.

Like with `matplotlib` and its extension libraries (i.e., `pandas` and `seaborn`), there are several other core libraries for creating interactive plots in Python, each with its own extension libraries that try to simplify the process of creating complex plots. One of these core libraries is `Bokeh`⁹ with `pandas-bokeh`, `holoviews` or `hvplot` as some of its extension libraries. You'll use `hvplot` in this section, but you can head over to pyviz.org if you want to read more about the other visualization libraries.

If you followed the instruction in chapter 2, you already have `hvplot` installed. If not, then you'll have to use the Anaconda Navigator and install it for the code examples ahead to work on your computer.¹⁰

Interactive sales plot

Because `pandas` is *the* data analysis tool in Python, most visualization libraries are compatible with `DataFrame` objects, and some even try to copy the `pandas plot` method. The `hvplot` library adds a method called `hvplot` to `DataFrame` objects¹¹ that is similar to the default `pandas plot` method you used in the previous section. To use the `hvplot` method on a `pandas DataFrame`, you first need to `import` the `hvplot` library in your notebook by running:

```
In [15]: import hvplot.pandas
```

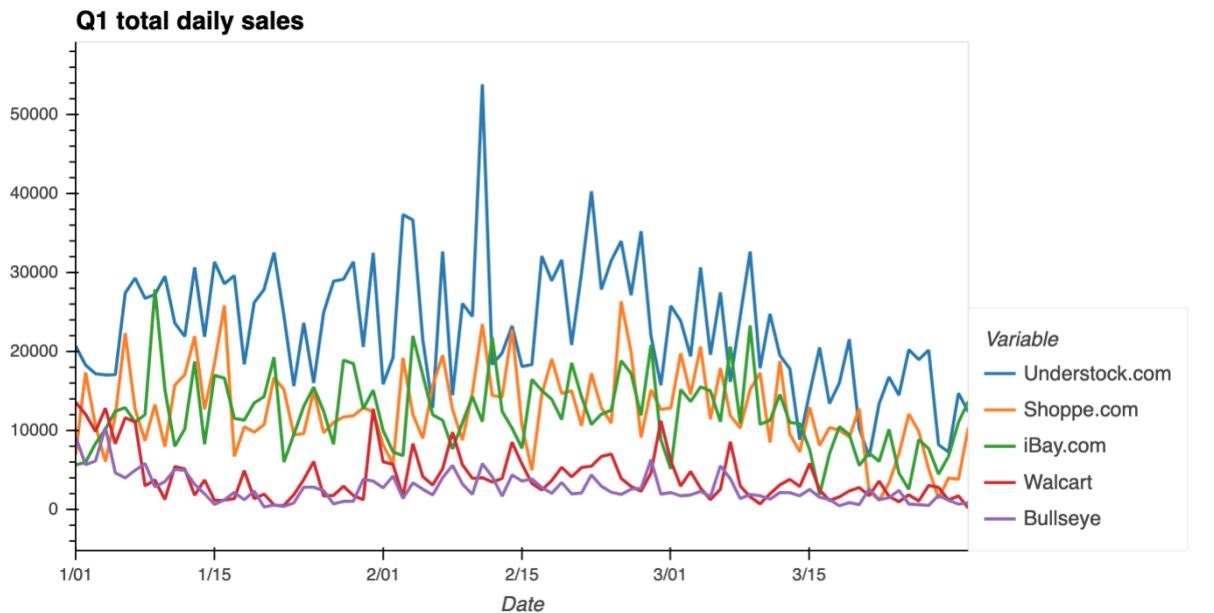
Behind the scenes, this `import` statement temporarily adds the `hvplot` method to all `DataFrame` objects you create (temporarily because `pandas`'s code hasn't changed; if you restart your notebook, `DataFrame` objects won't have the `hvplot` method anymore unless you re-run the `import` statement again). You can now use the `hvplot` method to create an interactive sales plot with:

```
In [16]: daily_sales_df.hvplot(  
    kind='line',  
    width=800,  
    height=400,  
    title='Q1 total daily sales'  
)
```

9: More on Bokeh at bokeh.org.

10: If you're using the Python for Accounting remote workspace, it's already available, you don't have to install it yourself.

11: This is because `pandas` can be extended with custom functions. How you do that is explained at pandas.pydata.org/pandas-docs/stable/development/extending.html.



This code example is very similar to using `pandas`'s `plot` method, but the output is an interactive Bokeh plot instead of a `matplotlib` plot.¹²

On the right, there are several controls you can use to interact with the plot: by default, the pan and hover tools are enabled, but you can also zoom, save or reset the plot to its original position. You can also click on legend items to enable or disable their corresponding elements on the plot (in this case, enable or disable lines).

The keyword arguments available with `hvplot` are not the same ones available with `pandas`'s `plot` method, but they are similar. The `hvplot` method can be used as an accessor as well, just like `pandas`'s `plot` method — if you want to learn what keyword arguments you can use with it, run `daily_sales_df.hvplot.line?` in a separate code cell.¹³

There are several other types of plots available with `hvplot`: type `daily_sales_df.hvplot.` in a separate code cell and press TAB to get a list of available plotting methods or visit the `hvplot` gallery at hvplot.holoviz.org/reference for more examples.

12: If you don't run the code above in your own notebook, you'll have to believe me that the output is an interactive plot.

13: Alternately, you can read more about `hvplot` options at hvplot.holoviz.org/user_guide.

Summary

This chapter showed you how to use `pandas` and `seaborn` to create plots from `DataFrame` objects. Both `pandas` and `seaborn` extend `matplotlib` to make plotting code shorter and easier to understand. Because they're extensions of `matplotlib`, you can still use `matplotlib` functions to customize `pandas` or `seaborn` plots.

The last part of the book starts next: in the chapters ahead, you'll put together all the Python tools you've been reading about to complete a typical management accounting analysis project.

SALES ANALYSIS PROJECT

PART FOUR

The previous chapters introduced some of Python’s most popular data handling tools. In real-world data analysis projects, you will almost always need to use several of these Python tools together, and knowing how to do that effectively is as much of a skill as writing code.

In this last part of the book, you’ll go through an end-to-end data analysis project that uses Python, Jupyter notebooks, `pandas`, and a few other Python libraries. On the way, you’ll also see how to handle the parts of a project that aren’t about writing code: setting goals for the analysis, organizing files, preparing data, finding answers, and sharing results.

The project you’ll go through is a typical management accounting analysis that looks at sales profitability. Even if your work doesn’t involve analyzing sales or inventory, you’ll find many ideas in the chapters ahead to help you organize your own projects.

In your *Python for Accounting* workspace, you should have a “*P4-Sales analysis*” folder. In this folder, you’ll find the data files needed for the sales analysis project:

- ▶ “*Q1Sales.xlsx*” through “*Q4Sales.xlsx*” are Excel files containing quarterly sales data from our wholesale supplier — you’ve been using “*Q1Sales.xlsx*” throughout the previous chapters, so you’re already familiar with these data.
- ▶ “*products.csv*” is a dataset containing additional information about each product sold by the wholesale supplier, including its brand and category labels;
- ▶ “*standard costs.csv*” is a Standard Cost of Goods dataset with costing information about each product.

You’ll explore the data files more in the following chapters. When you’re ready, let’s set up the project and define its goals.

Setting up your project

When starting a new project, it can be tempting to jump into writing code and wrangling data straight away. However, having a project structure and clearly defined goals for your analysis (that are well documented) will help you stay on track when writing code.

This short chapter looks at how you can keep your project notebooks and data files organized and how you can use Markdown cells in Jupyter notebooks to document your analysis goals.

Files and folders

All data analysis projects spread across multiple files (e.g., they source data from several files, use multiple notebooks, generate several presentations or reports, etc.). Keeping files organized needs practice, regardless of the tools you use to handle them. In this section, you'll find a few tips for keeping your data and code files organized when working with Python code.

Data files

It's often a good idea to keep your data files in a separate folder in your project workspace. This keeps your top-level project folder from getting cluttered with too many files and makes it easier for you to recognize which file does what (or contains what data).

Right now, the “*P4 - Sales analysis*” contains several Excel and CSV files. To move these files into a separate folder, create a new folder by clicking the `New Folder` button  in JupyterLab (which you'll find right above the file navigator) and name it `data`. Now you can move all the data files into the `data` folder, either with drag-and-drop or cut-and-paste, directly in JupyterLab.

At times you'll need to write new data files that can be used in various parts of your analysis. For instance, after cleaning and reshaping a dataset, you can save the clean dataset to a new file in the `data` folder so you can re-use it in other project notebooks without repeating all the cleaning steps. You can organize your data files further, by creating sub-folders in the `data` folder to distinguish between the data files you started with (i.e., files from external sources) and data files you created during the analysis. However, for this project, let's keep all data files in the `data` folder.

Jupyter notebooks

While you can develop your entire project in a single Jupyter notebook, it's common to break analysis code into several notebooks, each one performing a specific part of the analysis. For example, you can have a notebook for data cleaning and preparation, one for the actual analysis, another for making plots, etc.

The reason for breaking code up into separate notebooks is that it keeps conceptually related code together in a single file and makes it easier for you to understand where different parts of your analysis are located. It also makes it easier to re-run specific parts of your code when you need to (without having to run parts that are not required, such as data cleaning, if you already saved a clean dataset in your `data` folder).

You can keep Jupyter notebooks in your main project folder (i.e., not in a separate folder) and you can use a simple naming trick to keep them organized. In your top-level project folder, create a new notebook¹ and name it `"01 - Setup"`. You'll use this notebook to document the business goals for the sales analysis in the following section. The `01` at the beginning of the file name helps identify this notebook as the project's first notebook. You'll increment this indicator for the next notebooks you create (e.g., the second project notebook will be called `"02 - Data preparation"`). This simple naming trick not only keeps notebooks nicely listed in your file explorer, but it also indicates the order in which notebooks need to run to reproduce your project results.

1: Either by going to `File -> New -> Notebook` or through the `New Launcher` button.

Common code

When working with multiple Jupyter notebooks, you'll sometimes need to share code between notebooks (e.g., a variable or a custom function that you defined in one of your notebooks but need in the other ones). While you can copy-and-paste code that you want to re-use, this can quickly become a problem. For instance, if you modify a custom function that you copied elsewhere, you need to copy and paste it again in the other notebooks that need it. However, there's an easier way to share code among notebooks: you can put code that you want to share in a separate Python file in your project, from where all your project notebooks can `import` it.

A common code file is just a text file with a `.py` extension (i.e., a Python file). To create a common code file, go to `File -> New -> Text File` and rename it to `common_code.py` (you can give it any name you want to, but the file extension has to be `.py`). Open the file, add the following line of Python code and save it:

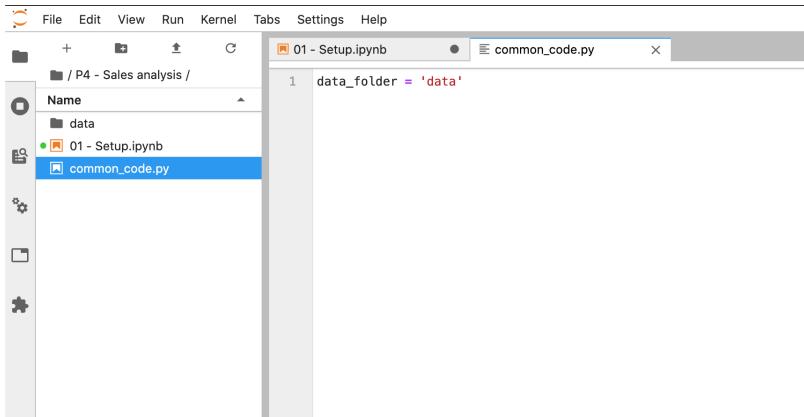


Figure 33.1: Your sales analysis project setup should be similar to the figure shown here.

```
data_folder = 'data'
```

The `data_folder` string variable above points to your project's data folder; you'll use this variable later in this chapter to construct a full path to your data files. Now you can `import` this variable (or any other Python code in the `common_code.py` file) in your project notebooks using:

```
from common_code import data_folder
```

The benefit of having the data folder path defined in the `common_code.py` file (rather than in every notebook that uses it) is that if you ever want to move your data files to a different location, you can update this variable and your notebooks will work just the same, without having to modify their code. Later in this chapter, you'll also add a custom function to this file, which you can then similarly `import` and re-use in any of your project notebooks.

If everything went smoothly, your JupyterLab interface should look similar to figure 33.1.

Documenting your analysis

The first step in any data analysis project is having clear goals and questions you want to answer. It can be tempting to jump into writing code straight away, but without clear goals, you can quickly get drawn into a cycle of slicing, pivoting and plotting data that doesn't lead anywhere (especially if you have lots of data to work with).

One of the advantages of using Jupyter notebooks (over analyzing data in Excel) is that you can add free-form text to describe and document your analysis using Markdown cells. Figure 33.2 shows a refresher of Markdown commands you can use to style text.

In general, it's a good idea to create a few Markdown cells at the top of every notebook you create to describe what the notebook

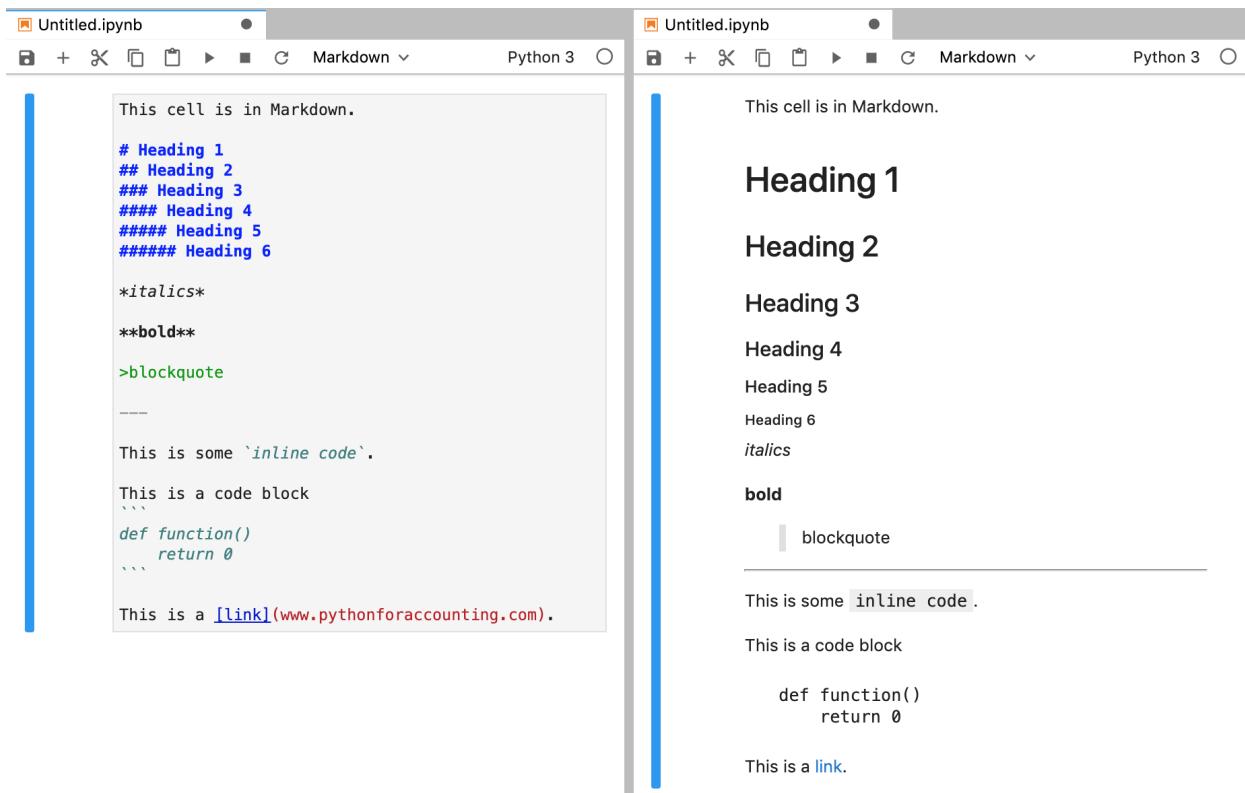


Figure 33.2: An almost complete Markdown reference for Jupyter notebooks. This reference shows side-by-side views of the same notebook in JupyterLab: the left view shows the Markdown text in edit mode, whereas the right view shows the rendered Markdown.

does. However, for larger projects like our sales analysis, you can even keep an entire notebook for documenting the project, which is what we'll do here. Not all projects follow the same pattern, but in general, you can document several aspects of your data projects before writing any actual code:

- ▶ **Description** – what the project is about, why it's needed, and who the key stakeholders are;
- ▶ **Questions** – a list of specific questions you want to answer through your analysis;
- ▶ **Metrics** – a list of metrics (e.g., *gross profit*) that are used in the analysis and how they're calculated;
- ▶ **Data sources** – what datasets you use in the project and how they're obtained.
- ▶ **Timelines** – a list of events related to the project and their description (e.g., when you started, when you last updated the code, how often the analysis is repeated, and why).

To document your sales analysis, open the “01 - Setup” Jupyter notebook you created earlier, add a new cell and change its type to Markdown;² add the following text to the Markdown cell:

```
# Sales analysis project
```

2: By pressing **m** in Command mode, or by selecting **Markdown** from the cell type dropdown menu in the toolbar right above the cell.

For the upcoming board meeting, the CFO is asking for an in-depth analysis of sales performance in 2020. In particular, management wants to know which sales channels and which product categories are most profitable.

This project brings together data from different sources (accounting, marketing, sales), and looks at sales profitability across channels and product categories.

Questions

1. Which sales channels are most profitable?
2. Which product categories are most profitable?
3. Are there differences in category profitability across sales channels?
4. What products are most/least profitable in each category?

Metrics

To determine product profitability, I use `Gross Profit` and `Margin per Unit` as profitability metrics. For each line item in the sales data, the following metrics are calculated:

1. `Gross Profit = Sales Amount - (Unit Cost * Quantity)`
2. `Profit per Unit = Gross Profit / Quantity`
`Margin per Unit = (Profit per Unit / Unit Price) * 100`

The `Margin per Unit` is a percentage (i.e., a value from 0 to 100).

Data sources

- Quarterly sales data is stored in Excel files exported from SAP (`Q1Sales.xlsx` through `Q4Sales.xlsx`);
- `products.csv` is a dataset containing information about all products, including their brand and category labels;
- `standard costs.csv` is a Standard Cost of Goods dataset with costing information about each sold product.

Timelines

- 1/13/2021 - Started project
- 2/18/2021 - Results deadline

When you run this cell, you should see the text displayed inline and styled based on the Markdown styling commands you used.

Documenting³ your analysis this way not only helps you stay on track when writing code, it also helps you when you return to the analysis to understand the details of your project.

Not all projects require this setup: if you just need to combine several Excel workbooks into a single one, there are no analysis metrics to describe — but you might still want to document where the original workbooks come from, why you are combining them, and who is interested in the combined output.

3: Documenting your analysis is an iterative process. Often the questions you want to answer change as the project progresses, and new questions come up. As your project evolves, make sure you return to your documentation and update it often, as this keeps your project organized and your analysis goals in focus.

Summary

This short chapter showed you a few tricks you can use to keep your data files, notebooks, and common code organized when analyzing data with Python. It also showed you how to use Markdown cells to document your analysis goals. Documenting your analysis keeps your project goals in focus and helps you stay on track when writing code.

With clear questions for your sales analysis written down, let's move on to cleaning and preparing the sales data next.

Preparing data

The second step in our sales analysis is preparing data: exploring the available datasets, cleaning and fixing issues with any of them, and merging them into a single working dataset.

For this step of the analysis, create a new notebook in your project folder and name it “02 - Data preparation” — you’ll use this notebook to explore and prepare data using pandas code. But before writing any code, let’s add a title and description to this notebook. Just like in the previous chapter, create a new cell at the top of the notebook, set its type to Markdown, and add the following content:

```
# Exploring and preparing sales data
```

```
This notebook prepares the original sales data, including costs and product datasets, for the analysis process.
```

Now in a new code cell, `import` pandas and the `os` Python module.

Also, `import` the `data_folder` variable from the `common_code` file you created earlier — you’ll use the `os` module together with the `data_folder` variable to list and read Excel files from the data folder later:

```
In [1]: import os  
       import pandas as pd  
  
       from common_code import data_folder
```

Product and cost data

To determine which product categories are most profitable, you need to use the products dataset (the sales data includes product IDs and product names, but not product categories). The products dataset is stored as a CSV file in the data folder; you can read it with:

```
In [2]: products_df = pd.read_csv(f'{data_folder}/products.csv')  
  
products_df.head()
```

```
Out [2]:   ProductID      Product Name        Brand      Category  
0  MI/SNA-81654  Snark SN-5 Tuner fo...    Snark  Musical Instruments  
1  MI/STU-67796  Studio Microphone M...  Generic  Musical Instruments  
2  MI/MUS-73312  Musician's Gear Tub...  Musician's Gear  Musical Instruments  
3  MI/STR-01505  String Swing CC01K ...  String Swing  Musical Instruments  
4  MI/DUN-82082  Dunlop 5005 Pick Ho...  Jim Dunlop  Musical Instruments
```

Notice that the full file path is constructed with the `data_folder` variable you imported from `common_code`, using a Python *f-string* — if you don't remember what a Python f-string is, head back to chapter 5 for a quick refresher.

In general, it's a good idea to inspect new datasets for missing values or irregularities right after you read them:

```
In [3]: products_df.info()
```

```
Out [3]: <class 'pandas.core.frame.DataFrame'>
RangeIndex: 47594 entries, 0 to 47593
Data columns (total 4 columns):
 #   Column      Non-Null Count  Dtype  
--- 
 0   ProductID   47594 non-null   object 
 1   Product Name 47594 non-null   object 
 2   Brand        47594 non-null   object 
 3   Category     47594 non-null   object 
dtypes: object(4)
memory usage: 1.5+ MB
```

These data seem clean, but the `products_df` DataFrame contains a '`Brand`' column we won't be using later, so you can drop it now:

```
In [4]: products_df = pd.read_csv(f'{data_folder}/products.csv')
products_df = products_df.drop('Brand', axis='columns')

products_df.head()
```

```
Out [4]:    ProductID          Product Name           Category
0  MI/SNA-81654  Snark SN-5 Tuner for Gui...  Musical Instruments
1  MI/STU-67796  Studio Microphone Mic Wi...  Musical Instruments
2  MI/MUS-73312  Musician's Gear Tubular ...  Musical Instruments
3  MI/STR-01505  String Swing CC01K Hardw...  Musical Instruments
4  MI/DUN-82082  Dunlop 5005 Pick Holder,...  Musical Instruments
```

To read the costs dataset, you can use the same `read_csv` function as above:

```
In [5]: costs_df = pd.read_csv(f'{data_folder}/standard costs.csv')

costs_df
```

```
Out [5]:    ProductID\tFOB\tDuty\tFreight\tOther\tStandard Unit Cost
0      A/DAN-29859\t2.85\t0.49...
1      A/DAN-94863\t2.97\t0.54...
2      AC&S/10X-13891\t8.47\t1....
3      AC&S/4 N-48073\t2.24\t0....
4      AC&S/ALV-45850\t14.12\t2...
...
3576   T&G/ZIN-76306\t2.18\t0.3...
3577   T&G/^DE-22075\t5.67\t0.9...
3578   VG/NIN-87997\t12.38\t2.0...
3579   VG/SEG-25084\t8.55\t1.42...
3580   VG/SKQ-07575\t9.57\t1.6...
```

```
[3581 rows x 1 columns]
```

The output above is not very useful because the file values are separated by a tab character, not a comma. You need to tell pandas to use the tab character as a value separator by specifying the `sep` keyword argument:

```
In [6]: costs_df = pd.read_csv(f'{data_folder}/standard costs.csv', sep='\t')
```

```
costs_df
```

```
Out [6]:      ProductID    FOB   Duty   Freight   Other   Standard Unit Cost
0           A/DAN-29859  2.85  0.49    0.25    0.29            3.88
1           A/DAN-94863  2.97  0.54    0.08    0.48            4.07
2          AC&S/10X-13891  8.47  1.48    0.28    1.51           11.74
3          AC&S/4 N-48073  2.24  0.40    0.19    0.32            3.15
4          AC&S/ALV-45850 14.12  2.42    1.08    2.30           19.92
...
3576        T&G/ZIN-76306  2.18  0.38    0.07    0.42            3.05
3577        T&G/^DE-22075  5.67  0.99    0.08    0.56            7.30
3578        VG/NIN-87997 12.38  2.04    0.78    1.40           16.60
3579        VG/SEG-25084  8.55  1.42    0.61    0.33           10.91
3580        VG/SKQ-07575  9.57  1.60    0.75    0.66           12.58
```

```
[3581 rows x 6 columns]
```

The costs dataset contains a cost breakdown for each product (e.g., duty, freight, other costs), but we are interested in using just the standard unit cost for this analysis. As with the previous datasets, you can keep the cost information you need and rename the '`Standard Unit Cost`' column to something shorter:

```
In [7]: costs_df = pd.read_csv(f'{data_folder}/standard costs.csv', sep='\t')
costs_df = costs_df.rename(columns={'Standard Unit Cost': 'Unit Cost'})
costs_df = costs_df[['ProductID', 'Unit Cost']]
```

```
costs_df
```

```
Out [7]:      ProductID  Unit Cost
0           A/DAN-29859     3.88
1           A/DAN-94863     4.07
2          AC&S/10X-13891    11.74
3          AC&S/4 N-48073     3.15
4          AC&S/ALV-45850    19.92
...
3576        T&G/ZIN-76306     3.05
3577        T&G/^DE-22075     7.30
3578        VG/NIN-87997    16.60
3579        VG/SEG-25084    10.91
3580        VG/SKQ-07575    12.58
```

```
[3581 rows x 2 columns]
```

Calling `info` on `costs_df` shows that there aren't any missing values in its columns, so you can use it as-is:

In [8]: `costs_df.info()`

```
Out [8]: <class 'pandas.core.frame.DataFrame'>
RangeIndex: 3581 entries, 0 to 3580
Data columns (total 2 columns):
 #   Column      Non-Null Count  Dtype  
---  --          -----          --    
 0   ProductID   3581 non-null    object 
 1   Unit Cost   3581 non-null    float64
dtypes: float64(1), object(1)
memory usage: 56.1+ KB
```

These datasets need to be merged with the actual sales data, but first, you need to read and fix any issues with the sales data themselves, which is what the following section looks at.

Sales data

This project's sales data is stored in four Excel files in the `data` folder, one file for every quarter in 2020; each of these files has three separate sheets, one for every month in its respective quarter. If you open either of them in Excel, you'll see something similar to figure 34.1.

	A	B	C	D	E	F	G	H	I	J	K	L
1	InvoiceNo	Channel	Product Name	ProductID	Account	AccountNo	Date	Deadline	Currency	Unit Price	Quantity	Total
2	61248	Understock.com	Hosa Dual-Channel F M/HOS-95885		Sales	5004	2020-07-01	06/01/20	USD	5.4	1	5.4
3	61249	Understock.com	T&G/LEA-9718		Sales	5004	2020-07-01	6-10-20	USD	7.41	5	37.05
4	61250	Shoppe.com	Neewer LAN Networ E/NEE-41490		Sales	5004	2020-07-01	2020/05/01	USD	6.51	3	19.53
5	61251	Shoppe.com	Chauvet Bubble Flui M/CHA-42247		Sales	5004	2020-07-01	05/02/20	USD	32.32	3	96.96
6	61252	iBay.com	LEGO Monster Fight T&G/LEG-58079		Sales	5004	2020-07-01	10-19-20	USD	5.55	2	11.1
7	61253	Understock.com	Eson 6x 194 168 282 M/ESO-72759		Sales	5004	2020-07-01	12/05/20	USD	4.37	8	34.96
8	61254	iBay.com	Electro-Harmonix P/M/ELE-12568		Sales	5004	2020-07-01	07/09/20	USD	13.6	11	149.6
9	61255	Shoppe.com	CP&A/BLU-94682		Sales	5004	2020-07-01	Wed Sep 2 00:00:00 USD		28.19	17	479.23
10	61256	Understock.com	Norpro 3398 Oven L K&D/NOR-97091		Sales	5004	2020-07-01	September 02 2020	USD	9.38	3	28.14
11	61257	Shoppe.com	New Samsung Chron E/NEW-55841		Sales	5004	2020-07-01	2020/08/10	USD	14.02	6	84.12
12	61258	Walcart	Minecraft Sheet Mag T&G/MIN-47994		Sales	5004	2020-07-01	4-14-20	USD	6.24	8	49.92
13	61259	Shoppe.com	The Original Pez Dri M/THE-56420		Sales	5004	2020-07-01	04/30/20	USD	6.69	7	46.83
14	61260	Understock.com	Now Foods Mood Su H&P/C/NOW-13370		Sales	5004	2020-07-01	Mon Jun 1 00:00:00 USD		13.34	1	13.34
15	61261	Understock.com	Battat Take-A-Part AI T&G/BAT-55701		Sales	5004	2020-07-01	9-21-20	USD	16.94	31	525.14
16	61262	iBay.com	K&D/Off-93349		Sales	5004	2020-07-01	June 06 2020	USD	36.14	11	397.54
17	61263	Understock.com	Lanikai Concert Uku M/LAN-88069		Sales	5004	2020-07-01	09/28/20	USD	5.85	14	81.9
18	61264	iBay.com	M/MUS-46070		Sales	5004	2020-07-01	05/25/20	USD	2.52	1	2.52
19	61265	Walcart	K&D/SPR-26924		Sales	5004	2020-07-01	June 07 2020	USD	2.94	4	11.76
20	61266	Understock.com	Peerless Performer M/M/PEE-12318		Sales	5004	2020-07-01	08/09/20	USD	3.98	2	7.96
21	61267	Understock.com	PL&G/STI-57850		Sales	5004	2020-07-01	08/01/20	USD	28.77	6	172.62
22	61268	Understock.com	Melissa & Doug Flow T&G/MEL-21962		Sales	5004	2020-07-01	8-23-20	USD	2.63	1	2.63
23	61269	Shoppe.com	Resmed Mirage Libe I&S/RES-33305		Sales	5004	2020-07-01	08/26/20	USD	22.33	2	44.66
24	61270	Understock.com	Gryphon Gryphette I&S/GRY-22231		Sales	5004	2020-07-01	10/14/20	USD	7.52	3	22.56
25	61271	iBay.com	M&T/IT-98362		Sales	5004	2020-07-01	September 02 2020	USD	24.75	2	49.5
26	61248	Understock.com	Battat Take-A-Part AI T&G/BAT-55701		Sales	5004	2020-07-01	9-21-20	USD	16.94	31	525.14
27	61273	Shoppe.com	Gibraltar SC-4248 Sj M/GIB-78850		Sales	5004	2020-07-01	07/11/20	USD	14.57	10	145.7
28	61274	Understock.com	Magic: the Gathering T&G/MAG-06636		Sales	5004	2020-07-01	5-18-20	USD	7.75	30	232.5
29	61275	Understock.com	Waltons WM1506 W M/WAL-82700		Sales	5004	2020-07-01	09/23/20	USD	27.19	7	190.33
30	61276	Understock.com	Chicco Red Bullet Ba T&G/CHI-97434		Sales	5004	2020-07-01	9-04-20	USD	43.83	1	43.83
31	61277	Understock.com	On Stage MY120 Cor M/ON-43095		Sales	5004	2020-07-01	12/14/20	USD	13.25	8	106
32	61278	Walcart	HP&C/LF-51688		Sales	5004	2020-07-01	Aug 00:00:00 USD		12.66	25	316.5
33	61279	Understock.com	Yu-Gi-Oh! - Gorz the T&G/YU-55789		Sales	5004	2020-07-01	07-01-20	USD	3.9	1	3.9
34	61280	Shoppe.com	Body Set w/ Hardwa T&G/BOD-83850		Sales	5004	2020-07-01	12-23-20	USD	14.98	13	194.74
35	61281	Understock.com	Philips Sonicare HX5 H&P/PHI-12653		Sales	5004	2020-07-01	Fri Jul 17 00:00:00 2 USD		2.4	5	12
36	61282	iBay.com	3 Collapsible Bowl S K&D/3-C07383		Sales	5004	2020-07-01	July 14 2020	USD	6.41	30	192.3

Figure 34.1: Screenshot of “Q3Sales.xlsx” opened with Excel.

To analyze all sales at once, you need to read data from each file and concatenate it into a single `DataFrame`. Because you need to repeat the same steps for each file (i.e., read the file, select relevant

columns, fill in missing values, etc.), you can define a custom function that performs any data cleaning steps and then use that function with each of the Excel files.

First, let's define a simple function that takes as input a file name (i.e., for any of the sales Excel files), reads data from that file and returns a `DataFrame` with:

```
In [9]: def get_sales(file_name):
    df = pd.read_excel(f'{data_folder}/{file_name}')
    return df
```

As with the previous datasets, the full file path is constructed with the `data_folder` variable you imported from `common_code` and the `file_name` parameter, using a Python *f-string*.

You'll slowly expand this function to include several data cleaning steps, but for now, you can use it to read just one of the Excel sales files containing sales data:

```
In [10]: get_sales('Q1Sales.xlsx')
```

```
Out [10]:   InvoiceNo      Channel      Product Name ... Unit Price Quantity  Total
0          1532  Shoppe.com  Cannon Water Bomb B... ...  20.11     14  281.54
1          1533      Walcart  LEGO Ninja Turtles ... ...  6.70      1   6.70
2          1534      Bullseye           NaN ...  11.67      5  58.35
3          1535      Bullseye  Transformers Age of... ...  13.46      6  80.76
4          1535      Bullseye  Transformers Age of... ...  13.46      6  80.76
...
14049      15581      Bullseye AC Adapter/Power Su... ...  28.72      8 229.76
14050      15582      Bullseye Cisco Systems Gigab... ...  33.39      1  33.39
14051      15583  Understock.com Philips AJ3116M/37 ... ...  4.18      1   4.18
14052      15584      iBay.com           NaN ...  4.78     25 119.50
14053      15585  Understock.com  Sirius Satellite Ra... ...  33.16      2   66.32
```

[14054 rows x 12 columns]

By default, `read_excel` reads data from the first sheet in an Excel workbook. To make the `get_sales` function read all sheets from a file, you need to extend it with:

```
In [11]: def get_sales(file_name):
    df = pd.concat(
        pd.read_excel(f'{data_folder}/{file_name}', sheet_name=None),
        ignore_index=True
    )

    return df
```

When you pass the `sheet_name=None` keyword argument to `read_excel`, instead of a single `DataFrame` you get a dictionary mapping sheet names to `DataFrame` objects. You can concatenate the `DataFrame` objects in this dictionary into a single `DataFrame` by using the `pd.concat` function as above. Now, if you call the `get_sales`

function again, you'll get data from all the sheets in "Q1Sales.xlsx" in a single DataFrame:

```
In [12]: sales_df = get_sales('Q1Sales.xlsx')
```

```
sales_df
```

```
Out [12]:
```

	InvoiceNo	Channel	Product Name	Unit Price	Quantity	Total
0	1532	Shoppe.com	Cannon Water Bomb B...	20.11	14	281.54
1	1533	Walcart	LEGO Ninja Turtles ...	6.70	1	6.70
2	1534	Bullseye		11.67	5	58.35
3	1535	Bullseye	Transformers Age of...	13.46	6	80.76
4	1535	Bullseye	Transformers Age of...	13.46	6	80.76
...
37703	39235	iBay.com	Nature's Bounty Gar...	5.55	2	11.10
37704	39216	Shoppe.com	Funko Wonder Woman ...	28.56	1	28.56
37705	39219	Shoppe.com	MONO GS1 GS1-BTY-BL...	3.33	1	3.33
37706	39238	Shoppe.com		34.76	10	347.60
37707	39239	Understock.com	3 Collapsible Bowl ...	6.39	15	95.85

[37708 rows x 12 columns]

If you call `info` on the `sales_df` variable you'll see it has several columns you won't need for the sales analysis. For instance, if you take a look at the number of unique values in each column in `sales_df`, some of its columns have only one unique value (which means they don't contain useful information for the analysis):

```
In [13]: sales_df.nunique()
```

1

```
Out [13]:
```

InvoiceNo	27679
Channel	5
Product Name	2447
ProductID	2500
Account	1
AccountNo	1
Date	91
Deadline	1713
Currency	1
Unit Price	3041
Quantity	154
Total	8229
dtype: int64	

Besides, there are a few other issues with these data: the 'Product Name' column contains missing values, and there are several duplicate rows in the DataFrame.

```
In [14]: sales_df.isna().sum()
```

```
Out [14]:
```

InvoiceNo	0
Channel	0
Product Name	4566
ProductID	0
Account	0
AccountNo	0

```
Date      0
Deadline 0
Currency 0
Unit Price 0
Quantity 0
Total     0
dtype: int64
```

In [15]: `sales_df.duplicated().sum()`

Out [15]: 176

Because the products dataset (i.e., `products_df`) contains all product names, the easiest way to fix missing values in the '`Product Name`' column is by dropping it entirely and then merging `sales_df` and `products_df` on their common '`ProductID`' column. Similarly, you can merge `sales_df` with the costs dataset to add cost information to the sales data. Extending `get_sales` further to merge datasets, remove duplicate rows, drop unnecessary columns, and convert column data types where possible, you get:

```
def get_sales(file_name):
    df = pd.concat([
        pd.read_excel(f'{data_folder}/{file_name}', sheet_name=None),
        ignore_index=True
    ])

    df = df.drop_duplicates()
    df = df.convert_dtypes()

    df = df[
        'InvoiceNo', 'Channel', 'ProductID',
        'Date', 'Unit Price', 'Quantity', 'Total'
    ]

    df = df.merge(products_df, on='ProductID', validate='many_to_one')
    df = df.merge(costs_df, on='ProductID', validate='many_to_one')
    df = df.sort_values('InvoiceNo')

    return df
```

In [17]: `sales_df = get_sales('Q1Sales.xlsx')`

`sales_df`

	InvoiceNo	Channel	ProductID	...	Product Name	Category	Unit Cost
0	1532	Shoppe.com	T&G/CAN-9...	...	Cannon Wa...	Toys & Games	17.64
29	1533	Walcart	T&G/LEG-3...	...	LEGO Ninj...	Toys & Games	5.84
50	1534	Bullseye	T&G/PET-1...	...	Pete the ...	Toys & Games	10.32
63	1535	Bullseye	T&G/TRA-2...	...	Transform...	Toys & Games	11.90
98	1537	Bullseye	CP&A/3X-0...	...	3x Anti-S...	Cell Phon...	5.13
...
1355	39227	Bullseye	CP&A/MUS-...	...	Music Jog...	Cell Phon...	6.41
28481	39229	Understoc...	K&D/ZER-9...	...	ZeroWater...	Kitchen &...	3.81
4517	39235	iBay.com	H&PC/NAT-...	...	Nature's ...	Health & ...	5.41

21855	39238	Shoppe.com	T&G/MAG-6...	...	Magic: th...	Toys & Games	30.49
15116	39239	Understoc...	K&D/3 C-0...	...	3 Collaps...	Kitchen &...	6.18

[37532 rows x 10 columns]

Notice that the output of `get_sales` now has fewer rows (i.e., 37532 instead of 37708) because you dropped duplicates. Also, row labels are not consecutive anymore because `get_sales` sorts rows based on the '`InvoiceNo`' column.

Let's add two more extensions to this function: a line of text at the top that briefly describes what the function does and some assertions to catch any issues with the data:

```
In [18]: def get_sales(file_name):
    # Reads sales Excel file, cleans and merges it with products and costs
    # data, and returns a new DataFrame.

    df = pd.concat(
        pd.read_excel(f'{data_folder}/{file_name}', sheet_name=None),
        ignore_index=True
    )

    df = df.drop_duplicates()
    df = df.convert_dtypes()

    # keep useful columns
    df = df[
        'InvoiceNo', 'Channel', 'ProductID',
        'Date', 'Unit Price', 'Quantity', 'Total'
    ]

    # merge with products and costs data
    df = df.merge(products_df, on='ProductID')
    df = df.merge(costs_df, on='ProductID')
    df = df.sort_values('InvoiceNo')

    # check for missing values in any of the columns
    # and for sale events spilling across quarters
    assert df.isna().sum().sum() == 0, 'Data contains NA values'
    assert len(df['Date'].dt.quarter.unique()) == 1, 'Data from multiple quarters'

    return df
```

I briefly mentioned the `assert` keyword in one of the earlier project chapters. It is useful in situations like this one, in which you make certain assumptions about your data or your variables, and while they *should* be correct, you want to be alerted if they aren't. In this example, on line 28, you `assert` that `df` does not have any missing values in any of its columns, and on line 29, you `assert` that the values in the `Date` column are all from a single quarter (i.e., each file should only have sales data from one quarter). Both of these assertions are followed by a custom error message — if you use

the `get_sales` function and either of these assertions fails, your code will show an `AssertionError` and the custom error message associated with the assertion that failed (you don't have to add custom messages to your `assert` statements, but they help figure out what went wrong). Assertions are a compact way of checking that what you think about your data is correct; use them often.

In general, you can craft your custom functions to include any data preparation steps you need. For this project, you can now use the `get_sales` function with all the sales data files. First, you need to get a list of all the Excel files that are in the data folder using the `os` module:

```
In [19]: os.listdir(data_folder)
```

```
Out [19]: ['products.csv',
 'Q1Sales.xlsx',
 'standard costs.csv',
 'Q2Sales.xlsx',
 'Q4Sales.xlsx',
 'Q3Sales.xlsx',
 'sales.csv']
```

The `os.listdir` function returns a list of file names (i.e., string values) from a folder path passed as an argument. In this case, the data folder path is stored as a variable that you imported from the `common_code` file.

The output above lists all files in the “`data`” folder, but there are a few files you need to exclude from this list if you want to use them with the `get_sales` function (i.e., all the files with names that don't end in `'.xlsx'`). Luckily, all the items in the list above are Python strings so you can work with them as with any other Python strings — for example, you can use the `endswith` string method and a Python list comprehension to keep only file names that end in `'.xlsx'`:

```
In [20]: [name for name in os.listdir(data_folder) if name.endswith('.xlsx')]
```

```
Out [20]: ['Q1Sales.xlsx', 'Q2Sales.xlsx', 'Q3Sales.xlsx', 'Q4Sales.xlsx']
```

Now that you have a list of files to process, you can go through this list and use the `get_sales` function for each file:

```
In [21]: sales_df = pd.concat(
    [get_sales(name) for name in os.listdir(data_folder) if name.endswith('.xlsx')],
    ignore_index=True
)
```

Here, you use the same Python list comprehension to go through file names in the data folder and call the `get_sales` function for each of the files.¹ The `get_sales` function returns a `DataFrame` each time it is called, so the list created above will hold four `DataFrames`, one for each of the quarterly sales files. Because you don't need

1: You can use a `for` loop instead, but this is just as readable and a more compact way of expressing the same loop logic.

the list of `DataFrame` objects, you can pass it directly to `pd.concat` so that all the `DataFrames` in the list are combined into a single `DataFrame`, which you then assign to the `sales_df` variable. If you inspect `sales_df` now, you'll see it has all rows from each of the separate sales files:

In [22]: `sales_df`

	InvoiceNo	Channel	ProductID	...	Product Name	Category	Unit Cost
0	1532	Shoppe.com	T&G/CAN-9...	...	Cannon Wa...	Toys & Games	17.64
1	1533	Walcart	T&G/LEG-3...	...	LEGO Ninj...	Toys & Games	5.84
2	1534	Bullseye	T&G/PET-1...	...	Pete the ...	Toys & Games	10.32
3	1535	Bullseye	T&G/TRA-2...	...	Transform...	Toys & Games	11.90
4	1537	Bullseye	CP&A/3X-0...	...	3x Anti-S...	Cell Phon...	5.13
...
119164	89228	Shoppe.com	MI/FEN-18728	...	Fender 09...	Musical I...	20.96
119165	89229	Understoc...	C&P/CAS-0...	...	Casio EXI...	Camera & ...	12.04
119166	89232	iBay.com	M&T/LEG-3...	...	Legally B...	Movies & TV	2.69
119167	89233	Shoppe.com	MI/ELE-71119	...	Electro-H...	Musical I...	45.24
119168	89237	Understoc...	T&G/MEL-5...	...	Melissa &...	Toys & Games	2.55

[119169 rows x 10 columns]

You now have all the sales data, clean and in shape, in one `DataFrame` variable. This `DataFrame` contains product and cost information, so you can use it to answer all the analysis questions. The last thing you need to do is compute the gross profit, unit profit, and unit margin for each sale — you can do that with:

```
In [23]: sales_df['Gross Profit'] = (sales_df['Total'] -
                                 (sales_df['Quantity'] * sales_df['Unit Cost']))

sales_df['Profit per Unit'] = sales_df['Gross Profit'] / sales_df['Quantity']
sales_df['Margin per Unit'] = ((sales_df['Profit per Unit'] /
                                sales_df['Unit Price']) * 100)
```

The current notebook is already quite long; adding more code for the actual analysis steps will make it longer and more difficult to handle. Besides, the analysis code you'll write to answer each question is conceptually unrelated to this notebook, which only cleans and prepares the sales data for analysis. In the following chapter, you'll use a separate notebook to write the code that answers the analysis questions.² However, to run that code, you'll need the clean sales data you just prepared. The easiest way to get the clean data in your new notebook is by writing `sales_df` to a file, which you can then read later:

In [24]: `sales_df.to_csv(f'{data_folder}/sales2020.csv', index=False)`

Writing data to CSV files is much faster than writing it to Excel files, which is why I used the `to_csv` function here — but you can use `to_excel` instead if you need an Excel file with the clean data.

2: Why not keep everything in one notebook? You can, but this separation of code into multiple notebooks makes it easier for you to know where different parts of your analysis are and easier to re-run notebooks if you need to, without repeating unnecessary steps.

Summary

This chapter showed you how to clean, reshape, and combine the different data available for the sales analysis project to create a “working” dataset that can answer the project questions. The following chapter shows you how to run the actual analysis, visualize your findings, and share your results with others.

Finding answers

Now that the sales data is ready for analysis and the project goals are clearly defined, the last step in this project is finding answers to the questions we started with.

For this chapter, create a new Jupyter notebook in your project folder and name it “03 - Sales analysis”. You’ll use this notebook to write the `pandas` code that finds answers and visualizes them. As always, it’s a good idea to add a title and a description at the top of your notebooks, so you know what they’re about when you come back to them. At the top of your new notebook, add the following content in a Markdown cell:

```
# Sales analysis 2020

Analysis of channel and category sales profits. The analysis looks at `Gross Profit` and `Quantity` as indicators of sales volume, and `Margin per Unit` as an indicator of sales performance.

**Questions:**

1. Which sales channels are most profitable?
2. Which product categories are most profitable?
3. Are there differences in product category profitability across sales channels?
4. What products are most/least profitable in each category?
```

The description above also includes the analysis questions (from the project setup notebook) to help keep track of the answers we’re looking for.

In this notebook, you’ll use `pandas` and the `seaborn` plotting library, as well as the `data_folder` variable from `common_code`. You can `import` them as before with:

```
In [1]: import pandas as pd
         import seaborn as sns

         from common_code import data_folder
```

You can use Markdown cells to split the notebook into separate sections, keeping your notebook easier to navigate. For example, you can mark a new section in the notebook with the following content in a new Markdown cell:

```
## Sales data
```

And then read in the clean sales data using a code cell:

```
In [2]: sales_df = pd.read_csv(f'{data_folder}/sales2020.csv')
```

Channel profits

The first question you need to answer is which sales channels are most profitable. Three distinct metrics measure different aspects of profitability (i.e., quantity, gross profit, and average margin per unit), so we want to look at each metric for every sales channel. First, let's add a new section to the notebook and briefly explain the question we're trying to answer in this section:

```
## 1. Which sales channels are the most profitable?

- Which sales channels have the highest `Gross Profit`?
- Which sales channels have the highest average `Margin per Unit`?
```

To get these profitability metrics for each channel in the sales data, you can group the sales by channel and then compute the aggregate metrics with:

```
In [3]: channel_profits_df = (
    sales_df
    .groupby('Channel')
    .agg({
        'Quantity': 'sum',
        'Gross Profit': 'sum',
        'Margin per Unit': 'mean'
    })
    .round(3)
    .sort_values('Gross Profit', ascending=False)
)
```

If you inspect `channel_profits_df`, it contains the required aggregate metrics for each channel, sorted by total gross profits:

```
In [4]: channel_profits_df
```

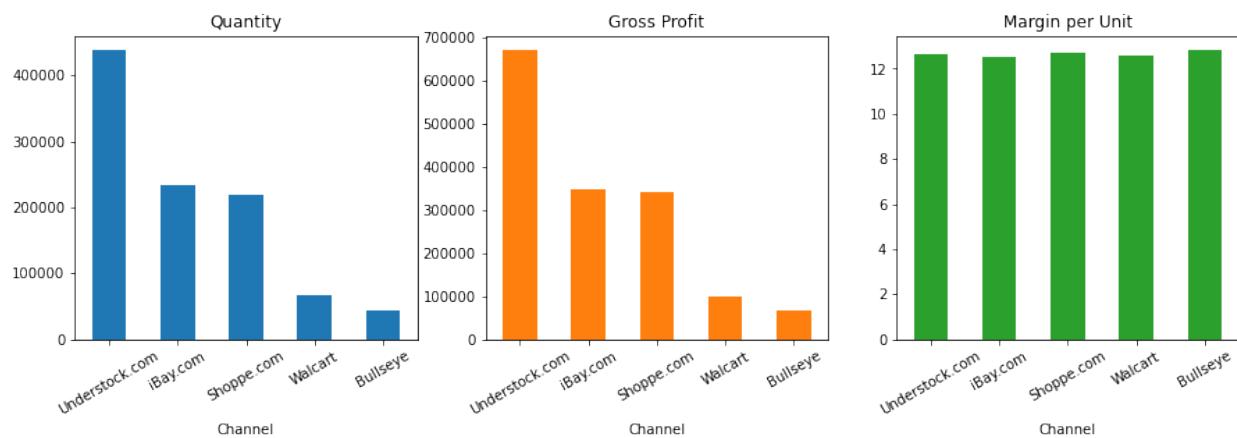
Channel	Quantity	Gross Profit	Margin per Unit
Understock.com	437696	668312.00	12.658
iBay.com	234064	348338.83	12.515
Shoppe.com	219271	339891.68	12.709
Walcart	67302	99492.20	12.591
Bullseye	43645	67410.53	12.803

It seems profits across channels are primarily driven by sales volume (i.e., `Quantity`), given that `Margin per Unit` values are similar across channels. In addition, online channels seem to be more profitable than brick-and-mortar channels.

The table output above gives you the answer you need, but it's always easier to understand data when you visualize it. You can quickly draw bar plots from `channel_profits_df` using the `pandas plot` method:

```
In [5]: channel_profits_df.plot(
    kind='bar', figsize=(15, 4), subplots=True,
    layout=(1, 3), legend=False, rot=30
);
```

1
2
3
4



The `subplots=True` and `layout=(1, 3)` keyword arguments passed to `plot` tell pandas to draw each column in `channel_profits_df` as a separate subplot and that the layout of the entire figure should have one row and three columns.

You can polish these plots further, using the `matplotlib` methods we explored in chapter 30, but even in this rough format, they highlight that sales profits in the two brick-and-mortar retailers (*Walcart* and *Bullseye*) are much lower compared to any of the online retailers, even though average margin per unit is almost identical across channels.

Category profits

The second question you need to answer is similar to the first one, and you can use the same approach as in the previous section, grouping the sales data by `Category` instead of `Channel` to compute the same aggregate metrics:

```
## 2. Which product categories are most profitable?
- Which product categories have the highest `Gross Profit`?
- Which product categories have the highest average `Margin per Unit`?
```

```
In [6]: category_profits_df = (
    sales_df
    .groupby('Category')
    .agg({
        'Quantity': 'sum',
        'Gross Profit': 'sum',
        'Margin per Unit': 'mean'
    })
);
```

1
2
3
4
5
6
7
8

```

    .round(3)
    .sort_values('Gross Profit', ascending=False)
)

```

In [7]: category_profits_df

Out [7]:

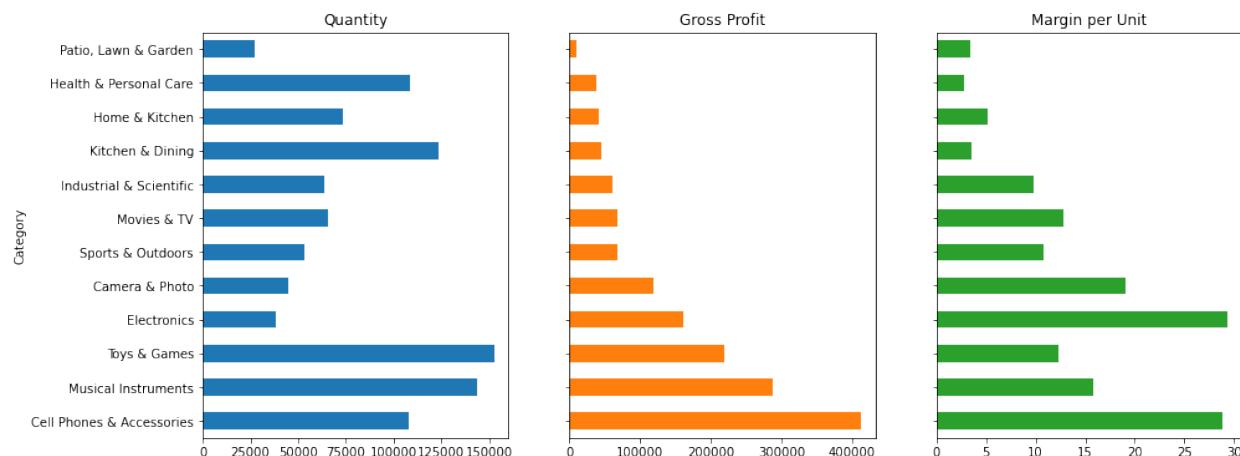
Category	Quantity	Gross Profit	Margin per Unit
Cell Phones & Accessories	107992	411948.43	28.88
Musical Instruments	143751	286849.57	15.76
Toys & Games	152419	217920.67	12.34
Electronics	38238	161513.80	29.38
Camera & Photo	44900	117772.55	19.11
Sports & Outdoors	52998	68542.63	10.74
Movies & TV	65321	66922.81	12.74
Industrial & Scientific	63424	59880.54	9.83
Kitchen & Dining	123573	44745.94	3.50
Home & Kitchen	73636	40594.83	5.17
Health & Personal Care	108765	37434.64	2.77
Patio, Lawn & Garden	26961	9318.83	3.42

You can use the same pandas plotting approach as before to visualize product category profits:

```

In [8]: category_profits_df.plot(
    kind='barh', figsize=(15, 6), subplots=True,
    layout=(1, 3), legend=False, sharex=False, sharey=True
)

```



Because there are a lot of product categories, I made the bars horizontal and increased the figure height (i.e., `kind='barh'` and `figsize=(15, 6)`). I also told pandas to draw the subplots with a common y-axis (`sharey=True`), so that the y-axis tick labels are not repeated for every subplot, but different x-axis values for each subplot (`sharex=False`).

A few categories stand out: *Electronics* and *Camera & Photo* generate relatively low gross profits, even though they have a high average margin per unit. Increasing sales volume for these two categories could increase overall profitability. In contrast, *Patio, Lawn & Garden*

generates almost no profits; removing this category from the product offering could reduce operating costs and increase overall profitability.

Channel and category profits

The third question asks you to look at differences in category profitability across sales channels. To answer this question, you need to compute the average margin per unit for each channel and category combination in the sales data. As with the previous questions, let's add a Markdown cell to mark a new notebook section:

```
## 3. Are there differences between the profitability of product categories across sales channels?
```

More specifically, are there differences between the average `Margin per Unit` across channels?

To get the sales data in the shape we need it, you can use `pd.pivot_table`:

```
In [9]: average_margin_per_category = pd.pivot_table(
    sales_df,
    index='Channel',
    columns='Category',
    values='Margin per Unit',
    aggfunc='mean'
).round(3)
```

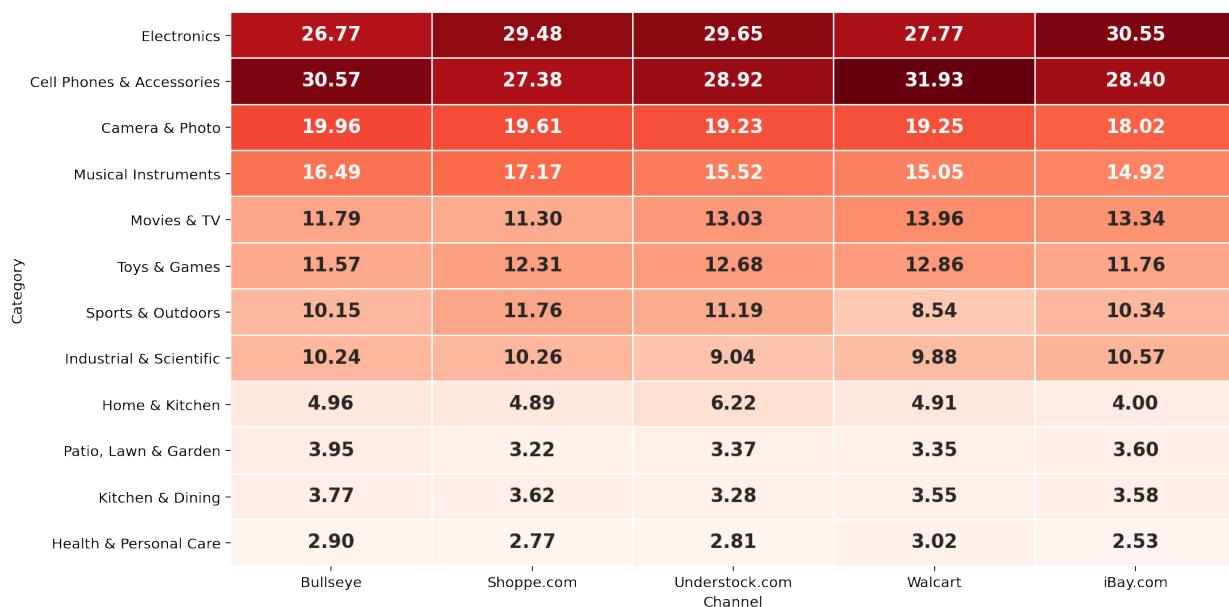
```
In [10]: average_margin_per_category
```

Channel	Bullseye	Shoppe.com	Understock.com	Walcart	iBay.com
Category					
Electronics	26.772	29.480	29.646	27.772	30.546
Cell Phones & Accessories	30.574	27.380	28.920	31.927	28.397
Camera & Photo	19.958	19.609	19.234	19.255	18.015
Musical Instruments	16.490	17.166	15.517	15.055	14.923
Movies & TV	11.791	11.300	13.033	13.964	13.339
Toys & Games	11.570	12.307	12.677	12.863	11.762
Sports & Outdoors	10.151	11.760	11.194	8.539	10.341
Industrial & Scientific	10.239	10.260	9.038	9.876	10.571
Home & Kitchen	4.965	4.886	6.225	4.909	3.996
Patio, Lawn & Garden	3.949	3.217	3.369	3.352	3.601
Kitchen & Dining	3.772	3.620	3.283	3.552	3.584
Health & Personal Care	2.899	2.775	2.809	3.022	2.534

The average margin per category varies across sales channels (e.g., the lowest average margin per unit for the *Electronics* product category is 26%, whereas the highest is 30%). Optimizing some categories by selling products through their most profitable channels might have an impact on total profits.

You can visualize the data above as a heatmap — pandas doesn't have a heatmap plotting function, but you can use `seaborn` instead to create a heatmap from `average_margin_per_category`:

```
In [11]: sns.heatmap(  
    data=average_margin_per_category.transpose(),  
    cmap='Reds',  
    annot=True,  
    fmt=".2f",  
    linewidths=.5,  
    cbar=False  
)
```



Product profits

To answer the last question in this project, you need to find the most and least profitable products in each product category. There are many categories to go through, so manually filtering the sales data for each would require a lot of typing — you can write a custom function to help quickly explore products in each category instead.

First, add a Markdown cell to create a new section, and briefly describe the analysis question:

```
## 4. What products are most/least profitable in each category?
```

- Based on Gross Profit (high/low revenue products).
- Based on Margin per Unit (high/low margin products).

As with the custom function we used to read and clean the sales data, you can start with a simple function to explore products in a

certain category and incrementally expand its features:

```
In [12]: def get_product_profits(df, category):
    return df[df['Category'] == category]
```

This simple function filters a `DataFrame` passed as the first argument on the values in its `Category` column. Because you'll be using this function with `sales_df`, it's safe to assume that the `DataFrame` passed as the first argument has a `Category` column.

You can call this function with:

```
In [13]: get_product_profits(sales_df, 'Toys & Games')
```

```
Out [13]:   InvoiceNo      Channel ... Profit per Unit Margin per Unit
0          1532     Shoppe.com ...
1          1949      Walcart ...
2          5401 Understock.com ...
3          8601 Understock.com ...
4          9860 Understock.com ...
...
87890     114220     Bullseye ...
87894     117532      Walcart ...
87896     118296 Understock.com ...
87897     118330 Understock.com ...
87898     118340 Understock.com ...

[16692 rows x 13 columns]
```

The output is a `DataFrame` containing sales of products from the *Toys & Games* category. However, you need to compute aggregate profits for each product in this table, not just filter sales by category. You can expand `get_product_profits` to compute per-product aggregate profit metrics with:

```
In [14]: def get_product_profits(df, category):
    df = df[df['Category'] == category]

    return (
        df.groupby('ProductID')
        .agg({
            'Product Name': 'first',
            'Unit Price': 'first',
            'Category': 'first',
            'Quantity': 'sum',
            'Gross Profit': 'sum',
            'Margin per Unit': 'mean'
        })
        .sort_values(by='Gross Profit', ascending=False)
        .reset_index()
        .round(3)
    )
```

```
In [15]: get_product_profits(sales_df, 'Toys & Games')
```

Out [15]:

	ProductID	Product Name	Unit Price	Quantity	Gross Profit	Margin per Unit
0	T&G/CHI-38293	Child Construc...	19.68	2864	6963.08	12.385
1	T&G/DIS-87606	Disneys Frozen...	13.02	4240	6955.44	12.458
2	T&G/MEL-91223	Melissa & Doug...	11.44	4323	6115.61	12.373
3	T&G/LEG-28766	LEGO The Lord ...	13.31	3498	5775.40	12.421
4	T&G/GOL-13352	Goldie Blox an...	12.57	3506	5482.67	12.381
..
502	T&G/WL-74772	Wl Products - ...	3.05	4	1.44	11.800
503	T&G/FIS-85290	Fisher Price G...	5.47	2	1.34	12.250
504	T&G/GRE-29463	Green Toys Roc...	8.29	1	1.07	12.910
505	T&G/LEA-17226	Learning Resou...	3.49	1	0.45	12.890
506	T&G/POW-45149	Power Wheels T...	2.10	1	0.26	12.380

[507 rows x 6 columns]

When calling the function, you get a table of unique products (from the category passed as the second argument to `get_product_profits`) and their associated profit metrics. Notice that I keep the `Product Name`, `Category`, and `Unit Price` columns in the output of `groupby` by aggregating them using the `first` function (i.e., for each group, the first occurring value in those columns is used as the result of the aggregation).

The output above is sorted by gross profit, with the highest grossing products in the *Toys & Games* category at the top of the table and the least profitable ones at the bottom. However, if you want to sort the table by margin values instead of gross profits or change the sort order, you need to edit the function again (or sort its output later). You can add a few more parameters to the function (with sensible default values) to make it easier to change the sort column and the sort order:

```
In [16]: def get_product_profits(df,
                           category='All', channel='All',
                           sort_column='Gross Profit', ascending=False):
    if category != 'All':
        df = df[df['Category'] == category]
    if channel != 'All':
        df = df[df['Channel'] == channel]
    return (
        df.groupby('ProductID')
        .agg({
            'Product Name': 'first',
            'Unit Price': 'first',
            'Category': 'first',
            'Quantity': 'sum',
            'Gross Profit': 'sum',
            'Margin per Unit': 'mean'
        })
        .sort_values(by=sort_column, ascending=ascending)
    )

```

```
.reset_index()
.round(3)
)
```

22
23
24

In addition to the `category` parameter, you also added a `channel` parameter to filter products from a certain sales channel, a `sort_column` parameter to set the column to sort by, and an `ascending` parameter to control the sort order when using this function. Each of these parameters has sensible default values, meaning that you don't have to specify values for them when calling `get_product_profits`. However, you can specify values for all (or any subset) of them if you need to:

```
In [17]: get_product_profits(sales_df,
                           category='Toys & Games',
                           channel='iBay.com',
                           sort_column='Margin per Unit',
                           ascending=True)
```

```
Out [17]:      ProductID    Product Name  Unit Price  Quantity  Gross Profit  Margin per Unit
0   T&G/LEG-31282  LEGO City Gre...     2.08       168      40.32      11.54
1   T&G/DUP-28439  DUPLO LEGO Vi...     2.76        15      4.80      11.59
2   T&G/RUB-64023  Rubber Pirate...     2.76       129      41.28      11.59
3   T&G/FUN-21237  Funko Pop! Di...     2.24      1128     293.28      11.61
4   T&G/MY-91022  My Little Pon...     3.70       193      82.99      11.62
..      ...
364  T&G/MY-60452  My Neighbor T...     2.70       176      56.32      11.85
365  T&G/PAW-31908  Paw Print Bal...     3.20       151      57.38      11.88
366  T&G/SWI-62367  Swimways Baby...     1.85        21      4.62      11.89
367  T&G/FUN-29714  Funko Mike My...     2.35       102      28.56      11.91
368  T&G/MAG-22549  Magic: the Ga...     0.06       326      3.26      16.67
```

[369 rows x 6 columns]

The output above answers the question about product profitability for the *Toys & Games* product category — we now need to repeat the steps for all the others. There are many product categories; it might be easier to explore each category's products in Excel. You can use a loop to go through each product category, call `get_product_profits`, and write its outputs to a separate Excel sheet using:

```
In [18]: with pd.ExcelWriter(f'{data_folder}/Product profits 2020.xlsx') as writer:
    for category in sales_df['Category'].unique():
        products_df = get_product_profits(sales_df, category)
        products_df.to_excel(writer, sheet_name=category, index=False)
```

After you run the code above, if you open “*Product profits 2020.xlsx*” (which should be in your `data` folder) with Excel, you'll see it contains one sheet for every product category, with product profits in each sheet sorted by their total gross profit.

You'll need the `get_product_profits` function again, in the notebook that we'll use in the following section. To make the function

available for other notebooks in your project, copy it in the `common_code.py` file you created earlier, then save and close the file.

You now have answers for all the analysis questions we started with (both as tables and as plots). The last step of this project (or any data analysis project) is communicating results with others. In general, depending on how you need to share results, you can either copy-paste tables or plots¹ directly into your reports or presentations or export Jupyter notebooks as HTML files, which you can then share over email or store in a public folder where others can access it. You can also create standalone, interactive HTML files containing Python-based plots and `DataFrame` views. Let's take a quick look at both options next.

1: To copy-paste figures from JupyterLab, you can right-click them while holding down the Shift key, which opens your browser's native context menu, with options to copy or save plots as images.

Sharing results

Sharing your data analysis files (i.e., Jupyter notebooks and data files) with others can be tricky. While data can easily be written to Excel workbooks and shared that way, notebooks with project documentation, analysis steps, figures and all your results are not as straightforward to share.

We'll explore two options for sharing Jupyter notebooks in this section: exporting notebooks as static HTML files and creating a simple, interactive HTML dashboard using the `panel` library.²

Exporting notebooks

JupyterLab allows you to export notebooks to several different formats (including HTML and PDF³) — go to `File -> Export Notebook As...` to see all the available options. Unfortunately, at the time of writing this chapter, the exporting feature in JupyterLab will include all the content from your notebooks in the final output, including all of your Python code, regardless of the format you export to; most likely, you would prefer your exports contained only Markdown, tables, and plots (not code).

An alternative to the default `Export Notebook As...` menu option is to use one of the command-line tools that comes with Anaconda called `jupyter-nbconvert`. With `nbconvert` you can easily remove all input cells from the final export, while keeping Markdown, tables, or plots. For example, to export the sales analysis notebook to a static HTML file, you can run the following command in a code cell:

```
In [19]: !jupyter-nbconvert "03 - Finding answers.ipynb" --no-input --output "Sales analysis.html";
```

2: You need to install the `panel` library yourself using Anaconda Navigator.

3: However, to export Jupyter notebooks to PDF files, there is a caveat: you need to install additional software called `LATEX` for the export to work. `LATEX` is a document preparation tool that can be used to generate PDF files, which JupyterLab uses internally — the details are beyond the scope of this chapter, but you can read more about it at www.latex-project.org.

```
[NbConvertApp] Converting notebook 02 - Sales analysis.ipynb to html  
[NbConvertApp] Writing 393626 bytes to Sales analysis.html
```

This will create an HTML file in your project folder called “*Sales analysis.html*”. If you open it (in JupyterLab or directly in your browser), you will see it has the same content as the sales analysis notebook, but without any of the Python code. If you want to create a PDF export instead, the easiest option is to export your notebook to HTML, and then use your browser’s native print command to print the HTML file to PDF.⁴

The code above is a quick and easy way to generate HTML files that you can share with others from your Jupyter notebooks. The following overthinking section looks at how you can generate a slightly improved HTML file from your Jupyter notebooks by using the `panel` Python library to create a simple, interactive dashboard that you can save and share.

4: On the other hand, if you want to install LATEX on your computer, you can use the same `nbconvert` command to export to PDF directly by changing the output file name to `Sales analysis.pdf`

Overthinking: Creating an interactive dashboard

This project’s last question looks at the most and least profitable products from each product category. While the product tables you produced (i.e., the Excel file with one product category per sheet) contain all the information you need to determine product profitability, it’s not easy to compare products within categories or across channels using these tables. To help compare products with each other, you can visualize product profits using an interactive scatter plot, where each point on the plot is a different product. Even better, you can turn this interactive scatterplot into a dashboard by connecting it to dropdown menus that filter the points shown on the plot by channel or category.

To start building this simple dashboard, create a new notebook in your project folder and name it “*04 - Products dashboard*”. After you create the notebook, open it, and import the tools you’ll be using:

```
In [20]: import pandas as pd  
  
import hvplot.pandas  
import panel as pn  
pn.extension();  
  
from common_code import data_folder, get_product_profits
```

You’ll use `pandas` and `hvplot` for the interactive scatter plot. After you have the plot, you’ll use `panel` to create a simple dashboard, with dropdown menus linked to the `get_product_profits` function to control which product categories or sales channels are shown on the scatter plot. The `pn.extension()` function call on line 5

makes panel components (i.e., the dropdown menus you'll be using) display directly in your Jupyter notebook, which allows you to see how they work and easily tweak them if you need to.

You'll be using the clean sales data in our dashboard; you can read it again in this notebook with:

```
In [21]: sales_df = pd.read_csv(f'{data_folder}/sales2020.csv')
```

Now we can use `get_product_profits` and `hvplot` to create an interactive scatter plot. To link `get_product_profits` to the dashboard menus later on, let's wrap the plotting code inside a new function called `products_scatterplot`:

```
In [22]: def products_scatterplot(category='All', channel='All'):
    df = get_product_profits(sales_df, category, channel)

    return df.hvplot(
        kind='scatter',
        x='Gross Profit',
        y='Margin per Unit',
        size='Quantity',
        color='Category',
        scale=0.2,
        grid=True,
        line_color='black',
        width=900,
        height=600,
        hover_cols=['ProductID', 'Product Name', 'Unit Price']
    )
```

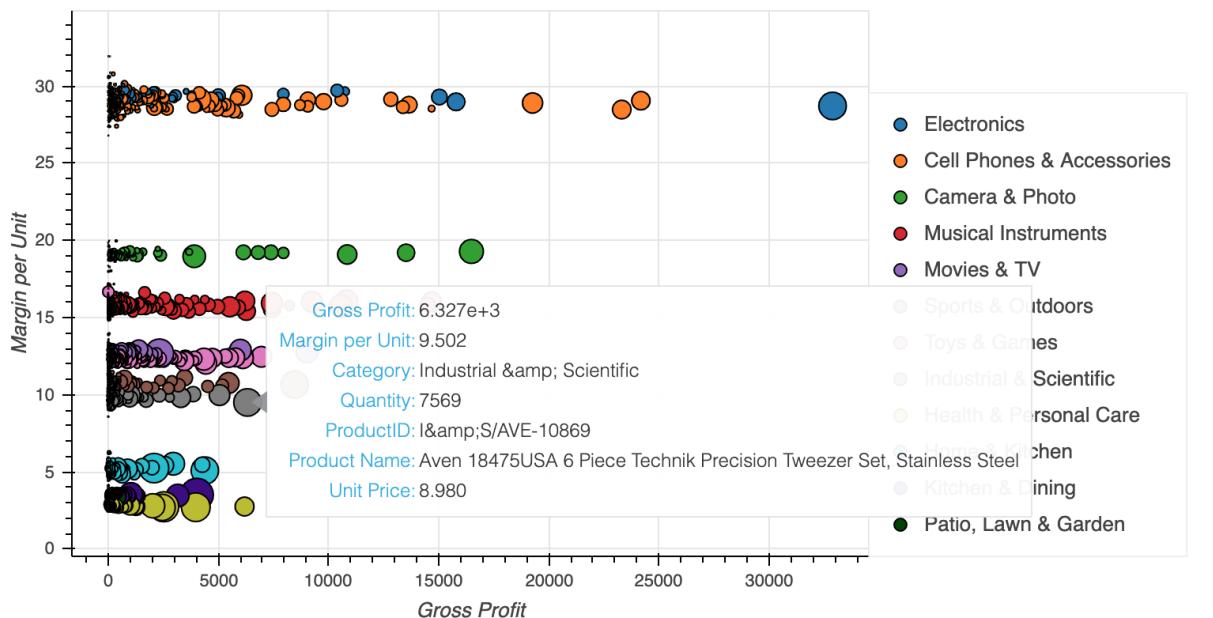
Notice `product_scatterplot`'s parameters get passed to the `get_product_profits` function you created previously, which filters products. Only the subset of products returned by `get_product_profits` is then visualized on the scatter plot. However, by default, the `product_scatterplot` function plots all the products available in the sales data.

For the scatter plot, you're using gross profit as the x-coordinate for each point and the average margin per product as the y-coordinate. You also encode the total sale quantity for each product as the size of the point marker and product category as each point's color. The other keyword arguments passed to `hvplot` control the plot's display in various ways (e.g., `scale=0.2` makes the points on the plot slightly smaller).

Now you can call the function and draw the scatter plot using:

```
In [23]: products_scatterplot()
```

Hover over any point on the plot. You'll see a tooltip with extra information about the product it represents, in addition to its x or y-coordinates (i.e., gross profit and average margin), size, and category. The tooltip information is controlled through the `hover_cols`



argument passed to `hvplot`, which specifies what values from the plotted `DataFrame` to include in the tooltip.

To turn this plot into a dashboard, you need to create two dropdown menus and connect them to `products_scatterplot`'s `category` and `channel` parameters. The `panel` library provides several widgets you can use to connect Python functions to graphical interface elements, including buttons, dropdown menus, or date pickers. To create a simple dropdown menu (using the `Select` panel widget), you can use:

```
In [24]: test_dropdown = pn.widgets.Select(name='Test dropdown', value='a', options=['a', 'b', 'c'])
```

And you can view it directly in your Jupyter notebook using:

```
In [25]: test_dropdown
```

Test dropdown

The keyword arguments passed to `Select` control the label shown above the menu, its default `value`, and the `options` available in the dropdown menu.

This test menu shows how you can use `panel`'s `Select` widget. To make a menu that is useful for the products dashboard, you can delete this `test_dropdown` menu and create a new one that displays different product sales channels. You'll modify the `products_scatterplot` function to use the values selected in this menu through the variable defined below. Because the plotting function will depend on the menu variable, it's a good idea to define them in cells above the `products_scatterplot` function. This way, when you re-start or re-run your notebook, the menu variable is created before

the plotting function; above the `products_scatterplot` function, add the following code cells:

```
In [26]: channels = ['All'] + sorted(sales_df['Channel'].unique())
channel_dropdown = pn.widgets.Select(name='Channel', value='All', options=channels) 1
2
```

```
In [27]: channel_dropdown
```



Notice that you create the options list on line 1, which includes an `'All'` value that will display all available products. The list of options is just a sorted Python list, created directly from the `Channel` column in our sales data, so that you don't have to type the different menu options yourself. You can use the same approach for the product categories, and create a separate menu using:

```
In [28]: categories = ['All'] + sorted(sales_df['Category'].unique())
category_dropdown = pn.widgets.Select(name='Category', value='All', options=categories) 1
2
```

```
In [29]: category_dropdown
```



The two menus don't work yet; you need to link them to the `products_scatterplot` function. To do that, you need to (slightly) change the plotting function, and include the menus and the products scatter plot in a `panel` component. First, add the following line of code above the `products_scatterplot` function:

```
In [30]: @pn.depends(category_dropdown, channel_dropdown)
def products_scatterplot(category='All', channel='All'):
    df = get_product_profits(sales_df, category, channel)

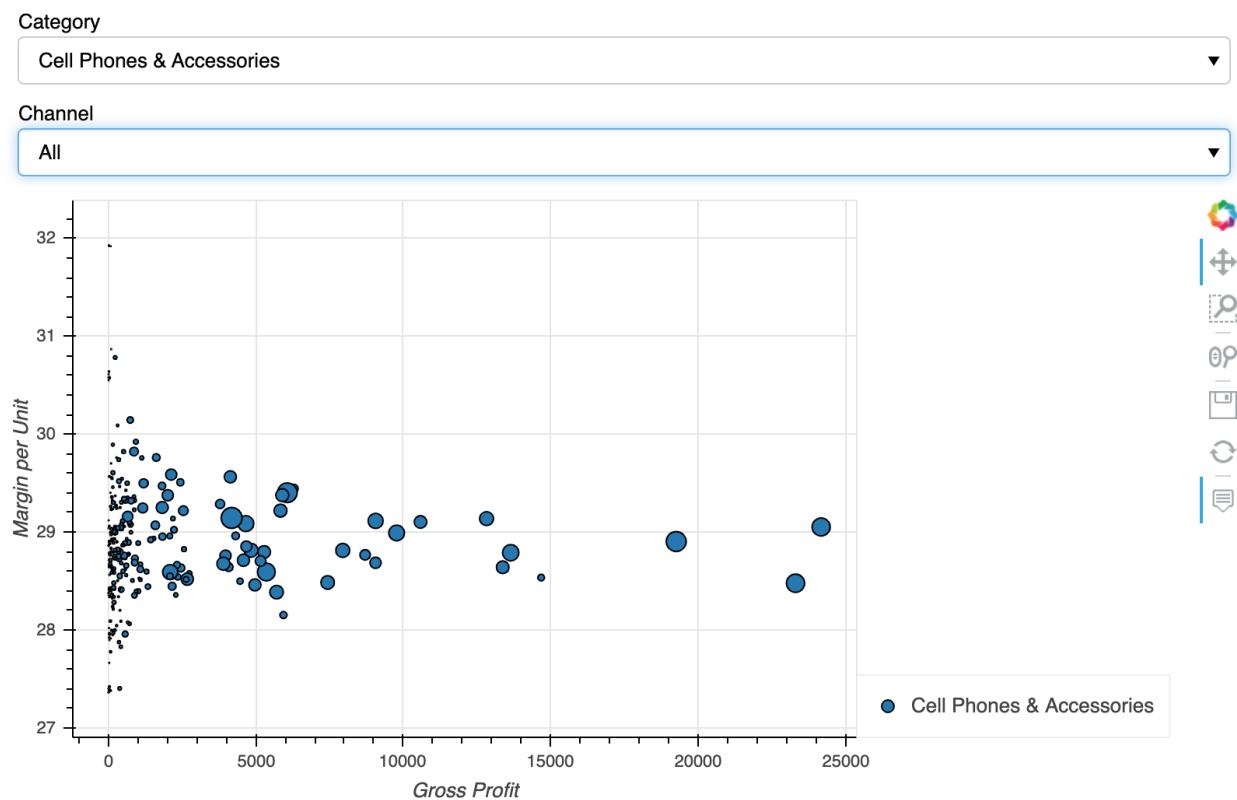
    return df.hvplot(
        kind='scatter',
        x='Gross Profit',
        y='Margin per Unit',
        size='Quantity',
        color='Category',
        scale=0.2,
        grid=True,
        line_color='black',
        width=900,
        height=600,
        hover_cols=['ProductID', 'Product Name', 'Unit Price']
    ) 1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
```

The `@pn.depends` is a *function decorator* provided by the `panel` library. Function decorators are Python language constructs that provide an easy-to-use mechanism to extend functions without modifying them — here, `panel` uses the `pn.depends` decorator to extend the

plotting function and link it to the two dropdown menus behind the scenes.⁵

Finally, to make the plotting function and the menus work together, you need to include them in a `panel` component using:

```
In [31]: pn.Column(  
    category_dropdown,  
    channel_dropdown,  
    products_scatterplot  
)  
1  
2  
3  
4  
5
```



You can now use the dropdown menus to filter data shown on the plot — the output above shows the plot after filtering for *Electronics* products. The `panel` library uses columns or rows to arrange widgets and plots: when placing components inside a `pn.Column`, they will be displayed vertically, one below the other. You can create more complex layouts by nesting rows and columns — more information about `panel` components and layout options is available at panel.holoviz.org/user_guide/Components.html.

You can also include Markdown text and commands inside your `panel` components (as Python strings), and they will be displayed in the final output. Let's add a title to the `panel` component you created above and assign it to a separate variable:

```
In [32]: dashboard = pn.Column(  
    "# Products dashboard",  
    "Select sales channel or product category using the menus below:",  
)  
1  
2  
3  
4  
5
```

5: Decorators are a fairly advanced topic. If you want to learn more about decorators, you can read the official Python documentation available at www.python.org/dev/peps/pep-0318/.

```
    category_dropdown,  
    channel_dropdown,  
    products_scatterplot  
)
```

Now, if you want to view the `dashboard` variable, you can either inspect it in a separate cell or use the `show` method to display the component in a separate browser tab:

```
In [33]: dashboard.show()
```

```
Out [33]: Launching server at http://localhost:52466  
<bokeh.server.server.Server at 0x1a227c65d0>
```

When you run the code above, a new tab should open in your browser, showing the scatter plot and dropdown menus as a separate web page (i.e., outside of your Jupyter notebook). This page still depends on your Jupyter notebook; to view this web page, your notebook needs to be running. However, you can save this page (including the interactive plot and dropdown menus) as a standalone HTML file with the `save` method:

```
In [34]: dashboard.save('Products dashboard.html', embed=True)
```

The first argument to `save` specifies the name (or path) of the file you want to save the dashboard to, and the `embed=True` argument tells `panel` to embed all the different menu options inside the HTML file — in this case, you save the dashboard to a file called “*Products dashboard.html*” in your project folder. When you run the code above, `panel` cycles through different values of the two dropdown menus and saves each generated plot inside the HTML file. Depending on how complex your dashboard is, this process can take some time, but the output is a standalone file that doesn’t depend on your Jupyter notebook and that you can share with others.

This dashboard is simple, but you can extend it to include various widgets and multiple plots (created using `matplotlib`, `seaborn`, or any other Python visualization library). To see more complex dashboard examples built with the `panel` library, visit panel.holoviz.org.

Summary

This chapter showed you how to answer the sales analysis questions using `pandas`. It also briefly showed you how to visualize your results and export them from your Jupyter notebook to HTML files that you can share with others.

This chapter marks the end of the analysis project, and the end of all Python coding in this book. The following chapter wraps

things up and points you to a few resources that can guide your Python-learning further.

Next steps

This concludes our tour of Python. I hope you now have a clear idea of how Python can help in your accounting work.

Programming books tend to resemble horoscopes: for them to be relevant to a wide range of readers, they stay general. In this book, I tried to cover the parts of Python’s data-world that will be useful to anyone working with accounting data, at a suitable level of abstraction and detail. However, there are many types of accounting in the real world, and the tasks you face in your day-to-day need specific solutions. You’ll have to create solutions to your problems with pieces from the Python puzzle — I think you now know how.

As I mentioned at the beginning of the book, Python has a large ecosystem of libraries and tools. Depending on where you want to take your Python learning journey next, below are some libraries you might find interesting.

Tools for automating more of your interaction with Excel

You already know that the `pandas` library can read and write Excel files. However, in the previous chapters, I didn’t mention that in its own code, `pandas` uses other Python libraries that enable it to interact with Excel files. These libraries are `openpyxl` and `xlsxwriter`.⁶ Both of these libraries provide advanced methods for interacting with spreadsheets. For instance, with `openpyxl`, you can set individual cell values in your Excel tables:

```
In [35]: from openpyxl import load_workbook
wb = load_workbook(filename='MyExcelFile.xlsx')

# get first worksheet
ws = wb.active

# set cell values
ws['A2'] = 42
ws['B2'] = 'Hello, openpyxl!'

wb.save('MyExcelFile.xlsx')
```

The alternative library, `xlsxwriter`, has more features than `openpyxl`; you can use it to generate Excel charts, create dropdown menus, or add images to your Excel files. However, you can’t use it to read or modify existing Excel files (unlike `openpyxl`). If you need more powerful interactions with your Excel files, take a closer look at these libraries.⁷

6: Because `pandas` uses these libraries in its code, they’re already installed on your computer.

7: You can read more about `openpyxl` at openpyxl.readthedocs.io, and about `xlsxwriter` at xlsxwriter.readthedocs.io.

Python tools for handling PDF files

If you work with PDF files often, you've probably already wondered if you can use Python to automate some of that work. The answer is yes: several Python libraries can help, but my favorite is `pikepdf`. With `pikepdf` you can edit, slice, transform, and extract text from PDF files. Unfortunately, it's not part of the Anaconda distribution, so you'll have to install it yourself using Anaconda Navigator.⁸

8: You can read more about `pikepdf` at github.com/pikepdf/pikepdf.

Python tools for GUI automation

Even though Python's libraries' ecosystem is vast, if you work with legacy or highly specialized systems, you'll struggle to find a Python library that can interact with them. However, you might find a solution in automating your interface interactions with those systems (i.e., write some Python code that moves your mouse and clicks the right buttons to extract data).

You can use several Python libraries to automate your interface interactions, with `pyautogui` being one of the most popular. The `pyautogui` library lets you use Python to control the mouse and keyboard on your computer and automate interactions with any application. It works on Windows, macOS, and Linux.⁹

9: Read more about `pyautogui` at github.com/asweigart/pyautogui.

The other library you might find interesting is `selenium`. It lets you automate interactions with websites, but instead of taking control of your mouse and keyboard, it runs in the background (so you can still use your computer while it's running). The `selenium` library is more complicated than `pyautogui`, and you'll need to learn more about web programming languages (i.e., HTML, CSS) before you use it. However, it can be a powerful tool if you need to automate browser-based interactions.

Interactive data visualization and dashboards

In part three, I mentioned that the Python ecosystem has many different libraries for data visualization, and it can sometimes be challenging to find the right one for your needs.¹⁰ Two visualization libraries I want to highlight are `plotly` and `bokeh`. Both allow you to create interactive visualizations and dashboards that can be embedded into other web pages, and both are used by other libraries as a plotting foundation (i.e., just like `seaborn` or `pandas` use `matplotlib` as their plotting foundation). The `plotly` library is slightly more popular than `bokeh`, but they are both mature tools — which one to use is a matter of preference.

10: You can read more about all the different Python visualization tools at pyviz.org.

Statistics, machine learning, and forecasting tools

It seems like everyone is talking about data science, machine learning, or artificial intelligence. In part, Python has been driving their popularity because it enables straightforward access to highly specialized machine learning and data science tools.

One of the most popular Python toolkits for statistics is the `scipy`¹¹ library. The `scipy` library is a bundle of functions for mathematics, statistics, and engineering. If you ever need to run statistical tests on your data, you should look up methods in `scipy`'s `stats` submodule.¹²

Similarly, the most popular toolkit for machine learning in Python is the `scikit-learn` library. It is built on top of `scipy` and includes most current machine learning algorithms (from linear regression to neural networks).¹³ However, `scikit-learn` is designed more as an engineering tool than an analytics one. If you need more descriptive statistical modeling (e.g., extensive details about your regression coefficients), you should consider using the `statsmodels` library instead.¹⁴

Finally, if your work involves inventory, sales or other kinds of forecasting, you might find the `prophet` library useful. It provides robust procedures for forecasting time series data with yearly, weekly, and daily seasonality, plus holiday effects.¹⁵

Thanks for reading, have fun with Python!

11: Pronounced "*Sigh Pie*".

12: scipy.org.

13: scikit-learn.org.

14: statsmodels.org.

15: github.com/facebook/prophet.